12-21-2018

STATIC ENFORCEMENT OF TERMINATION-SENSITIVE NONINTERFERENCE USING THE C++ TEMPLATE TYPE SYSTEM

Scott Douglas Constable
Syracuse University

Abstract

A side channel is an observable attribute of program execution other than explicit communication, e.g., power usage, execution time, or page fault patterns. A side-channel attack occurs when a malicious adversary observes program secrets through a side channel. This dissertation introduces Covert C++, a library which uses template metaprogramming to superimpose a security-type system on top of C++'s existing type system. Covert C++ enforces an information-flow policy that prevents secret data from influencing program control flow and memory access patterns, thus obviating side-channel leaks. Formally, Covert C++ can facilitate an extended definition of the classical noninterference property, broadened to also cover the dynamic execution property of memory-trace obliviousness. This solution does not require any modifications to the compiler, linker, or C++ standard.

To verify that these security properties can be preserved by the compiler (i.e., by compiler optimizations), this dissertation introduces the Noninterference Verification Tool (NVT). The NVT employs a novel dynamic analysis technique which combines input fuzzing with dynamic memory tracing. Specifically, the NVT detects when secret data influences a program's memory trace, i.e., the sequence of instruction fetches and data accesses. Moreover, the NVT signals when a program leaks secret data to a publicly-observable storage channel. The Covert C++ library and the NVT are two components of the broader Covert C++ toolchain. The toolchain also provides a collection of refactoring tools to interactively transform legacy C or C++ code into Covert C++ code. Finally, the dissertation introduces libOblivious, a library to facilitate high-performance memory-trace oblivious computation with Covert C++.

STATIC ENFORCEMENT OF TERMINATION-SENSITIVE NONINTERFERENCE USING THE C++ TEMPLATE TYPE SYSTEM

by

Scott Constable

B.A., Ithaca College, 2012

M.S., Syracuse University, 2014

Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer and Information Science and Engineering

Syracuse University

December 2018

For my wife

Acknowledgments

The body of work presented in this dissertation would not have been possible without the help and guidance of numerous other individuals throughout my entire life, and especially over the past couple of years.

I would first like to thank my advisor, Steve Chapin, for giving me the opportunity to join the EECS department at Syracuse University and pursue my graduate studies in computer science. You have shepherded me through my academics for the past six years, and for that I am deeply grateful.

I must thank all of my terrific dissertation committee members: Pete Wilcoxen, Yuzhe Tang, Jim Royer, Jim Fawcett, Shiu-Kai Chin, and Stu Card. All of you have given me tremendously helpful feedback on my thesis work over the past year, and I especially appreciate that you were willing to read and review my dissertation. Special thanks are in order for Pete Wilcoxen, my committee chair, who helped me out with some graduate school paperwork that I did not realize I was required to complete before defending my dissertation.

I thank Critical Technologies Inc. and the CASE at Syracuse University for funding my tuition and stipend for several semesters, and for allowing me to work on some very interesting formal verification projects. I also thank the family of Wilbur R. LePage—especially his wife Mrs. Eveline LePage—for establishing the LePage fellowship, which funded my education at Syracuse University for two full years. Their generosity allowed me to focus on completing my qualifying examinations and broaden my research interests.

Covert C++ began as an internship research project at Intel Labs in Hillsboro, Oregon.

For the opportunity to spend an entire summer developing the first prototype of Covert C++, I am deeply grateful to Intel Corporation. I also received some terrific feedback from the (very brilliant) folks in the security and privacy group at Intel Labs.

I have been blessed with so many wonderful teachers through my life, spanning from kindergarten through graduate school. I had some particularly rough-and-tough teachers who worked me to the bone—and made me stronger and smarter because of it: Ms. Wetherby (AP world history, 9th and 10th grade), Mrs. Quinlan (10th grade trigonometry), Mrs. Paolini (AP language and comprehension, 11th grade), John Rosenthal (abstract algebra, college), and Brian Zimmer (music, trombone). You all taught me the value of hard work, for which I am grateful (even though I'm sure I wasn't back then). I also had two terrific advisers at Ithaca College, Tom Pfaff and David Brown, who helped steer me towards graduate school. Thank you both.

I thank all of my childhood (and adulthood) friends: Kyle, Kenny, Rob, Jeff, Beeks, Mike, and too many others to name. I am an only child, and you guys have always been like brothers to me. Thank you for always being with me through thick and thin.

I would like to thank my parents, Douglas and Kristine Constable, for loving and raising me ever since I was so inconceivably small and helpless. Thank you especially for passing on your love of music to me, which is now a permanent fixture of my life. Many thanks to Ann Toni DuBois for baking I-don't-know-how-many cookies. And thanks to my in-laws, who had been so supportive while I was preparing to defend my dissertation.

Finally, I would like to thank my wife, Fatima. You are the love of my life. Each and every day, you push me to be a better person than I was the day before. Although I frequently fall short, I always appreciate your persistence. Without your unrelenting support and encouragement, there is a non-zero probability that I would have never been able to finish this dissertation.

I have no doubt that there are many more people whose names belong here. If yours is one of those names, please rest assured that I am grateful to you as well.

Contents

List of Figures

List of Tables

List of Theorems

Listings

Chapter 1

Introduction

The recent proliferation of infrastructure as a service (IaaS) has allowed big data computations to be offloaded to the cloud. While this confers computational and cost benefits to small companies, large companies, research organizations, and end users, it has also been plagued by concerns over data privacy. Technologies such as the secure sockets layer (SSL) preserve confidentiality and integrity of sensitive data as it is transferred to or from the cloud. New hardware technologies including Intel SGX [7], AMD SME/SEV [92], and ARM TrustZone [1] employ strategies such as encryption and process isolation to protect data while it being used or stored in the cloud. These techniques are not sufficient to prevent an unintentional leak of sensitive information from the protected hardware environment. Although this problem has been studied intensely for the past several years, security findings made in 2018 have brought public attention to the issue of data privacy in the cloud.

The recently discovered Spectre [96], Meltdown [103], and Foreshadow [48] attacks exploit a common CPU optimization feature called speculative execution. These vulnerabilities are so severe that numerous software products, including operating systems and web browsers, needed to be patched immediately [42, 129, 132]. Moreover, portions of commodity CPUs are still being redesigned with new protection schemes [166]. More recent research has even shown that these pernicious attacks can be launched across networks [139]. One attribute shared by these attacks and numerous other attacks on cloud infrastructure is that they are all designed to circumvent hardware, software, and/or cryptographic protections, with the intent of exposing sensitive data.

One way in which sensitive information can be exposed accidentally is through an unprotected storage channel. Suppose that a software engineer is using a particular API library to run aggregation algorithms over secret program data. The program will be deployed in a protected hardware environment which provides encrypted RAM to protect the program's data. However, the engineer is unaware that the data aggregation API sometimes emits logging information to an unprotected log file located somewhere else on the system. If the data emitted to the log depends on the API inputs containing sensitive data, then that sensitive data could leak to a malicious adversary with access to the log file.

Sensitive information can also be leaked by more subtle means. A *channel* is any mechanism which signals information. A *side channel* is an inadvertent revelation of information by some mechanism of an algorithm's implementation, not attributable to some flaw in the algorithm's specification. When a malicious adversary is able to observe or infer sensitive information through one or more side channels, this is called a *side-channel attack*[1]. Common side-channel attacks include timing analysis attacks (exploiting a timing channel), power analysis attacks (exploiting a power channel), differential fault analysis attacks, and cache-based attacks, among others (e.g., [148, 165]). For example, Kocker et al. [95] observed that the power current drawn from a device running the Data Encryption Standard (DES) cryptographic algorithm could leak enough information to reveal the secret encryption key. The Spectre [96] attack is also an example of a side-channel attack. Other examples are discussed in greater depth in Chapter 2.

Among the most widely studied categories of side-channel attacks have been attacks which infer information leaked through a program's control flow structure [135]. For instance, a conditional branch instruction might leak the value of the branch condition operand, assuming

[1]The original definition of side-channel attack specifically pertained to inadvertent information leakage from a cryptographic function, i.e., to assist in cryptanalysis [94].

```
1 int memcmp(const void *s1, const void *s2, size_t n) {
2   const uint8_t *u1 = s1, *u2 = s2;
3   while (n--) {
4     uint8_t diff = *u1++ - *u2++;
5     if (diff) // 'diff' influences control flow
6       return diff;
7   }
8   return 0;
9 }
```

Listing 1.1: A simple `memcmp()` implementation

the adversary has some means by which to observe whether or not a jump has occurred. If the branch is a loop termination condition, then the number of jumps to the beginning of the loop could be inferred through a timing channel.

Example 1.1 (`memcmp()` Vulnerability)**.** Listing 1.1 gives a sample implementation of the `memcmp` function in C. The `if` condition at Line 5 makes a control flow decision depending on whether or not two bytes at identical offsets in the input buffers match; if they do not, `memcmp` returns a non-zero result. Semantically, the observable effect of any call to `memcmp` is only the return value; this function has no side effects. However the control flow of the program will vary with respect to the input buffers.

Consider the following scenario. A call to `memcmp` is made with two 128-byte buffers, one of which is secret. Suppose the adversary Mary is able to provide the non-secret buffer, and she can trigger a call to `memcmp` as many times as she would please. If Mary can only observe the return value, then she will discover the value of the secret buffer after at most $2^{128\times 8} = 2^{1024}$ attempts. If Mary can also observe the program's control flow, then she can count the number of loop iterations taken before `memcmp` returns. Mary proceeds by attempting all 2^8 possible values for the first byte. When the loop takes one additional iteration, Mary knows she has correctly guessed the first byte. By repeating this process, she will discover the value of the secret buffer after at most $2^8 \times 128 = 2^{15}$ attempts, a reduction in work factor of over 2^{1000}. △

The efficacy of side-channel attacks has been well-demonstrated over the past two decades, especially in cloud computing and the Internet of things [19, 40, 107, 116, 171]. In particular, hardware platforms which support trusted execution environments (TEEs) [14] or isolated execution environments (IEEs) [7] are of interest because their aim is to provide trusted computing capabilities within in an untrusted environment. IEEs such as Intel SGX use hardware-based dynamic DRAM encryption to provide confidentiality [112]. However, encryption alone is insufficient to guarantee confidentiality when a malicious adversary may exert bounded control over server hardware, e.g., by recording a trace of page faults triggered by the victim application [171], or by observing timing channels, power channels, etc.

Many contemporary techniques for defeating or mitigating side-channel attacks are deployed at a low level, for instance by modifying a compiler (e.g., [73, 105, 175]), transforming binaries (e.g., [74, 133]), or dynamically obfuscating memory access patterns (e.g., [40, 136]). These strategies use computationally expensive program trace obfuscation techniques. Hardware-based techniques (e.g., [108]) may introduce less overhead, but are not cross-platform, and typically address only one kind of attack.

At a higher level, recent software-layer solutions have deployed data-oblivious computation strategies. A program is said to be *data oblivious* if the adversary's view of its data memory accesses (i.e., reads and writes) does not reveal sensitive data which may have determined the locations of those accesses [79, 124]. An even stronger property is *memory-trace obliviousness*, which additionally covers the number and order of instruction fetches made by a program [104, 105, 133]. This dissertation sometimes uses the truncated term *obliviousness* to refer to memory-trace obliviousness.

The designation of *secret* versus *public* (non-secret) program data is often subjective. In general, sensitive program data should be treated as secret; non-sensitive program data and program metadata should be treated as public. Sensitive data can include anything from credit card numbers, patient medical histories, and intellectual property, to state secrets, etc. Sensitive data can also be compositional. For example, the median income of several members of a political organization may be sensitive. A program which operates on sensitive data may propagate that data throughout process memory and CPU registers. This effect of spreading footprints of sensitive data throughout the process state is known as *label creep* [135]. Informally, a program which does not leak any of its sensitive data can be considered *secure*.

Depending on the security parameters of the application, it may be acceptable to treat program metadata as public. Program metadata can include, for instance, the number of elements in a data structure, or the size of each element in the structure. Thus an oblivious search algorithm must not leak the value of the search query or any of the values in the dataset being searched, though it may expose the size of the dataset, or the number of queries on the dataset. If scalar metadata describing the size of an object or dataset must be kept secret, then the developer must identify a suitable upper bound, and fix the size to that upper bound [56, 176].

Memory-trace obliviousness is generally considered an effective countermeasure against control flow-based side-channel attacks [104], but it does not guarantee the absence of leaks via storage channels. The classical security notion of *noninterference* does address these leaks. A program has the property of noninterference if its secret inputs do not “interfere” with its public outputs [76, 135]. That is, public program outputs must not vary with respect to secret program inputs. Mirroring the broad security goals set by Liu et al. [104], but in a more applied setting, this dissertation adopts an extended definition of noninterference to additionally cover memory-trace obliviousness. Hence a program which has the property of noninterference should not leak sensitive information through control flow-based side channels (including cache-based side channels) and storage channels.

A general solution to enforce noninterference should at least have the following three characteristics: it should (1) allow the program author to annotate or otherwise distinguish between secret and public data, (2) provide a mechanism to track the propagation of secret data through the program, and (3) implement constraints on secret data to prevent it from being leaked through either a side channel or a storage channel. The first requirement hints at a language-based approach: the subjective assignment of security classifications to program data is entirely motivated by the meaning of that data—which is visible to the developer at the programming language level. Once program data has been categorized according to sensitivity, requirements (2) and (3) could be satisfied by a variety of techniques. Both can be accomplished at the programming language level with security-type systems (this approach is most common, e.g., [102, 104, 106, 119, 131, 135, 143, 175]), with static analysis (e.g., [133, 144, 156]), or with enforcement at the computer architecture level (e.g., [27, 28, 105]).

The novel solution proposed in this dissertation uses template metaprogramming (to be defined in Chapter 2) to implement a security-type system on top of C++'s existing type system. The security-type system statically enforces an information-flow policy which constrains the propagation and usage of secret data. The implementation utilizes C++'s Turing-complete template type system to compute—at compile time—whether a given program conforms to the information-flow policy. In particular, the policy enforces noninterference. Hence secret data cannot influence program control flow or memory access patterns, nor can secret data be written to public variables or memory, or to a public storage channel. Secret data also cannot determine whether or not a program will terminate, or when it will terminate. This new type system is layered on top of C++'s type system to form a new programming language: *Covert C++*.

Thesis: The C++ template type system is sufficiently expressive to statically facilitate noninterference in a program; moreover, this property can be preserved by the compiler and linker.

The first conjunct of the thesis statement is established by Covert C++. The word "facilitate" is significant. The Covert C++ security-type system itself cannot guarantee noninterference because C++ is not a type-safe language. However, with some additional assumptions about the source program, it is possible to make an informal argument that Covert C++ enforces noninterference; an informal proof is presented in Chapter 5. A formal proof of noninterference for a strict subset of Covert C++ is presented in Chapter 4.

The second conjunct of the thesis is unfortunately not an immediate consequence of the former. As observed by D'Silva et. al. [68], compiler optimizations are only guaranteed to preserve semantics; they do not guarantee the preservation of security properties which are not entirely characterized by semantics.

To verify that the compiled binaries of functions which type check in Covert C++ actually satisfy the noninterference property, this dissertation introduces a new Noninterference Verification Tool (NVT). The NVT is a dynamic analysis tool which repeatedly fuzzes the secret inputs of a given function and monitors the procedure's memory trace for inconsistencies between fuzz iterations. This is a new solution to the classical problem of verifying noninterference for a given program and machine architecture [77].

The thesis statement above is not the only contribution made by this dissertation. In summary, the complete list of novel ideas presented in this dissertation for computer science and the computer industry are as follows:

- Show that the C++ template type system is sufficiently expressive to allow another type system to be superimposed on top of it.
- Demonstrate that an unmodified C++ compiler can be used to detect a variety of security violations, including side-channel leak vulnerabilities.
- Show how to exploit C++ overload resolution to "train" an unmodified C++ compiler to automatically optimize program code, depending on program security parameters.
- Show how to perform memory-trace oblivious computations with the C++ Standard Template Library (STL).
- Present an extension to the Covert C++ type system to facilitate secure multi-party computation (SMPC).
- Introduce a novel dynamic analysis technique to test program code for adherence to the noninterference property.
- Provide a refactoring toolchain to transform legacy C and C++ codebases into security-typed Covert C++ programs.

- Establish the utility of language-based security in industrial programming languages, especially security-type systems.

The remaining chapters of this dissertation describe each of these points in greater detail. Their collective contribution is to establish a framework for secure cloud computing, with minimal deviation from the programming conventions, languages, and tools already being used in industry.

The dissertation is organized as follows:

Chapter 2 introduces the preliminary information required to understand the remainder of the dissertation. Background topics include information-flow analysis, noninterference, security-type systems, side-channel attacks, template metaprogramming, and oblivious memory techniques.

Chapter 3 is a glossary of commonly-used definitions in the dissertation.

Chapter 4 introduces the *Core Covert* language. Core Covert is a simplified representation of Covert C++ which has been formally verified to enforce a strong form of termination-sensitive noninterference, including memory-trace obliviousness. The purpose of this chapter is to build an intuition about the motivations for the typing rules which characterize Covert C++.

Chapters 5 and 6 describe the solution to the problem presented in the thesis statement. Chapter 5 provides an overview of the Covert C++ type system, and then discusses the complex details of its implementation. The main theorem of this dissertation establishes that the Covert C++ type system enforces noninterference, subject to several reasonable assumptions. It concludes with a case study demonstrating the application of Covert C++ to protect a digital rights management (DRM) SGX enclave application. Chapter 6 describes the NVT in depth, and presents noninterference analysis results for a series of Covert C++ algorithms.

Chapters 7 and 8 present two extensions to Covert C++. Chapter 7 argues that the problem of secure multi-party computation can be reduced to a noninterference problem. The solution provided by Covert C++ is to generalize the security classes (secret and public) to identify specific principals participating in a multi-party computation. If the security classes of principals form a lattice, then the Covert C++ type system can guarantee a generalized form of noninterference: the sensitive data contributed by principal A does not interfere with the public output data observable by principal B, and vice-versa. This solution extends to an arbitrary number of principals.

Chapter 8 introduces Covert C++ oblivious iterators. Oblivious iterators facilitate high-performance algorithm design, without sacrificing noninterference. The oblivious iterators are provided by libOblivious, an independent C/C++ library which provides primitives, algorithms, containers, and iterators to facilitate memory-trace oblivious programming, without relying on additional hardware support or external tools.

Chapter 9 gives an overview of the wide body of work related to Covert C++. The breadth of the side-channel attack surface combined with the urgency of addressing data privacy has driven intense research in this area over the past decade. This chapter reviews the latest advances in side-channel attack defense, and compares Covert C++ against competing solutions.

Chapter 10 concludes the dissertation. It summarizes the reasons why Covert C++ (and its broader toolchain) substantiate the thesis statement. Furthermore, the techniques developed in this dissertation to implement Covert C++ can be extended to address other problem domains.

Chapter 2

Background and Preliminaries

This chapter introduces the concepts and definitions required to understand the subsequent solution chapters. Covert C++ is designed primarily to address a variety of side-channel attacks on the memory hierarchy. Side-channel attacks were first studied in depth in the late 1990s, in the area of applied cryptography. With the advent of cloud computing, research on side-channel attacks has surged in recent years. Yet the mechanisms employed by Covert C++ are rooted in the subject of information-flow analysis, which dates back to the 1970s.

The following sections survey the history of these and other topics which have influenced the development of Covert C++. This chapter also introduces formal definitions for terms which were mentioned in Chapter 1, and form the theoretical foundation on which the solution chapters are rooted.

2.1 Side-Channel Attacks

Interest in side-channel attacks first arose in the area of cryptanalysis, the study of attacks on cryptographic algorithms. Kocher [97] first observed that by measuring the amount of time required to execute a cryptographic key exchange algorithm, a malicious adversary could infer characteristics of the secret inputs. For example, fixed Diffie-Hellman exponents and factorizations of RSA keys could be recovered on certain implementations, effectively breaking these cryptosystems. Both Diffie-Hellman [66] and RSA [134] use exponentiation to generate cryptographic keys. The attack relies on an important property of exponentiation on most computing devices: the amount of time required to compute an exponential scales with the value of the exponent. Hence the adversary can guess the exponent bit-by-bit, observing when a change in the given bit causes the implementation to run a little bit longer.

Biham and Shamir [36] demonstrated a *differential fault analysis* attack on cryptographic devices such as smart cards. The authors found that an attacker with physical access to the device could in many cases manually inject hardware faults (e.g., by altering the device voltage). If the device is being used for encryption, then the attacker can compare a known-good ciphertext output against several (e.g., 50-200) fault-induced ciphertext outputs. Subsequent cryptanalysis could, for example, recover a DES key. A theoretical description of differential fault analysis attacks on RSA was outlined in [38].

Hall et al. [84] described several *reaction attacks* on public-key cryptosystems. These attacks require the adversary to observe the reaction of a victim (e.g., facial expression, body language, etc.). While observing the victim, the adversary iteratively adjusts some decryption parameters bit-by-bit, using the victim's reaction to determine when the decryption succeeds or fails. For example, this attack could recover the error correction vector used to encrypt a message in the McEliece public-key cryptosystem [111] by constructing a ciphertext containing one more error than the maximum which could be handled by the underlying error correction algorithm. The attacker could then twiddle the bits containing the errors—while observing the victim's reactions—to precisely determine the correct values of those bits. With an error-free ciphertext, the attacker can then apply a previously discovered cryptanalysis technique [20] to recover the plaintext message. Several related attacks were reported in the same year by Bellovin [33].

Kocher et al. [95] introduced *simple power analysis* (SPA) and *differential power analysis* (DPA), techniques which can decipher cryptographic keys from power current measurements, taken while a block cipher cryptoalgorithm is running. SPA can be used to infer information about a cryptographic device's operation, including encryption key material. The attack proceeds by measuring variations in power current consumed by the device while it is running. A trace of the current drawn by the device can be used to identify the various stages of block encryption. Furthermore, the variation of current drawn due to conditional branching, comparison operations, multipliers, and exponentiators can leak information about the data being operated on, similar to the timing attack by Kocher [97] described earlier. DPA is an extension to SPA which requires the adversary to examine multiple power current traces to determine whether a guess of a given key block is correct. By repeating this process for all of the key blocks, the entire encryption key can be recovered. The authors demonstrated the efficacy of the attack on DES.

All of these attacks fall under the broad category of side-channel attacks defined by Kelsey et al.: "A side-channel attack occurs when an attacker is able to use some additional information leaked from the implementation of a cryptographic function to cryptanalyze the function"[1] [94]. The key word in this definition is "implementation." A side-channel attack on an algorithm does not necessarily indicate a vulnerability in the algorithm itself. Rather, a side-channel attack is made possible by some implementation detail, and/or some mechanism of the underlying hardware which may leak information through a side channel.

The work in this dissertation addresses a specific category of side-channel attacks: attacks which observe flows of information by exploiting information leakage in the memory hierarchy. Hu [86] first considered the leakage of information through the CPU cache, but in the context of covert-channel attacks [165]. Although other works (e.g., [94, 128, 159]) postulated side-channel attacks exploiting the state of the CPU cache, Osvik et al. [127] and Bernstein [35] were concurrently the first to demonstrate a practical approach.

[1]It may not be immediately clear to the reader why reaction attacks are side-channel attacks. The physical implementation of a device such as an ATM assumes the physical presence of the user. Under this premise, the screen of the machine and the face of the user constitute a communicate channel over which a narrow band (e.g., a "yes"/"no" answer) of data may be transmitted. That is, both the machine output and the user's reactions reveal additional information from which a nearby observer could deduce more sensitive information about the transaction. This vulnerability in cyber/human interaction systems has been considered and addressed with creative solutions (e.g., [123, 151]).

The Prime+Probe technique described in [127] can extract information from some victim tenant on a multi-tenant machine, without requiring any administrator privileges. The strategy is to first "prime" a CPU cache by loading a contiguous byte array large enough to cover the entire cache, then wait for the victim tenant to execute for some fixed time interval, and finally "probe" the cache by measuring the time required to access memory within the blocks that had been primed. If a probe on a block that had been primed is slower after the victim has run, then that block must have been evicted by the victim. If the victim is operating on a data structure containing sensitive information, then this probe can reveal which elements of the structure were accessed.

The Evict+Time technique is also described in [127] (a similar variant is given in [35]). Osvik et al. apply this technique to an optimized implementation of the Advanced Encryption Standard (AES) [70] which uses a precomputed lookup table to improve performance. The attacker must have prior knowledge of the virtual memory addresses corresponding to the lookup tables. The attacker first times an execution of AES over a given plaintext p, and then uses her knowledge of virtual memory mappings to selectively evict the cache set containing the address of some index y in the lookup table. The attacker then triggers another encryption of p, and times the execution. The attacker will observe a slowdown if the cache set corresponding to y was accessed by the victim during the encryption process. If no slowdown is observed, then y definitely was not accessed.

One limitation of these two cache side-channel attacks is that they require the adversary and the victim processes to be executing on the same core. One reason is that the victim must remain paused while the adversary performs her measurements. Another reason is that these attacks are much easier to perform on the L1 cache because it is smaller and faster, and thus more likely to allow the adversary to complete the attack steps before being preempted by the process scheduler. Since the L1 cache is local to each core, the attack must be performed on a single core. These constraints are only partial; for instance, the adversary can simultaneously direct an attack on the CPU scheduler [121, 158].

An approach which can exploit cache state across CPU cores was proposed by Yarom and Falkner [174]. The Flush+Reload side-channel attack uses the x86 `clflush` instruction to selectively flush specific cache blocks being used by the victim process, which is possibly running on a separate core from the attacker process. The key property of `clflush` is that it flushes the given cache block from all levels of the cache hierarchy, and on all cores. The attack also assumes that the attacker and victim are sharing some pages, possibly due to content-based sharing (a.k.a. memory deduplication), an optimization feature of Linux, Windows, and several hypervisors [174]. Content-based sharing automatically shares pages between processes when those pages contain duplicate data, thus reducing total memory utilization in the system [24]. The attacker uses `clflush` to evict some cache blocks from the shared page(s), then waits for the victim to access those cache blocks, and finally probes (reloads) those cache blocks to determine which were accessed by the victim.

The Evict+Time, Prime+Probe, and Flush+Reload attacks constitute the "Big 3" cache side-channel attacks [72]. There are several other attacks which are related in part to the Big 3 (e.g., [81, 82, 83]).

Over the past several years, the proliferation of cloud computing and concerns over data privacy in the cloud have prompted renewed interest in side-channel attacks. Intel Software Guard Extensions (SGX) [112] is a set of x86-64 instructions which allow a user application (or OS) to create an isolated execution environment, called an SGX enclave. That is, the CPU provides an encrypted region of RAM for use by the application, such that no other process is allowed to see the plaintext contents, nor access the ciphertext contents. Other features of SGX include data sealing, key generation and management, and remote attestation [23].

The basic usage model for SGX in the cloud is depicted in Figure 2.1. Alice, the client, has an image of a program which she wants to run on an untrusted cloud server, controlled by Bob. Alice encrypts the secret portions of the program (including secret data) on her own trusted machine, then hashes the (transformed) program, and finally uploads the program to the untrusted server. Bob may create an enclave and load the contents of Alice's program

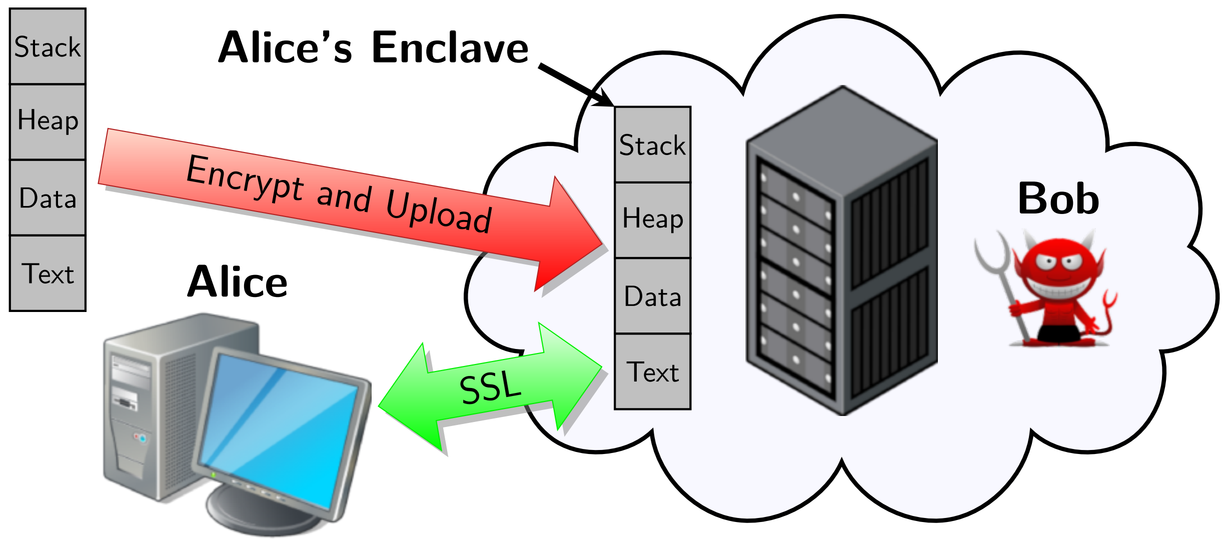

Figure 2.1: Setting up an SGX enclave in the cloud

page-by-page into the enclave. Each page that is added to an enclave is also extended into a hash digest for the enclave. After the enclave has been built and initialized, Alice can perform an attestation check on the enclave, which will allow her to verify that the contents of the enclave—recorded in its tamper-proof hash digest—match the hash she computed offline. Hence Alice would be able to detect any change made by Bob to the enclave program. Once Alice has successfully observed the integrity of her enclave, she can open an encrypted channel between her trusted machine and the enclave application, thus circumventing the host OS on Bob's cloud server. Alice securely uploads the key that was used to encrypt the enclave contents, which the enclave program then uses to decrypt its secret code and data. The enclave application can then proceed to run in its isolated execution environment, and Alice can communicate with it over an encyrpted channel (e.g., SSL). SGX's memory encryption will not allow Bob to see Alice's secrets in plaintext. Variations on this basic usage scheme have been used to implement secure MapReduce [138] and isolated cloud VMs [31].

However, SGX does not provide any intrinsic protections against side-channel attacks. The Intel SGX user's guide states that "Intel® Software Guard Extensions is not designed to handle side channel attacks or reverse engineering. It is up to the Intel® SGX developers to build enclaves that are protected against these types of attacks" [15].

Xu et al. [171] demonstrated on an Intel SGX platform that an adversary with control over the untrusted operating system running on that platform can observe the sequence of pages in memory touched by an SGX enclave program. By persistently evicting the enclave program's pages from memory, the adversary can force each memory access attempted by the enclave program to trigger a page fault. Thus the adversary is notified whenever the enclave program accesses any given page. If the sequence of pages touched by the enclave program depends on secret data (e.g., some secret data determines an index into a very large array) then the adversary may be able to infer the value of that secret data. The adversary can also deduce a secret when it is used to determine a branch that might cross a page boundary.

On x86 platforms the page size is 4 KB. The page resolution of this side-channel attack may be sufficient for attacks on applications which operate on large datasets, such as images [171]. Side-channel attacks on SGX have also been demonstrated at finer resolution. Specifically, Liu et al. [107] demonstrated that last-level cache attacks are possible on multi-tenant server platforms. Their implementation uses the Prime+Probe technique [127] described above.

2.2 Memory-Trace Oblivious Computation

Side-channel attacks on the memory hierarchy can be prevented by obfuscating memory accesses. Indeed, this was one of the defense strategies proposed by Osvik et al. in their seminal paper [127] on side-channel attacks against AES. There are several popular methods for obfuscating memory accesses, which vary in their performance overhead and protection.

This section first formalizes the notion of memory-trace obliviousness that was introduced in Chapter 1. Let $\mathcal{M}$ be a model of a machine, which is controlled by the adversary. The adversary can use $\mathcal{M}$ to run a program Π with secret and public inputs I_{Secret} and I_{Public}, respectively. Non-secret inputs are visible to the adversary; secret inputs are not visible (e.g., they may have been encrypted). On a concrete machine, I_{Secret} and I_{Public} may correspond to data in memory or in CPU registers, or data read from a file, `stdin`, etc. Execution of a program Π on $\mathcal{M}$ emits a memory trace τ, denoted $\mathcal{M}(\Pi, I_{Secret}, I_{Public}) \rightsquigarrow \tau$. The trace τ is a (possibly infinite) sequence of memory addresses of length $|\tau|$. It records the history of memory reads, writes, and instruction fetches made by Π.

Definition 2.1. The adversary's observational power is characterized by a relation $\sim_{Adv}$ such that for some positive integer constant n, $\tau \sim_{Adv} \tau'$ if:

1. $|\tau| = |\tau'|$, and
2. each pair of corresponding addresses in τ and τ' is equal, with the (possible) exception of the n least significant bits in each address.

The positive integer n is called the *granularity* of $\sim_{Adv}$[2]. △

Definition 2.2. An adversary with observational granularity equal to 0 is said to have *perfect* observational granularity. △

Example 2.1. An adversary using the forced page fault strategy [171] to observe τ has an observational granularity of 12 bits (the number of bits required to address 4 KB). △

Example 2.2. An adversary using the cache block Prime+Probe strategy [107, 127] to observe τ has an observational granularity of 6 bits. △

Definition 2.3. If the granularity of $\sim_{Adv}$ is n, then a contiguous region of memory with size and alignment equal to 2^n bytes is called the *mask* of $\sim_{Adv}$. △

For instance, the mask of $\sim_{Adv}$ in Example 2.1 is a 4 KB page, and in Example 2.2 it is a cache block. Intuitively, the mask is the smallest contiguous unit of memory within which the adversary cannot distinguish between accesses at different memory addresses.

Definition 2.4. A program Π is *memory trace oblivious* if, for all I_{Public}, I_{Secret}, and I'_{Secret},

$$\mathcal{M}(\Pi, I_{Secret}, I_{Public}) \rightsquigarrow \tau \text{ and } \mathcal{M}(\Pi, I'_{Secret}, I_{Public}) \rightsquigarrow \tau'$$

implies $\tau \sim_{Adv} \tau'$. △

[2]Thus $\tau \sim_{Adv} \tau'$ if $\tau = \tau'$, but $\tau \sim_{Adv} \tau'$ does not necessarily imply $\tau = \tau'$.

That is, for any fixed public input, any variation in the secret input must not affect the memory trace visible to the adversary. Hence the adversary cannot deduce anything about the values of the secret inputs by observing the memory traces.

The definition of memory-trace obliviousness presented in this dissertation is similar to, but more refined than the definition used in previous works. Liu et al. [104] defined trace equivalence to mean syntactic equivalence, i.e., two memory traces are equivalent if they recorded the same commands with the same arguments in the same order. The later GhostRider [105] architecture takes trace equivalence to mean that the order of events (e.g., fetches, oblivious reads, and oblivious writes) is identical. Their definitions also do not account for adversaries with differing observational power.

One substantial limitation of Definition 2.4 (and also of [104] and [105]) is that it is only useful when applied to deterministic (e.g., single-threaded) programs. For example, if Π is multi-threaded and has no secret inputs, it should intuitively be secure because there is no secret data to leak. However, any two executions of Π over the same public inputs could trivially yield two memory traces which are distinguishable to the adversary, and thus Π would not be considered secure. Hence Definition 2.4 is an over-approximation of security for this adversary model. On the other hand, a noteworthy advantage of Definition 2.4 is that it allows for verifiable security for compiled programs, as discussed later in Chapter 6.

Theorem 2.1 (Granularity Subsumption)**.** *Let $\sim_{Adv_1}$ and $\sim_{Adv_2}$ characterize two adversaries with observational granularities n and m, respectively. If $n \leq m$, then a program Π which is memory trace oblivious for $\sim_{Adv_1}$, is also memory trace oblivious for $\sim_{Adv_2}$.*

Proof. Suppose that Π is memory trace oblivious for $\sim_{Adv_1}$, and let I_{Secret} and I'_{Secret} be arbitrary secret input sequences accepted by Π, and likewise have I_{Public} as an arbitrary public input sequence accepted by Π. Hence if

$$\mathcal{M}(\Pi, I_{Secret}, I_{Public}) \rightsquigarrow \tau \text{ and } \mathcal{M}(\Pi, I'_{Secret}, I_{Public}) \rightsquigarrow \tau',$$

then $\tau \sim_{Adv_1} \tau'$. By Definition 2.1, $|\tau| = |\tau'|$ and all corresponding addresses in τ and τ' are equivalent, with the possible exception of their n least significant bits. Since $n \leq m$, it follows that all corresponding addresses in τ and τ' are also equivalent with the possible exception of their m least significant bits. Thus $\tau \sim_{Adv_2} \tau'$. Because I_{Secret}, I'_{Secret}, and I_{Public} were arbitrary, it follows from Definition 2.4 that Π is memory trace oblivious for $\sim_{Adv_2}$. $\square$

Example 2.3. By Theorem 2.1, a program which is memory-trace oblivious against an adversary using the cache block PRIME+PROBE strategy [107, 127] is also memory-trace oblivious against an adversary using the forced page eviction strategy [171], because the x86 page size is divisible by the x86 block size. $\triangle$

Memory-trace oblivious computations must not allow secret data to influence memory accesses in two ways: instruction fetches (via control flow) and data accesses (reads and writes). This implies that secret data must not be used as a loop termination condition or any other branch, such as an `if`/`else if`/`else` or `switch`. This rule also extends to branching operators (ternary `?:`) and short-circuiting operators (`&&` and `||`) which only conditionally evaluate some of their arguments.

These constraints may at first seem too restrictive for the developer. However, many algorithms can be expressed in terms of ternary operators. A non-branching ternary operator can be constructed using the `cmov` family of instructions on x86 [58]. This technique has been deployed in related projects (e.g., [124, 133, 136]).

With a non-branching ternary operator (call this operation `o_copy()`), it becomes possible to construct oblivious read and write operations. One such solution is given in Listing 2.1. The `read()` function iterates through the entire region of memory spanning the range [`I`, `E`), reading from every integer-aligned address. Only when the current address matches the argument `addr` is the value of `ret` updated to the value of `*I`, which at that point is equal to the value of `*addr`. This heuristic does not leak the value of `addr` because every value in [`I`, `E`) is read exactly once. Chapter 8 describes how common algorithms can be built using just a few oblivious primitives such as the non-branching ternary.

If the adversary model assumes a cache block mask for x86-64, the machine $\mathcal{M}$ does not

```
int32_t read(const int32_t *I, const int32_t *E,
             const int32_t *addr) {
  int32_t ret = 0;
  while (I++ != E) {
    ret = o_copy(I == addr, *I, ret);
  }
  return ret;
}

int32_t write(const int32_t *I, const int32_t *E,
              int32_t *addr, int32_t val) {
  while (I++ != E) {
    *I = o_copy(I == addr, *I, val);
  }
}
```

Listing 2.1: Naïve memory-trace oblivious read and write operations

emit a memory trace at perfect (0-bit) granularity. Because the adversary can only observe which cache block is touched on a given access, and not the specific address within the block, the x86-64 model of $\mathcal{M}$ emits a memory trace with 6-bit granularity. Thus the solution in Listing 2.1 is naïve (it overestimates the power of the adversary) because it is not necessary to read from every element in the array. It suffices to read only one value from each cache block within a range of memory, rather than read every single value [41]. Listing 2.1 could then be modified to read one integer from each cache block covered by $[\texttt{I}, \texttt{E})$, e.g., it could read every 16th integer before and after `addr`.

An even more efficient solution is to vectorize the read. New x86-64 CPUs provide AVX2 instructions which operate on 256-bit vector registers [58]. In particular, AVX2 introduced the `vpagther` instruction family. A single `vpgatherdd` instruction can read eight 32-bit integers from non-contiguous addresses in memory. As shown in Figure 2.2, `vpgatherdd` can be used to touch up to eight cache blocks per instruction. This vectorized oblivious read strategy has also been deployed in related work (e.g., [124, 133]).

The weaker form of memory obfuscation is data obliviousness, which requires only that data access patterns must not reveal sensitive data. It does not consider the number of

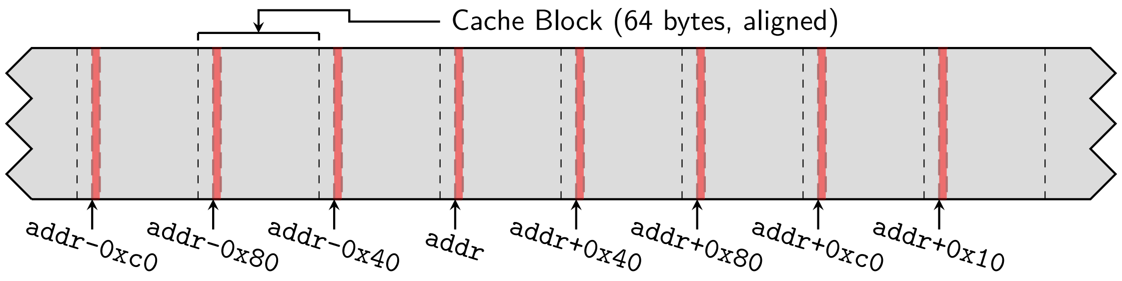

Figure 2.2: Oblivious read using `vpgatherdd`

accesses made, or the control flow of the program. However, note that this definition does not pose any constraints on the particular order of accesses, thus it allows for probabilistic approaches, etc. The traditional data-oblivious memory obfuscation technique is oblivious RAM (ORAM) [51, 78, 79], which uses probabilistic dynamic memory shuffling strategies to achieve polylogarithmic overhead. Since the original description of ORAM by Goldreich [78], several more ORAM algorithms with improved performance have been identified (e.g., [52, 80, 141, 150]). Rane et al. [133] observed that the vectorized memory scanning technique using `vpgather` instructions is actually faster than Path ORAM [150] by several orders of magnitude for an array containing up to 1 billion elements.

2.3 Information-Flow Analysis and Noninterference

Information-flow analysis is the study of the propagation of information through a program. This is distinct from access control, which restricts the release of information from a resource. Access control lists and capabilities, the building blocks of access control [50], are orthogonal methods of associating resources with access rights. Once read access to a resource has been granted to a principal (e.g., a user), neither method alone can restrict what that principal may do with the information obtained from that resource. For example, if Alice is allowed to download a certain file, and Bob is not allowed to download that same file, then Alice can simply download the file and give it to Bob. Information-flow analysis provides the means to address this problem by tracking and constraining the propagation of information.

Information-flow analysis can be decomposed into two distinct problem areas. *Information-flow policies* describe the manner by which data propagation affects data confidentiality. *Information-flow controls* are the mechanisms that enforce a given information-flow policy. In particular, the work in this dissertation is motivated by the security properties of a particular information-flow policy: noninterference. Many information-flow policies can be enforced by a programming language's type system. Hence the type system becomes the information-flow control. This is the approach used by Covert C++.

The following subsections survey the history of information-flow analysis (as it relates to this dissertation) since its conception by Denning and Denning in 1977 [64]. A much more comprehensive history of information-flow analysis has been chronicled by Sabelfeld and Myers [135]. A more friendly introduction to information-flow analysis was given by Smith [145].

2.3.1 Information-Flow Analysis

Denning and Denning [64] conceptualized the flow of information within a program in the following manner: information *flows* from an object x to another object y whenever (1) some information stored in x is transferred to y, or (2) information stored in x is used to derive or influence the value in y. The former is called an *explicit flow*, denoted $x \Rightarrow_E y$. The latter is an *implicit flow*, $x \Rightarrow_I y$. All flows are either explicit or implicit; a flow of either or both categories can be denoted $x \Rightarrow y$. All flows are transitive, thus $x \Rightarrow y$ and $y \Rightarrow z$ implies $x \Rightarrow z$.

Explicit flows result from assignment operators, passing arguments to a function, invoking a `memcpy()`, etc. Implicit flows result from program control flow and array accesses. For instance, in the C program:

```
int y;
if (x) { y = 42; }
else { y = 12; }
printf("y:␣\%d\n", y);
```

the value of x is not stored in y. However, the value of x is used to determine the value to be stored in y. Thus an observer who is only able to see the value of y may be able to infer the value of x, or at least infer some property of x, such as whether or not x == 0. Some operations combine both explicit and implicit flows:

```
int y = arr[n];
```

Here, the assignment operator copies the value in arr[n] to y, hence arr[n] $\Rightarrow_E$ y. The value of n also influences the assignment to y: n $\Rightarrow_I$ y. This distinction will become crucially important in Chapter 8.

Flows on their own are interesting, but not particularly useful. They become useful when combined with a information-flow policy, represented as $\langle S, \rightarrow \rangle$, where S is a lattice of security classes, and $\rightarrow$ is a relation on objects specifying when a flow is permitted, given the security classes of those objects. For example, S could be the set of military security classes $\mathcal{L}$: public (P), confidential (C), secret (S), and top secret (TS). They form a linear priority lattice $(\mathcal{L}, \leq)$ wherein for each pair $(c, c') \in \{(P, C), (C, S), (S, TS)\}$, $c \leq c'$ and $c' \not\leq c$. The information-flow policy $\langle \mathcal{L}, \rightarrow \rangle$ could define the flow relation such that $x \rightarrow y$ only if $class(x) \leq class(y)$. For instance, top secret information should not be allowed to flow into a public object. The policy $\langle \mathcal{L}, \rightarrow \rangle$ enforces the Simple Security and * (star) Properties ("no read-up" and "no write-down," respectively) of the Bell-LaPadula Model [32].

A program Π adheres to its information-flow policy if no possible execution of Π results in a flow $x \Rightarrow y$ where $x \rightarrow y$ is not allowed by the information-flow policy. Unfortunately, this general formulation of information-flow security is difficult to apply because it is undecidable, i.e., it reduces to the halting problem [61]. Suppose that the given information-flow policy does allow x $\rightarrow$ y, and consider:

```
while (true) {} // loops forever
y = x;
```

Clearly, this short program does adhere to the information-flow policy. However, a solution to the halting problem would be required to make this kind of decision, in general. Denning and Denning formulated a more restrictive definition of information-flow security, which is decidable:

$$\text{“}x \Rightarrow y \text{ is specified by } \Pi \text{ only if } x \to y\text{” [64].} \tag{2.1}$$

Note that this formulation offers less precision in program certification; it would reject the program with the infinite loop. The crucial improvement is that it applies to a program's specification, rather than its execution. This hints at a solution which can be enforced statically, for example at compile time. Indeed, a compiler modification to enforce "certification semantics" guaranteeing (2.1) was proposed by Denning and Denning [64].

The final consideration is how to apply an information-flow policy when the value of one object is influenced by flows from several other objects. Recall the scenario when reading from an array: `int y = arr[n]`. Suppose that this program uses the lattice of military security labels $(\mathcal{L}, \leq)$, and that $\mathit{class}(\mathtt{y}) = TS$, $\mathit{class}(\mathtt{arr[n]}) = P$, and $\mathit{class}(\mathtt{n}) = S$. Instead of certifying $\mathtt{arr[n]} \Rightarrow \mathtt{y}$ and $\mathtt{n} \Rightarrow \mathtt{y}$ separately, the certification check can compute the least upper bound (denoted $\sqcup$) of the labels $\mathit{class}(\mathtt{arr[n]})$ and $\mathit{class}(\mathtt{n})$: $P \sqcup S = S$. The check can then determine whether the flow from some temporary rvalue object t with $\mathit{class}(t) = S$ is allowed to flow $t \Rightarrow \mathtt{y}$. This is allowed by $\langle \mathcal{L}, \to \rangle$ because $\mathit{class}(\mathtt{y}) = TS$, and $S \leq TS$.

2.3.2 Noninterference

The notion of noninterference was first introduced by Goguen and Meseguer in 1982 [76], and later refined in [77]. From their original paper:

> "one group of users, using a certain set of commands, is *noninterfering* with another group of users if what the first group does with those commands has no effect on what the second group of users can see" [76].

A common special case of this description is when the "second group of users" consists solely of the adversary, and the "first group" is the victim. Hence the goal is to shield the victim's computation and data from the adversary's view.

Goguen and Meseguer described an *unwinding* technique [77] for verifying noninterference: for all system states and output commands, demonstrate that the output visible to the adversary does not depend on any state transition command invoked by the victim. For a formal description of a programming language or a set of system commands, this definition is easy to apply. An induction proof over the semantic description of the language will suffice. McLean [115] suggested an alternative verification approach using trace semantics, i.e., sequences of procedure calls.

Unwinding is less useful when attempting to verify noninterference for a language without a formal description, or for a given program written in a language not intended to enforce noninterference. In this case, it is better to retrospectively consider Cohen's work on strong dependency [54, 55], an early solution to the confinement problem [99].

Definition 2.5. Given a program Π, an object b *strongly depends* on the value of an object a, denoted $a \mathrel{[\!\!\rhd^{\Pi}} b$, when there exist initial program states σ_1 and σ_2 such that

$$\sigma_1 =_a \sigma_2 \text{ and } \Pi(\sigma_1).b \neq \Pi(\sigma_2).b,$$

where $\sigma_1 =_a \sigma_2$ represents two states that are equal for all of their objects except for a. $\triangle$

Intuitively, $a \mathrel{[\!\!\rhd^{\Pi}} b$ when some execution of Π causes a to affect b. If I is the set of all inputs to Π provided by a group of users G_1, and O is the set of all outputs from Π visible to another group of users G_2, then noninterference can be equivalently characterized as the absence of strong dependency by any element of O on any element of I, i.e.,

$$\forall i \in I,\, o \in O.\;\; i \mathrel{[\!\!\rhd^{\Pi}} o \longrightarrow \bot.$$

Volpano and Smith [160] later argued that noninterference should also consider termination channels, which can leak a surprisingly large amount of information [25]. A *termination channel* leaks information when a program either converges or diverges, or terminates early due to an exception. Informally, *termination-sensitive noninterference* guarantees that secret inputs do not determine whether or when a program terminates. The Goguen-Meseguer model [76, 77] is a form of *termination-insensitive noninterference.*

The definition of noninterference used in this dissertation is an extension to memory-trace obliviousness (Definition 2.4) which additionally guarantees the absence of public outputs that strongly depend on secret inputs. In addition to emitting a memory trace τ, a program Π executing on $\mathcal{M}$ may also emit an output trace ω consisting of a sequence of outputs to any number of storage channels, denoted $\mathcal{M}(\Pi, I_{Secret}, I_{Public}) \rightarrow \omega$. The output sequence ω is a (possibly infinite) sequence of values of length $|\omega|$. A *storage channel* is an overt communication channel "maintained by the supervisor which can be written by the service and read by an unconfined program, either shortly after it is written or at some later time" [99]. An output to a storage channel on UNIX-like systems typically entails writing to a file descriptor. Henceforth, the term *trace* simply refers to the combined memory trace and output trace emitted by a program Π executing on $\mathcal{M}$ for a given set of inputs.

Let $=_{Adv}$ be a relation characterizing the adversary's view of the program outputs. For example, a program may emit both secret and public information, but the secret information is always encrypted. So if the adversary is assumed not to be able to read encrypted data, $=_{Adv}$ will only cover the public outputs. It is always assumed that the adversary can observe the number of outputs in ω, and the size of each output.

Definition 2.6. The *combined observational power* of the adversary $\simeq_{Adv}$ is given by

$$\mathcal{M}(\Pi, I_{Secret}, I_{Public}) \simeq_{Adv} \mathcal{M}(\Pi, I'_{Secret}, I_{Public})$$

if and only if

1. $\mathcal{M}(\Pi, I_{Secret}, I_{Public}) \rightsquigarrow \tau$ and $\mathcal{M}(\Pi, I'_{Secret}, I_{Public}) \rightsquigarrow \tau'$ implies $\tau \sim_{Adv} \tau'$, and
2. $\mathcal{M}(\Pi, I_{Secret}, I_{Public}) \rightarrow \omega$ and $\mathcal{M}(\Pi, I'_{Secret}, I_{Public}) \rightarrow \omega'$ implies $\omega =_{Adv} \omega'$

for all programs Π and input sequences I_{Public}, I_{Secret}, and I'_{Secret}, △

Definition 2.7. A program Π has the property of *noninterference* if

$$\mathcal{M}(\Pi, I_{Secret}, I_{Public}) \simeq_{Adv} \mathcal{M}(\Pi, I'_{Secret}, I_{Public})$$

for all I_{Public}, I_{Secret}, and I'_{Secret}[3]. △

[3]**A pedantic note on terminology:** To the best of the author's knowledge, this is the first time that

Theorem 2.2 (Noninterference implies Memory-Trace Obliviousness)**.** *If a program* Π *has the property of noninterference, then* Π *is also memory trace oblivious.*

Proof. Follows directly from unfolding Definitions 2.4, 2.6, and 2.7. □

Informally, noninterference guarantees that from the perspective of a malicious adversary, (1) public outputs do not depend on secret inputs, (2) the program's memory trace is not affected by secret inputs, and (3) secret inputs cannot determine whether or not a program will terminate. Property (1) corresponds to the absence of strong dependency [135] and property (2) corresponds to memory-trace obliviousness. Property (2) actually implies property (3), because when a program diverges, its trace has length equal to $|\mathbb{N}|$. When a program converges, its trace has length strictly less than $|\mathbb{N}|$. Similarly, a program which exits early due to an exception will have a shorter trace than a program which runs to completion, assuming both programs had made all of the same branch decisions before the exception had occurred in the first process. Hence Definition 2.7 is a form of termination-sensitive noninterference.

Note that because Definition 2.7 encompasses memory-trace obliviousness, this definition of noninterference also assumes that Π is deterministic. Yet noninterference does not always need to be held to this constraint. In fact there is a large body of theoretical work on nondeterministic noninterference [135].

For instance, Wittbold and Johnson [170] describe *possibilistic* and *probabilistic* noninterference. Possibilistic noninterference requires that secret inputs must not influence the set of

noninterference has been defined in precisely this manner. The closest comparison is the definition of memory-trace obliviousness given by Liu et al. [104], who conversely define memory-trace obliviousness as an extension to noninterference. Specifically, they define memory-trace obliviousness to mean that (1) low-equivalent initial memories imply low-equivalent output memories, and (2) memory traces are indistinguishable to the attacker. The term "memory trace oblivious" is sensible because both properties apply only to observable characteristics of memories. The key difference between Liu et al.'s work and the work presented in this dissertation is that they use a semantic model of execution akin to Volpano et al. [163], and Covert C++ uses a machine model of execution which consumes inputs and produces outputs, similar to the original Goguen-Meseguer model [76]. The machine model was selected for this dissertation because it easily incorporates storage channels. If memory accesses and instruction fetches are treated as outputs (which they are, from the adversary's perspective), then Definition 2.7 fits naturally into the Goguen-Meseguer model. Storage channels are not easily represented by the memory trace or state of program memory, hence the more general term "noninterference" was preferred.

```
{ // thread 1
  leak = secret;
}
```

```
{ // thread 2
  leak = random(100);
}
```

Figure 2.3: A multi-threaded program which has possibilistic noninterference, but not probabilistic noninterference

possible public outputs. However, for most practical scenarios this condition is insufficient to guarantee security. Consider the example in Figure 2.3 from McLean [113], consisting of two threads running concurrently.

Assume that `secret` is some number between `1` and `100`, and that `random(100)` also returns some number between `1` and `100`, with uniform probability. If the program outputs the value of `leak` upon termination, then the program will satisfy the definition of possibilistic noninterference because the value of `secret` does not affect the set of possible values for `leak`. However, suppose that the thread scheduler selects between threads 1 and 2 with equal probability. Then the program output will be the value of `secret` with probability $\frac{101}{200}$, and each other possible value with probability $\frac{1}{200}$. Thus by running the program multiple times, the adversary could certainly deduce the value of `secret` by observing the frequency distribution of output values. Probabilistic noninterference addresses this issue by requiring that secret inputs not affect the joint probability distribution of public outputs [170].

The discussion on nondeterministic noninterference so far applies only to classical noninterference, i.e., the relationship between secret inputs and public outputs. When additionally considering side channels, a useful definition of nondeterministic noninterference becomes more elusive—especially when the memory trace is an observable side channel.

Generalized noninterference, originally proposed by McCullough [110] and later refined by McLean [114], does provide a solution when the sequence of memory accesses and instruction fetches is treated as a part of the trace of low outputs. However, it is not immediately clear how to apply generalized noninterference to certify security on an actual platform, where the space of program traces is prohibitively large. For instance, given a program which uses t

threads and n as an upper bound on the number of instructions which can be executed by any of the t threads before the program terminates, there are $O(t^n)$ possible permutations of interleaved thread instructions that would constitute the program trace.

The purpose of the thesis is not only to construct a practical programming language to enforce noninterference, but also to demonstrate that the noninterference property can be preserved by an optimizing compiler. To this end, the strength of the noninterference requirement must be sufficient to allow compiled binaries to be *verifiably secure*. This is the topic of Chapter 6. Contemporary work on memory-trace obliviousness also does not consider nondeterminism [104, 105].

2.3.3 Language-Based Information Flow

As mentioned in the prior section, Denning and Denning [64] observed that the problem of certifying information-flow security for a program becomes decidable when the information-flow policy is embedded in the specification of the program. Their certification method—the information-flow control—was a compiler check on the certification semantics of the program [64].

Volpano et al. [163] simplified this concept by embedding an information-flow policy in a type system, thus creating a *security-type system*. In a security-type system, types additionally assign security intent to program data. A *security-typed language* is a programming language with a security-type system. The type checking tool for a security-typed language (possibly a part of the compiler) constitutes the information-flow control.

The soundness of the security-type system in [163] establishes noninterference over a lattice of security classes. Though the original formulation was somewhat involved, a simple, yet equivalent security-type system was later described by Sabelfeld and Myers [135], and is shown in Figure 2.4. This model is not designed to address side-channel attacks. It simply enforces the classical notion of noninterference: no low variable in the output state strongly depends on any high variable(s) in the input state. For simplicity, this language only

$$
\begin{array}{llll}
\text{[E1–2]} & \vdash exp : high & \dfrac{h \notin Vars(exp)}{\vdash exp : low} & \\[2ex]
\text{[C1–3]} & [pc] \vdash \mathsf{skip} & [pc] \vdash h{:=}exp & \dfrac{\vdash exp : low}{[low] \vdash l{:=}exp} \\[2ex]
\text{[C4–5]} & \dfrac{[pc] \vdash C_1 \quad [pc] \vdash C_2}{[pc] \vdash C_1; C_2} & \dfrac{\vdash exp : pc \quad [pc] \vdash C}{[pc] \vdash \mathsf{while}\ exp\ \mathsf{do}\ C} & \\[2ex]
\text{[C6–7]} & \dfrac{\vdash exp : pc \quad [pc] \vdash C_1 \quad [pc] \vdash C_2}{[pc] \vdash \mathsf{if}\ exp\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2} & \dfrac{[high] \vdash C}{[low] \vdash C} &
\end{array}
$$

Figure 2.4: A simple security-type system [135]

considers two variables: a *low* variable l, and a *high* variable h (*low* and *high* are types). The judgment $[pc] \vdash C$ means that the command C is valid in the program context *pc*. Similarly, the judgment $[pc] \vdash exp$ means that the expression *exp* is typable in *pc*. The purpose of *pc* is to track implicit flows. It can be thought of as assigning a security type class to the CPU's program counter. Note that this type system only guarantees termination-insensitive noninterference because a *high* while loop could potentially loop forever.

Rules E1-2 state that unless a given expression does not contain h, it must have type *high*. Rules C2-3 assert that l can only assume the value of a *low* expression, and in a *low* program context. Specifically, rule C3 prevents any explicit or implicit flow that would violate the security requirement. Rule C4 simply states that the program context is preserved across compound commands. Rules C5-6 govern implicit flows to the program context. Whenever a branch decision is to be made, the program context assumes the type of the branch condition. Hence any assignment made within that branch will be *high* if the branch condition was *high*. Rule C7 is a subsumption rule, allowing the program context to be reset to *low* after exiting a *high* conditional or loop.

Volpano and Smith later examined the effect of nondeterminism on the security-type system described above. They found that additional typing restrictions on control-flow commands (e.g., loops) were required to preserve noninterference for multi-threaded programs [146, 161].

Another of their works [162] postulated a language extension with first-order functions, polymorphically parameterized by security classes. One important contribution of this work was to formalize the notion of type inference for a security-typed language. In a programming language, *type inference* refers to automatic deduction (usually by the compiler) of types. For a security-typed language, type inference allows the compiler to automatically deduce the security class of an unlabeled expression or variable, given the context in which the expression or variable is being evaluated/declared.

In 1999, Myers [119] described the first practical security-typed language, based on Java. JFlow is a security-typed language which assigns to each variable both a primitive type (e.g., `int`) and a security label. The security label can either indicate the sensitivity of the variable, or the principal to which the data belongs. This latter category of labels is a realization of Myers and Liskov's earlier work [118, 120] on the *decentralized label model*, which allows principals to attach information-flow policies to certain pieces of data, and also to delegate capabilities on data to other principals. JFlow uses a mix of static and dynamic enforcement to track both explicit and implicit information flows. The security goal is to prevent information leaks through storage channels. Several contemporary examples of security-typed languages are summarized later in Chapter 9.

Henceforth, the term *classical noninterference* refers to the definition of noninterference given in Volpano et al. [163], i.e., secret inputs do not influence public outputs. The term *noninterference* strictly refers to Definition 2.7.

2.4 Secure Multi-Party Computation

The *secure multi-party computation* (SMPC) problem can be stated in generality as follows. Principals $P_0, P_1, \ldots, P_{n-1}$, each have input data $x_0, x_1, \ldots, x_{n-1}$, respectively. They wish to mutually compute the output value of a function $f(x_0, x_1, \ldots, x_{n-1})$, subject to the following constraints:

- **Confidentiality:** For all i, j such that $i \neq j$, P_i must not be able to infer x_j from any observable aspect of the computation, except for the final output of $f(x_0, x_1, \ldots, x_{n-1})$.
- **Integrity:** The final output of $f(x_0, x_1, \ldots, x_{n-1})$ must be correct, possibly within an acceptable margin of error [91]. Or the final output may be incorrect, but each P_i must be able to determine whether or not the final output is correct.

The computation of f may proceed on any one of the principal's machines, or any combination of them, or on an external third party's machine. The first accepted solution to this problem was the garbled circuit construction proposed by Yao [173]. Later solutions employed homomorphic encryption [34].

Cloud computing and IaaS have sparked a flurry of new research on SMPC platforms. SMPC has real and prospective applications in genomics (e.g., [56]), Internet of things and crypto banking (e.g., [177]), and even in sugar beet auctions [37]. One of the principle drawbacks to wider adoption has been the relatively limited scalability of garbled circuits and homomorphic encryptions over large data sets [56]. With the advent of IEEs such as Intel SGX, end users now have access to cloud-based platforms with built-in cryptographically protected execution environments. SGX not only has proved apt at shielding applications and data belonging to a single principal [31, 138], it has recently been examined as a candidate platform for SMPC [29, 124].

2.5 The C++ Template Type System

C++ first introduced templates in 1990, when the ANSI C++ committee was founded [6]. The purpose of templates was to introduce parameterized types that would be both expressive and efficient [153]. The motivating factor at the time was a need for a standard library that could provide polymorphic containers, such as vectors and maps.

C++ templates are instantiated by substituting types and/or constant values for the template parameters. For instance, the C++ Standard Template Library (STL) provides a

```
template <unsigned N> struct factorial {
  static const unsigned value =
      N * factorial<N - 1>::value;
};
template <> struct factorial<0> {
  static const unsigned value = 1;
};
```

Listing 2.2: A factorial metafunction

linked list template `std::list` which is parameterized on the type of the container element. So `std::list<int>` is a type for a list of integers, and likewise `std::list<std::string>` is a type for a list of strings, etc.

The C++ template system exceeded its original goals, empirically achieving "better than hand-coding performance," primarily because it facilitates inlining optimization [155]. Moreover, it is Turing-complete [98]. Listing 2.2 illustrates this phenomenon.

The `factorial` structure is defined twice, once for an arbitrary template argument `N`, and once for the specialized case where the template argument is `0`. When a new `factorial<T>` is instantiated, the compiler will prefer the most specialized definition, according to the partial ordering rules for template specializations [90]. The second definition of `factorial` will only be selected when the template argument is `0`. When a template class is instantiated, its static immutable members are evaluated—using constant folding if necessary. If other template expressions are involved in the constant expression being evaluated, these must be evaluated as well. In the first `factorial` definition, this behavior is exploited to recursively evaluate the factorial function on sequentially decreasing values of `N`. When the compiler must evaluate `factorial<0>`, the definition of the static member is the constant `1`, and the recursion halts. For example, given the statement `factorial<5>::value`, the C++ compiler will recursively fold the `value` member of `factorial<5>` into the constant `120`.

A *metafunction* is a compile-time function which accepts types and/or constants as parameters, and returns a type or constant as a result. The technique of compile-time computation using metafunctions is known as *template metaprogramming* [98].

Chapter 3

Definitions

This chapter serves as a glossary for terms that are frequently used throughout the dissertation. Some terms will be formally introduced in later chapters. An exhaustive list of terms used in this dissertation can be found in the Index.

application programming interface (API) Clearly-defined methods to facilitate communications between components in a system, for example between a program and a software library.

basic block A contiguous unit of execution, i.e., a sequence of instructions with a single entry point and a single exit point.

canonical type An unlabeled type, or a type of the form `SE<T, S1,...,SN>` where `T` is a labeled type with type depth equal to `N`.

channel Any mechanism which signals information, either implictly or explicitly.

classical noninterference A program has the property of classical noninterference when its public outputs do not strongly depend on its sensitive (secret) inputs.

cover An object in memory covers one or more masks when any portion of the object's contiguous memory lies within those masks.

covert type A type that is either malformed or canonical.

decentralized label model Assigns labels to data according to its ownership, and allows for the specification of policies that dictate how data can be created and declassified, and how capabilities on data can be delegated, etc.

enclave A protected region of memory that is dynamically encrypted, and can only be accessed by a process running in enclave mode.

flow A transfer of information from an object x to an object y. A flow is explicit if information is copied from x to y. A flow is implicit if information stored in x is used to derive or determine a value to be stored in y.

generalized label model Allows any object in a program to be assigned a security label. These labels must form a bounded join-semilattice.

granularity The number of least significant bits that the adversary is not able to observe in a memory trace. If this number is 0, the granularity is perfect. See Definitions 2.1 and 2.2.

information-flow control The mechanisms by which an information-flow policy is enforced.

information-flow policy Describes the manner in which data is allowed to propagate through a program.

inner type The data type in a covert type. For example, the inner type `SE<int, H>` is `int`.

input fuzzing Repeatedly invoking a program or function with one or more of its inputs adjusted before each invocation.

labeled type A type with non-zero type depth.

lattice An algebraic structure in which each pair of elements has a least upper bound and a greatest lower bound. A semilattice has either a least upper bound or a greatest lower bound for every pair. A lattice is bounded above if there exists an element that is greater than or equal to every element in the lattice, and similarly for bounded below.

least upper bound For two elements x and y in a lattice, the least upper bound of x and y (denoted $x \sqcup y$) is the least element z such that $x \leq z$ and $y \leq z$.

malformed type A type that is either `SE` or points to some `SE` type, but is not canonical.

mask The smallest contiguous unit of memory within which the adversary cannot distinguish between accesses at different memory addresses. See Definition 2.3.

memory trace The sequence of memory accesses τ made by a program during execution, including instruction fetches and data accesses (reads and writes).

memory trace oblivious A program has the property of memory-trace obliviousness if its memory trace does not strongly depend on any of its sensitive (secret) inputs. See Definition 2.4.

metadata Data used to describe or summarize other data, e.g., the size of an array.

metafunction A compile-time function which accepts types and/or constants as parameters, and returns a type or constant as a result.

noninterference A program has the property of noninterference if it has classical noninterference and it is memory-trace oblivious. See Definition 2.7. Moreover, noninterference is termination sensitive if secret inputs cannot affect whether or not a program will terminate (diverge); otherwise it is termination insensitive.

oblivious Short for memory-trace oblivious.

oblivious iterator An iterator whose memory access semantics (e.g., `*`, `[]` operators) are such that the memory trace does not reveal the address being accessed.

oblivious type A type with pointer-like semantics (e.g., a pointer or iterator) and security typing. Oblivious types are covert types..

output trace The sequence of outputs ω emitted by a program during execution, e.g., to `stdout` or to a file on disk.

primitive type A type belonging to any of the following categories: arithmetic (`int`, `char`, etc.), `enum`, non-function pointer, or pointer to any non-`SE` type, at any level.

pure Covert C++ Covert C++ without the libOblivious extensions described in Chapter 8.

secure multi-party computation (SMPC) A computation using input data from multiple principals, the goal of which is to produce an accurate result, without each principal learning any sensitive aspect of any other principal's data. See Section 2.4.

security label A representation of a security class in a Covert C++ program.

security-type system A type system in which types additionally assign security intent to program data. Moreover, the type system defines an information-flow policy to constrain the flow of program data, depending on its security type.

security-typed language A programming language with a security-type system, and wherein the type checking mechanism serves as the information-flow control.

security-upgrade aliasing A kind of pointer aliasing which allows, for instance, a covert pointer to a high object to alias a covert pointer to a low object (of the same type).

SGX An architectural extension to x86-64 that allows for the creation of protected regions of memory called enclaves.

side channel An inadvertent revelation of information by some mechanism of an algorithm's implementation, not attributable to some flaw in the algorithm's specification.

side-channel attack Occurs when a malicious adversary is able to infer sensitive data through a side channel. Can either be active (the adversary must do some additional work, such as evicting pages from an enclave) or passive (the adversary can simply observe the side channel).

storage channel A channel that can be written to by a service and read from by a supervisor, e.g., files on disk, network packets, `stdout`, etc.

strong dependency An object or output b strongly depends on an object or input a if the value of a can affect the value of b, for some initial program state. See Definition 2.5.

template metaprogramming The use of C++ templates to perform a computation at compile time. See Section 2.5.

termination channel A termination channel leaks information when a program either converges or diverges, or terminates early due to an exception.

test application A test module, together with any other libraries or modules it uses.

test module A shared object or DLL that exports NVT tests.

trace Combination of memory trace and output trace (τ, ω).

Turing complete Able to simulate a Turing machine. By the Turing-Church thesis [57], a Turing-complete system can perform any computation that can be performed by a human being following an algorithm.

type depth The number of security labels assigned to a given type.

type inference Automatic deduction of types, usually by the compiler.

unlabeled type A non-`SE` type with non-zero type depth.

Chapter 4

Core Covert

This chapter serves as an introduction to the motivation for the Covert C++ language, which is described in the next chapter. *Core Covert* is an imperative programming language with security typing over a lattice of security classes—specifically, a bounded join-semilattice[1]. It enforces an information-flow policy with classical noninterference similar to that of the original work by Volpano et al. [163] which was described in Chapter 2. Moreover, Core Covert places rigid constraints on the typing for statements and expressions which access memory and influence program control flow. These constraints enforce memory-trace obliviousness. Hence the information-flow policy establishes the property of noninterference given in Definition 2.7. Core Covert forms the blueprint on which Covert C++ is based.

Section 4.1 gives a formal description of the Core Covert language, along with an informal discussion about the purpose of the typing constraints, such as how they satisfy noninterference. In particular, Theorem 4.1 establishes a fundamental confidentiality property. Section 4.2 formulates and proves a series of properties about Core Covert. The main result is Theorem 4.7, which establishes that a Core Covert program which is typable enforces termination-sensitive noninterference. Section 4.3 revisits the `memcmp()` leak example from Chapter 1, and demonstrates how Core Covert's security-type system can detect the leak.

[1]A *bounded join-semilattice* is a pair $(\mathcal{L}, \leq)$ such that for every $x, y \in \mathcal{L}$, x and y have a join (a.k.a. least upper bound) in $\mathcal{L}$, and there exist some $\bot, \top \in \mathcal{L}$ such that $\bot \leq z$ and $z \leq \top$ for all $z \in \mathcal{L}$.

4.1 The Core Covert Language

The grammar of Core Covert is listed in Figure 4.1. A Core Covert program Π has an abstract syntax tree (AST) which consists of nested statements, expressions, and locations. The program context comprises a store Σ and a sequence of (immutable) inputs λ. Program execution, denoted $\Downarrow_\Pi$, emits a trace of memory accesses and a sequence of outputs, τ and ω, respectively. That is, $\Downarrow_\Pi$: $\Sigma \Rightarrow \lambda \Rightarrow \Pi \Rightarrow \tau \times \omega$.

The trace of memory accesses τ is represented as a sequence of addresses touched by the program. Program control flow decisions, memory reads, and memory writes are all recorded in τ. The one-to-one polymorphic "address-of" function α determines the address of a given expression, statement, or location. For example, the trace $\alpha(s) \triangleright \alpha(a+4) \triangleright \alpha(e_1)$ describes a program execution which branched to s, then touched array a at offset 4, and finally branched to expression e_1. The $\triangleright$ operator indicates the concatenation of two traces, and is left associative. Two memory traces are equivalent, written $\tau \equiv \tau'$, when each trace records the same addresses, and in the same order. The judgments to determine memory trace equivalence are given in Figure 4.2.

The trace of outputs ω is simply the sequence of integers emitted by the program during execution, i.e., whenever an output() expression is evaluated. The judgments to determine output trace equivalence, denoted $\omega \doteq \omega'$, are also shown in Figure 4.2.

The input context $\lambda[i \mapsto n]$ is a finite function mapping argument indices i to values n. The program store $\Sigma[a + n \mapsto n]$, a representation of DRAM, consists of a finite number of arrays of unbounded size. This description of the store implies that Core Covert is categorically memory safe; it is not possible to make an out-of-bounds access on an array.

Arrays can be accessed through locations ℓ, each of which evaluates to a specific position within a given array. Locations are evaluated via the function $\Downarrow_\ell$: $\Sigma \Rightarrow \lambda \Rightarrow \ell \Rightarrow (a+n) \times \tau \times \omega$. Intuitively, a location is like a pointer in C or C++. Formally, an (evaluated) location is a pair consisting of an array's base $a$and an offset n, denoted $a + n$. The pair $a + n$ is called a *point*. Two points $a_1 + n_1$ and $a_2 + n_2$ are equivalent when their bases are judgmentally equal

$$
\begin{array}{llcl}
\text{Locations} & \ell & ::= & a + e \\
\\
\text{Expressions} & e & ::= & n \\
 & & \mid & i \\
 & & \mid & \mathsf{read}(\ell) \\
 & & \mid & e_1 \odot e_2 \\
 & & \mid & e_1 \ ?\ e_2 \circ e_3 \\
 & & \mid & \mathsf{output}(e) \\
\\
\text{Statements} & s & ::= & \ell := e \\
 & & \mid & s_1 ; s_2 \\
 & & \mid & \mathsf{skip} \\
 & & \mid & \mathsf{select}\ e\ \{s_1, \ldots, s_m\} \\
 & & \mid & \mathsf{iterate}\ e\ s \\
\\
\text{Programs} & \Pi & ::= & \mathsf{main}\ \{\ s\ \} \\
\\
\text{Store} & \Sigma & ::= & \Sigma[a + n \mapsto n] \\
 & & \mid & \cdot \\
\\
\text{Store typing} & \Psi & ::= & \Psi[\ell : \gamma] \\
 & & \mid & \cdot \\
\\
\text{Input} & \lambda & ::= & \lambda[i \mapsto n] \\
 & & \mid & \cdot \\
\\
\text{Input typing} & \phi & ::= & \phi[i : \gamma] \\
 & & \mid & \cdot \\
\\
\text{Output trace} & \omega & ::= & n \\
 & & \mid & \omega_1 \triangleright \omega_2 \\
\\
\text{Memory trace} & \tau & ::= & \alpha(a + n) \\
 & & \mid & \alpha(e) \\
 & & \mid & \alpha(s) \\
 & & \mid & \tau_1 \triangleright \tau_2
\end{array}
$$

Figure 4.1: Core Covert grammar

$$\textsc{Out-Refl}\ \frac{}{\omega \doteq \omega}$$

$$\textsc{Out-Value}\ \frac{}{n \doteq n}$$

$$\textsc{Out-Concat}\ \frac{\omega_1 \doteq \omega_2 \qquad \omega_1' \doteq \omega_2'}{\omega_1 \triangleright \omega_1' \doteq \omega_2 \triangleright \omega_2'}$$

$$\textsc{Mem-Refl}\ \frac{}{\tau \equiv \tau}$$

$$\textsc{Mem-Addr}\ \frac{a_1 + n_1 = a_2 + n_2}{\alpha(a_1 + n_1) \equiv \alpha(a_2 + n_2)}$$

$$\textsc{Mem-Expr}\ \frac{}{\alpha(e) \equiv \alpha(e)}$$

$$\textsc{Mem-Stmt}\ \frac{}{\alpha(s) \equiv \alpha(s)}$$

$$\textsc{Mem-Concat}\ \frac{\tau_1 \equiv \tau_2 \qquad \tau_1' \equiv \tau_2'}{\tau_1 \triangleright \tau_1' \equiv \tau_2 \triangleright \tau_2'}$$

Figure 4.2: Core Covert trace judgments

(they refer to the same array), and their offsets are equal. The notation $\Sigma[a + n/n']$ indicates that the memory at point $a + n$ in store Σ has been updated, and now contains the value n'.

A statement $s$represents a transformation of the store, and statement evaluation $\Downarrow_s$ may also emit outputs and/or a memory trace. Hence $\Downarrow_s$: $\Sigma \Rightarrow \lambda \Rightarrow s \Rightarrow \Sigma \times \tau \times \omega$. Notably, statements are the only sources of program divergence (non-termination) in Core Covert.

An expression e computes an integer value n, and expression evaluation may emit outputs and/or a memory trace. That is, $\Downarrow_e$: $\Sigma \Rightarrow \lambda \Rightarrow e \Rightarrow n \times \tau \times \omega$. Addresses and integers are the only two data types in Core Covert. Interestingly, their separation is enforced by the syntax of the language, rather than the type system.

Types in Core Covert are security classes which form a lattice under $\leq$. Every lattice must have a bottom, denoted $\perp$. In the context of Core Covert and Covert C++, $\perp$ is the type assigned to data that is public (i.e., non-secret). Any data that is typed as $\perp$ may be disclosed to the adversary, and any data that is not $\perp$ must not be disclosed. Each Core Covert program has associated typing contexts $\Psi[\ell : \gamma]$ and $\phi[i : \gamma]$ for locations and inputs, respectively. Ψ is inductively defined over unevaluated locations:

$$\Psi(a + e) := \Phi(a),$$

where Φ assigns a type to each array. Hence every location within an array has the same type, i.e., the type of the array as dictated by Φ.

The typing rules for locations, expressions, statements, and programs are given in Figure 4.3. For example, typing judgments for expressions have the form $\Psi, \phi \vdash e : \gamma$, meaning that e can be assigned type γ in the typing context Ψ, ϕ. Inference rules are of the form

$$\frac{\begin{array}{c} \mathit{hypothesis}_1 \\ \vdots \\ \mathit{hypothesis}_n \end{array}}{\mathit{conclusion}}$$

such that *conclusion* is valid only when all hypotheses have been satisfied. Hence the ADDRESS-T typing rule states that the typing judgment $\Psi, \phi \vdash a + e : \gamma$ is valid only when the judgment $\Psi, \phi \vdash e : \gamma'$ is satisfied, where $\gamma' \leq \gamma$.

A typing judgment for a location or expression assigns to it a security class. On the other hand, a typing judgment for a program or statement only determines whether that program or statement is well-formed. That is, a program or statement is valid whenever all of its nested sub-locations and sub-expressions are typable, and all of its nested sub-statements are valid. These typing judgments are also given in Figure 4.3.

Figure 4.4 gives the inference rules for semantic judgments on the evaluation of locations, expressions, statements, and programs. For example, the judgment

$$\Sigma, \lambda \vdash a + e \Downarrow_{\ell} (a + n,\, \tau,\, \omega)$$

can be interpreted as, “the location $a + e$ evaluates to the address $a + n$ and emits traces τ and ω in the execution context Σ, λ.”

4.1.1 Design Considerations

The primary objective of Core Covert is to capture the essence of Covert C++, while omitting all of the pedantic details of C++ which make formal verification intractable. To that end, only the features of C++ which can potentially violate termination-sensitive noninterference were considered, and only the features of the Covert C++ type system which are required to enforce termination-sensitive noninterference were included in Core Covert.

The machine model for noninterference described in Chapter 2 is a reasonable approximation of execution for a compiled C++ program running on a conventional platform, such as x86 or ARM. Assuming that the adversary cannot observe the plaintext contents of program state (i.e., the memory and registers), the visible channels would include program outputs and aspects of the memory trace. Rather than model file descriptors, system calls, etc. in Core Covert, all storage channels have been reduced to a simple output(e) command, which emits an integer that is recorded in the output trace. Similarly, the treatment of all memory reads and writes has been reduced to two operations: read(ℓ), which reads the integer stored in the given location in memory, and assignment $\ell := e$, which stores a computed integer into

$$\textsc{Address-T}\;\frac{\begin{array}{c}\Psi,\phi \vdash e : \gamma' \\ \gamma' \leq \gamma\end{array}}{\Psi,\phi \vdash a + e : \gamma}$$

$$\textsc{Value-T}\;\frac{}{\Psi,\phi \vdash n : \bot} \qquad \textsc{Input-T}\;\frac{\begin{array}{c}\phi(i) = \gamma' \\ \gamma' \leq \gamma\end{array}}{\Psi,\phi \vdash i : \gamma}$$

$$\textsc{Arith-T}\;\frac{\begin{array}{c}\Psi,\phi \vdash e_1 : \gamma_1 \\ \Psi,\phi \vdash e_2 : \gamma_2 \\ \gamma_1 \sqcup \gamma_2 \leq \gamma\end{array}}{\Psi,\phi \vdash e_1 \odot e_2 : \gamma} \qquad \textsc{Ternary-T}\;\frac{\begin{array}{c}\Psi,\phi \vdash e_1 : \bot \\ \Psi,\phi \vdash e_2 : \gamma_2 \\ \Psi,\phi \vdash e_3 : \gamma_3 \\ \gamma_2 \sqcup \gamma_3 \leq \gamma\end{array}}{\Psi,\phi \vdash e_1 \mathrel{?} e_2 \circ e_3 : \gamma}$$

$$\textsc{Read-T}\;\frac{\begin{array}{c}\Psi,\phi \vdash \ell : \bot \\ \Psi(\ell) = \gamma' \\ \gamma' \leq \gamma\end{array}}{\Psi,\phi \vdash \mathsf{read}(\ell) : \gamma} \qquad \textsc{Output-T}\;\frac{\Psi,\phi \vdash e : \bot}{\Psi,\phi \vdash \mathsf{output}(e) : \bot}$$

$$\textsc{Skip-T}\;\frac{}{\Psi,\phi \vdash \mathsf{skip}\ \ \mathsf{valid}} \qquad \textsc{Assign-T}\;\frac{\begin{array}{c}\Psi,\phi \vdash \ell : \bot \\ \Psi,\phi \vdash e : \gamma \\ \Psi(\ell) = \gamma' \\ \gamma \leq \gamma'\end{array}}{\Psi,\phi \vdash \ell := e\ \ \mathsf{valid}}$$

$$\textsc{Sequence-T}\;\frac{\begin{array}{c}\Psi,\phi \vdash s_1\ \ \mathsf{valid} \\ \Psi,\phi \vdash s_2\ \ \mathsf{valid}\end{array}}{\Psi,\phi \vdash s_1; s_2\ \ \mathsf{valid}} \qquad \textsc{Iterate-T}\;\frac{\begin{array}{c}\Psi,\phi \vdash e : \bot \\ \Psi,\phi \vdash s\ \ \mathsf{valid}\end{array}}{\Psi,\phi \vdash \mathsf{iterate}\ e\ s\ \ \mathsf{valid}}$$

$$\textsc{Select-T}\;\frac{\begin{array}{c}\Psi,\phi \vdash e : \bot \\ \Psi,\phi \vdash s_1\ \ \mathsf{valid} \quad \cdots \quad \Psi,\phi \vdash s_m\ \ \mathsf{valid}\end{array}}{\Psi,\phi \vdash \mathsf{select}\ e\ \{s_1, \ldots, s_m\}\ \ \mathsf{valid}}$$

$$\textsc{Program-T}\;\frac{\Psi,\phi \vdash s\ \ \mathsf{valid}}{\Psi,\phi \vdash \mathsf{main}\ \{\ s\ \}\ \ \mathsf{valid}}$$

Figure 4.3: Core Covert type inference rules

$$\textsc{Address}\ \frac{\Sigma, \lambda \vdash e \Downarrow_e (n,\, \tau,\, \omega)}{\Sigma, \lambda \vdash a + e \Downarrow_\ell (a + n,\, \tau,\, \omega)}$$

$$\textsc{Value}\ \frac{}{\Sigma, \lambda \vdash n \Downarrow_e (n,\, \cdot,\, \cdot)} \qquad \textsc{Input}\ \frac{}{\Sigma, \lambda \vdash i \Downarrow_e (\lambda(i),\, \cdot,\, \cdot)}$$

$$\textsc{Read}\ \frac{\Sigma, \lambda \vdash \ell \Downarrow_\ell (a + n,\, \tau,\, \omega)}{\Sigma, \lambda \vdash \mathsf{read}(\ell) \Downarrow_e (\Sigma(a + n),\, \tau \triangleright \alpha(a + n),\, \omega)}$$

$$\textsc{Arith}\ \frac{\Sigma, \lambda \vdash e_1 \Downarrow_e (n_1,\, \tau,\, \omega) \qquad \Sigma, \lambda \vdash e_2 \Downarrow_e (n_2,\, \tau',\, \omega')}{\Sigma, \lambda \vdash e_1 \odot e_2 \Downarrow_e (n_1 \odot n_2,\, \tau \triangleright \tau',\, \omega \triangleright \omega')}$$

$$\textsc{Ternary (1)}\ \frac{\Sigma, \lambda \vdash e_1 \Downarrow_e (n_1,\, \tau,\, \omega) \qquad \Sigma, \lambda \vdash e_2 \Downarrow_e (n_2,\, \tau',\, \omega') \qquad n_1 \neq 0}{\Sigma, \lambda \vdash e_1\ ?\ e_2 \circ e_3 \Downarrow_e (n_2,\, \tau \triangleright \alpha(e_2) \triangleright \tau',\, \omega \triangleright \omega')}$$

$$\textsc{Ternary (2)}\ \frac{\Sigma, \lambda \vdash e_1 \Downarrow_e (n_1,\, \tau,\, \omega) \qquad \Sigma, \lambda \vdash e_3 \Downarrow_e (n_3,\, \tau',\, \omega') \qquad n_1 = 0}{\Sigma, \lambda \vdash e_1\ ?\ e_2 \circ e_3 \Downarrow_e (n_3,\, \tau \triangleright \alpha(e_3) \triangleright \tau',\, \omega \triangleright \omega')}$$

$$\textsc{Output}\ \frac{\Sigma, \lambda \vdash e \Downarrow_e (n,\, \tau,\, \omega)}{\Sigma, \lambda \vdash \mathsf{output}(e) \Downarrow_e (0,\, \tau,\, \omega \triangleright n)} \qquad \textsc{Skip}\ \frac{}{\Sigma, \lambda \vdash \mathsf{skip} \Downarrow_s (\Sigma,\, \cdot,\, \cdot)}$$

$$\textsc{Assign}\ \frac{\Sigma, \lambda \vdash \ell \Downarrow_\ell (a + n,\, \tau,\, \omega) \qquad \Sigma, \lambda \vdash e \Downarrow_e (n',\, \tau',\, \omega')}{\Sigma, \lambda \vdash \ell := e \Downarrow_s (\Sigma[a + n/n'],\, \tau \triangleright \tau' \triangleright \alpha(a + n),\, \omega \triangleright \omega')}$$

$$\textsc{Sequence}\ \frac{\Sigma, \lambda \vdash s_1 \Downarrow_s (\Sigma'',\, \tau,\, \omega) \qquad \Sigma'', \lambda \vdash s_2 \Downarrow_s (\Sigma',\, \tau',\, \omega')}{\Sigma, \lambda \vdash s_1; s_2 \Downarrow_s (\Sigma',\, \tau \triangleright \tau',\, \omega \triangleright \omega')}$$

$$\textsc{Iterate (1)}\ \frac{\Sigma, \lambda \vdash e \Downarrow_e (n,\, \tau,\, \omega) \qquad n = 0}{\Sigma, \lambda \vdash \mathsf{iterate}\ e\ s \Downarrow_s (\Sigma,\, \tau,\, \omega)}$$

$$\textsc{Iterate (2)}\ \frac{\begin{array}{c} \Sigma, \lambda \vdash e \Downarrow_e (n,\, \tau,\, \omega) \qquad n \neq 0 \\ \Sigma, \lambda \vdash s \Downarrow_s (\Sigma'',\, \tau',\, \omega') \\ \Sigma'', \lambda \vdash \mathsf{iterate}\ e\ s \Downarrow_s (\Sigma',\, \tau'',\, \omega'') \end{array}}{\Sigma, \lambda \vdash \mathsf{iterate}\ e\ s \Downarrow_s (\Sigma',\, \tau \triangleright \alpha(s) \triangleright \tau' \triangleright \tau'',\, \omega \triangleright \omega' \triangleright \omega'')}$$

$$\textsc{Select}\ \frac{\begin{array}{c} \Sigma, \lambda \vdash e \Downarrow_e (n,\, \tau,\, \omega) \\ n \in \{1, \ldots, m\} \\ \Sigma, \lambda \vdash s_n \Downarrow_s (\Sigma',\, \tau',\, \omega') \end{array}}{\Sigma, \lambda \vdash \mathsf{select}\ e\ \{s_1, \ldots, s_m\} \Downarrow_s (\Sigma',\, \tau \triangleright \alpha(s_n) \triangleright \tau',\, \omega \triangleright \omega')}$$

$$\textsc{Program}\ \frac{\Sigma, \lambda \vdash s \Downarrow_s (\Sigma',\, \tau,\, \omega)}{\Sigma, \lambda \vdash \mathsf{main}\ \{\ s\ \} \Downarrow_\Pi (\tau,\, \omega)}$$

Figure 4.4: Core Covert semantics

a given memory location. These accesses are all recorded into the memory trace.

The treatment of program control flow is more nuanced. C++ has several constructs which determine control flow, and these cannot all be easily grouped into a single representative command, although some of them can. The C++ standard [90] partitions the syntax-directed control statements `if`/`else if`/`else`, `for`, `while`, `do while`, and `switch` into one of two categories: *selection* statements or *iteration* statements. Selection statements "choose one of several flows of control" [90], whereas iteration statements "specify looping" [90] for a given statement, subject to a termination condition. The `if`/`else if`/`else` and `switch` are selection statements, and `for`, `while`, and `do while` are iteration statements. The Select and Iterate (1,2) rules generalize the behavior of these control flow statements. Moreover, they encode the control flow logic in the memory trace. For example, the address of the statement s_n chosen by a select statement is concatenated to the memory trace after s_n is chosen, but before s_n is executed.

Control flow in C++ can also be introduced by more subtle means. For instance, the binary logical operators `&&` and `||` have a short-circuiting behavior which elides evaluation of the right-hand side if the result of evaluating the left-hand side determines the value of the expression. If the operands are sufficiently simple (e.g., each operand is an `int` already stored in a register), then the compiled expression may not involve a branch. If the right-hand side is more complex (e.g., it makes a function call), then the logical operator must make a branch decision. The ternary operator `?:` has similar behavior. If the first operand evaluates to true, then only the second operand is evaluated. Otherwise, only the third operand is evaluated. For brevity, only the ternary operator is modeled in Core Covert, by the Ternary (1,2) rules. Similar to the selection and iteration statements, when a ternary is computed, the chosen expression is recorded into the memory trace after it is chosen, but before it is evaluated.

One control-flow feature of C++ which is not modeled in Core Covert is dynamic dispatch, the process of looking up a procedure call in a virtual function table at runtime. This feature is somewhat complex, and as argued later in Chapter 5, Covert C++ typing rules for pointers are sufficient to prevent a dynamic dispatch that would reveal sensitive information through the memory trace.

The construction of memory traces arising from control flow may seem overly simplified. An alternative formulation could have included every statement and expression in the memory trace. This is unnecessary. In Core Covert, given any memory trace sequence of the form

$$\alpha(r) \triangleright \alpha(a_1 + n_1) \triangleright \cdots \triangleright \alpha(a_m + n_m) \triangleright \alpha(r'),$$

where $r ::= e \mid s$, the subsequence

$$\alpha(r) \triangleright \alpha(a_1 + n_1) \triangleright \cdots \triangleright \alpha(a_m + n_m),$$

corresponds to a compiler basic block, and $\alpha(r')$ is the beginning of the next basic block. "A *basic block* is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed)" [22]. Hence the additional statements or expressions within the basic block do not reveal anything about the control flow of the program; they are invariably executed in the exact same sequence. The noninterference verification tool introduced in Chapter 6 makes a similar simplification, but for dynamic basic blocks.

Just as in a programming language like C++, all expressions and locations (pointers) are typed. In Core Covert, the type assigned to an expression or location establishes an upper bound on the security class of the data "flowing" from the expression or location when it is resolved to an integer or point during program execution. Ideally, no source of information in an expression (e.g., an input) should have a security class which exceeds the security class of the expression. The following theorem establishes this property for Core Covert.

Theorem 4.1 (Simple Security)**.**

1. *If $\Psi, \phi \vdash \ell : \gamma$, then for every ℓ' and i in ℓ, $\Psi(\ell') \leq \gamma$ and $\phi(i) \leq \gamma$.*
2. *If $\Psi, \phi \vdash e : \gamma$, then for every ℓ and i in e, $\Psi(\ell) \leq \gamma$ and $\phi(i) \leq \gamma$.*

Proof. By mutual induction on the structure of e and ℓ. The cases for i and n are trivial.

$\boxed{a + e}$ Suppose that $\Psi, \phi \vdash a + e : \gamma$ by rule ADDRESS-T. Then $\Psi, \phi \vdash e : \gamma'$, and by induction $\Psi(\ell) \leq \gamma'$ and $\phi(i) \leq \gamma'$ for every ℓ and i in e. Since $\gamma' \leq \gamma$ and $\leq$ is transitive, it follows that $\Psi(\ell) \leq \gamma$ and $\phi(i) \leq \gamma$ for every ℓ and i in $a + e$.

$\boxed{e_1 \odot e_2}$ Suppose that $\Psi, \phi \vdash e_1 \odot e_2 : \gamma$ by rule ARITH-T. Then $\Psi, \phi \vdash e_1 : \gamma_1$ and $\Psi, \phi \vdash e_2 : \gamma_2$. Hence by induction $\Psi(\ell) \leq \gamma_1$ and $\phi(i) \leq \gamma_1$ for every ℓ and i in e_1, and similarly $\Psi(\ell) \leq \gamma_2$ and $\phi(i) \leq \gamma_2$ for every ℓ and i in e_2. Since $\gamma_1 \sqcup \gamma_2 \leq \gamma$, by the transitive property $\Psi(\ell) \leq \gamma$ and $\phi(i) \leq \gamma$ for every ℓ and i in $e_1 \odot e_2$.

$\boxed{e_1 \ ? \ e_2 \circ e_3}$ Suppose that $\Psi, \phi \vdash e_1 \ ? \ e_2 \circ e_3 : \gamma$ by rule TERNARY-T. Then $\Psi, \phi \vdash e_1 : \bot$, $\Psi, \phi \vdash e_2 : \gamma_2$, and $\Psi, \phi \vdash e_3 : \gamma_3$. By induction $\Psi(\ell) \leq \bot$ and $\phi(i) \leq \bot$ for every ℓ and i in e_1, $\Psi(\ell) \leq \gamma_2$ and $\phi(i) \leq \gamma_2$ for every ℓ and i in e_2, and similarly for e_3. Since $\bot \leq \gamma$ and $\gamma_2 \sqcup \gamma_3 \leq \gamma$, by the transitive property $\Psi(\ell) \leq \gamma$ and $\phi(i) \leq \gamma$ for every ℓ and i in $e_1 \odot e_2$.

$\boxed{\mathsf{read}(\ell)}$ Suppose that $\Psi, \phi \vdash \mathsf{read}(\ell) : \gamma$ by READ-T. Then $\Psi(\ell) = \gamma'$ with $\gamma' \leq \gamma$, and $\Psi, \phi \vdash \ell : \bot$. By induction $\Psi(\ell') \leq \bot$ and $\phi(i) \leq \bot$ for every ℓ' and i in ℓ. Thus $\Psi(\ell') \leq \gamma$ and $\phi(i) \leq \gamma$ for every ℓ' and i in $\mathsf{read}(\ell)$, including ℓ itself.

$\boxed{\mathsf{output}(e)}$ Finally, suppose that $\Psi, \phi \vdash \mathsf{output}(e) : \bot$ by OUTPUT-T. Then $\Psi(e) = \bot$, and by induction $\Psi(\ell) \leq \bot$ and $\phi(i) \leq \bot$ for every ℓ and i in e. Thus $\Psi(\ell) \leq \bot$ and $\phi(i) \leq \bot$ for every ℓ and i in $\mathsf{output}(e)$. $\square$

Now, to quote from Chapter 1 of this dissertation, "A general solution to enforce noninterference should at least have the following three characteristics: it should (1) allow the program author to annotate or otherwise distinguish between secret and public data, (2) provide a mechanism to track the propagation of secret data through the program, and (3) implement constraints on secret data to prevent it from being leaked through either a side channel or a storage channel." Condition (1) is satisfied by Core Covert's security-type system, which requires that all inputs and arrays must be typed according to security class, i.e., all points and inputs referenced by a given program must be typed by Ψ and ϕ, respectively. Although

Core Covert does not model type inference (this is complicated, see [162]), Theorem 4.1 does place an upper bound on the security class that can be propagated from an evaluated expression, e.g., to memory or an output. Thus (2) is satisfied.

The final ingredient is the information-flow policy, the rules which describe the security requirement. There are essentially three means by which noninterference can be violated: a leak of sensitive information through a program control-flow decision (including non-termination), a memory access, or an output to a publicly-observable channel. First consider control flow. The only procedures in Core Covert which make control-flow decisions are the ternary expression, select, and $\mathsf{iterate}$. For each of these, the expression which determines the branch is required to have type $\perp$. Hence, with Theorem 4.1, non-$\perp$ information cannot influence the branch decision. This constraint is encoded in the typing rules TERNARY-T, SELECT-T, and ITERATE-T.

Next, consider memory accesses. The only procedures which allow memory accesses are the $\mathsf{read}(\ell)$ expression and the $\ell := e$ (assignment) statement. By typing rules READ-T and ASSIGN-T, ℓ must have type $\perp$. Thus for the same reasons as before, sensitive data cannot influence a memory access location. The final case is program outputs. Only one expression can write to the output trace: $\mathsf{output}(e)$. Again, typing rule OUTPUT-T requires that e have type $\perp$. Hence the type inference rules in Figure 4.3 are sufficient to enforce noninterference. The following section formalizes this argument.

4.2 Core Covert Enforces Noninterference

This section presents a formal proof that Core Covert statically enforces a form of termination-sensitive noninterference. That is, non-$\perp$ program inputs noninterfere with $\perp$ program outputs, including the memory trace. Moreover, non-$\perp$ inputs do not affect whether or not a Core Covert program terminates. The first lemma establishes that a computed integer value or address of type γ is entirely determined by value(s) of type $\leq \gamma$.

Lemma 4.2 (Simple Evaluation)**.** *Let $dom(\Sigma) = dom(\Sigma') = dom(\lambda) = dom(\lambda') = dom(\Psi) = dom(\phi)$, with* (I) *$\lambda(i) = \lambda'(i)$ for all i such that $\phi(i) \leq \gamma$. Furthermore, suppose that* (II) *$\Sigma(a+n) = \Sigma'(a+n)$ for all $a+n$ such that $\Psi(a+n) \leq \gamma$.*

1. *Suppose*

   (a) $\Psi, \phi \vdash \ell : \gamma'$,

   (b) $\gamma' \leq \gamma$,

   (c) $\Sigma, \lambda \vdash \ell \Downarrow_\ell (a+n,\, \tau, \omega)$, *and*

   (d) $\Sigma', \lambda' \vdash \ell \Downarrow_\ell (a'+n',\, \tau', \omega')$.

   Then $a+n = a'+n'$.

2. *Suppose*

   (a) $\Psi, \phi \vdash e : \gamma'$,

   (b) $\gamma' \leq \gamma$,

   (c) $\Sigma, \lambda \vdash e \Downarrow_e (n,\, \tau, \omega)$, *and*

   (d) $\Sigma', \lambda' \vdash e \Downarrow_e (n',\, \tau', \omega')$.

   Then $n = n'$.

Proof. By mutual induction on the structure of e and ℓ. The cases for n and $\mathsf{output}(\ell)$ are trivial.

$\boxed{i}$ Suppose that $\Sigma, \lambda \vdash i \Downarrow_e (\lambda(i),\, \cdot,\, \cdot)$ and $\Sigma', \lambda' \vdash i \Downarrow_e (\lambda'(i),\, \cdot,\, \cdot)$ by rule Input. By hypothesis 2(a) and rule Input-T, there exists some γ'' such that $\phi(i) = \gamma''$ and $\gamma'' \leq \gamma'$. Since $\leq$ is transitive, $\phi(i) \leq \gamma$ using hypothesis 2(b). By hypothesis (I), $\lambda(i) = \lambda'(i)$.

$\boxed{a+e}$ Suppose that $\Sigma, \lambda \vdash a{+}e \Downarrow_e (a{+}n,\, \tau,\, \omega)$ and $\Sigma,' \lambda' \vdash a{+}e \Downarrow_e (a{+}n',\, \tau',\, \omega')$ by rule Address, where $\Sigma, \lambda \vdash e \Downarrow_e (n,\, \tau,\, \omega)$ and $\Sigma,' \lambda' \vdash e \Downarrow_e (n',\, \tau',\, \omega')$. By hypothesis 2(a) and rule Address-T, there exists some γ'' such that $\Psi, \phi \vdash e : \gamma''$ and $\gamma'' \leq \gamma'$. By the transitive property of $\leq$ and hypothesis 2(b), $\gamma'' \leq \gamma$, and by induction $n = n'$. Thus $a+n = a+n'$.

$\boxed{e_1 \odot e_2}$ This case follows similarly to $a+e$, except that induction is applied once for each of e_1 and e_2.

$\boxed{e_1 \ ?\ e_2 \circ e_3}$ By hypothesis 2(a) and rule TERNARY-T, $\Psi, \phi \vdash e_1 : \bot$. There are two cases to consider:

1. $\Sigma, \lambda \vdash e_1 \ ?\ e_2 \circ e_3 \Downarrow_e (n_2,\ \tau_1 \triangleright \alpha(e_2) \triangleright \tau_2,\ \omega_1 \triangleright \omega_2)$ by rule TERNARY (1) on hypothesis 2(c). Then TERNARY (1) must also apply to hypothesis 2(d) because, by induction, $\Sigma, \lambda \vdash e_1 \Downarrow_e (n_1,\ \tau,\ \omega)$ and $\Sigma', \lambda' \vdash e_1 \Downarrow_e (n_1',\ \tau',\ \omega')$ implies that $n_1 = n_1'$. Hence $\Sigma', \lambda' \vdash e_1 \ ?\ e_2 \circ e_3 \Downarrow_e (n_2',\ \tau_1' \triangleright \alpha(e_2) \triangleright \tau_2',\ \omega_1' \triangleright \omega_2')$. Moreover, by hypothesis 2(a) and TERNARY-T, there exists some γ_2 such that $\Psi, \phi \vdash e_2 : \gamma_2$ and $\gamma_2 \leq \gamma'$. By hypothesis 2(b), $\gamma_2 \leq \gamma$. Thus by another use of induction on e_2, $n_2 = n_2'$.
2. $\Sigma, \lambda \vdash e_1 \ ?\ e_2 \circ e_3 \Downarrow_e (n_3,\ \tau_1 \triangleright \alpha(e_2) \triangleright \tau_3,\ \omega_1 \triangleright \omega_3)$ by rule TERNARY (2) on hypothesis 2(c). This case follows similarly.

$\boxed{\mathsf{read}(\ell)}$ Suppose that $\Sigma, \lambda \vdash \mathsf{read}(\ell) \Downarrow_e (\Sigma(a + n),\ \tau \triangleright \alpha(a + n),\ \omega)$ and $\Sigma', \lambda' \vdash \mathsf{read}(\ell) \Downarrow_e (\Sigma'(a' + n'),\ \tau' \triangleright \alpha(a' + n'),\ \omega')$ by rule READ, where $\Sigma, \lambda \vdash \ell \Downarrow_\ell (a + n,\ \tau,\ \omega)$ and $\Sigma', \lambda' \vdash \ell \Downarrow_\ell (a' + n',\ \tau',\ \omega')$. By hypothesis 1(a) and rule READ-T, there exists some γ'' such that $\Psi(\ell) = \gamma''$ and $\gamma'' \leq \gamma'$, and $\Psi, \phi \vdash \ell : \bot$. Hence by induction, $a + n = a' + n'$. By the transitive property and hypothesis 1(b), $\Psi(\ell) \leq \gamma$, thus $\Sigma(a + n) = \Sigma'(a' + n')$ by hypothesis (II). □

The next lemma establishes that memory and output traces emitted during the evaluation of an expression are entirely determined by $\bot$ inputs.

Lemma 4.3 (Expression Noninterference)**.** *Let* $dom(\Sigma) = dom(\Sigma') = dom(\lambda) = dom(\lambda') = dom(\Psi) = dom(\phi)$*, with* (I) $\lambda(i) = \lambda'(i)$ *for all* i *such that* $\phi(i) = \bot$*. Furthermore, suppose that* (II) $\Sigma(a + n) = \Sigma'(a + n)$ *for all* $a + n$ *such that* $\Psi(a + n) = \bot$*.*

1. *Suppose*
   (a) $\Psi, \phi \vdash \ell : \gamma$ *(ℓ is typable in Ψ and ϕ),*
   (b) $\Sigma, \lambda \vdash \ell \Downarrow_\ell (a + n,\ \tau, \omega)$*, and*
   (c) $\Sigma', \lambda' \vdash \ell \Downarrow_\ell (a' + n',\ \tau', \omega')$*.*

Then $\tau \equiv \tau'$ and $\omega \doteq \omega'$.

2. *Suppose*

   (a) $\Psi, \phi \vdash e : \gamma$ *(e is typable in Ψ and ϕ),*

   (b) $\Sigma, \lambda \vdash e \Downarrow_e (n,\, \tau,\, \omega)$, *and*

   (c) $\Sigma', \lambda' \vdash e \Downarrow_e (n',\, \tau',\, \omega')$.

   Then $\tau \equiv \tau'$ and $\omega \doteq \omega'$.

Proof. By mutual induction on the structure of e and ℓ. The cases for i and n are trivial.

$\boxed{a + e}$ Suppose that $\Sigma, \lambda \vdash a + e \Downarrow_\ell (a + n,\, \tau,\, \omega)$ and $\Sigma', \lambda' \vdash a + e \Downarrow_\ell (a + n',\, \tau',\, \omega')$ by rule Address, where $\Sigma, \lambda \vdash e \Downarrow_\ell (n,\, \tau,\, \omega)$ and $\Sigma', \lambda' \vdash e \Downarrow_\ell (n',\, \tau',\, \omega')$. Thus $\tau \equiv \tau'$ and $\omega \doteq \omega'$ by induction.

$\boxed{e_1 \odot e_2}$ Suppose that $\Sigma, \lambda \vdash e_1 \odot e_2 \Downarrow_e (n_1 \odot n_2,\, \tau_1 \triangleright \tau_2,\, \omega_1 \triangleright \omega_2)$ and $\Sigma', \lambda' \vdash e_1 \odot e_2 \Downarrow_e (n_1' \odot n_2',\, \tau_1' \triangleright \tau_2',\, \omega_1' \triangleright \omega_2')$ by rule Arith. By two applications of induction, $\tau_1 \equiv \tau_1'$ and $\tau_2 \equiv \tau_2'$, hence $\tau_1 \triangleright \tau_2 \equiv \tau_1' \triangleright \tau_2'$ by Mem-Concat. $\omega_1 \triangleright \omega_2 \doteq \omega_1' \triangleright \omega_2'$ follows similarly with Out-Concat.

$\boxed{e_1 \;?\; e_2 \circ e_3}$ By hypothesis 2(a) and rule Ternary-T, $\Psi, \phi \vdash e_1 : \bot$. There are two cases to consider:

1. $\Sigma, \lambda \vdash e_1 \;?\; e_2 \circ e_3 \Downarrow_e (n_2,\, \tau_1 \triangleright \alpha(e_2) \triangleright \tau_2,\, \omega_1 \triangleright \omega_2)$ by rule Ternary (1) on hypothesis 2(b). Then Ternary (1) must also apply to hypothesis 2(d) because, by Lemma 4.2 with hypotheses (I) and (II), $\Sigma, \lambda \vdash e_1 \Downarrow_e (n_1,\, \tau,\, \omega)$ and $\Sigma', \lambda' \vdash e_1 \Downarrow_e (n_1',\, \tau',\, \omega')$ implies $n_1 = n_1'$, so $n_1' \neq 0$. Hence $\Sigma', \lambda' \vdash e_1 \;?\; e_2 \circ e_3 \Downarrow_e (n_2',\, \tau_1' \triangleright \alpha(e_2) \triangleright \tau_2',\, \omega_1' \triangleright \omega_2')$. By two applications of induction, $\tau_1 \equiv \tau_1'$ and $\tau_2 \equiv \tau_2'$, hence $\tau_1 \triangleright \alpha(e_2) \triangleright \tau_2 \equiv \tau_1' \triangleright \alpha(e_2) \triangleright \tau_2'$ by Mem-Concat and Mem-Expr. $\omega_1 \triangleright \omega_2 \doteq \omega_1' \triangleright \omega_2'$ follows similarly with Out-Concat.
2. $\Sigma, \lambda \vdash e_1 \;?\; e_2 \circ e_3 \Downarrow_e (n_3,\, \tau_1 \triangleright \alpha(e_2) \triangleright \tau_3,\, \omega_1 \triangleright \omega_3)$ by rule Ternary (2) on hypothesis 2(b). This case follows similarly.

$\boxed{\mathsf{read}(\ell)}$ Suppose that $\Sigma, \lambda \vdash \mathsf{read}(\ell) \Downarrow_e (n,\, \tau \triangleright \alpha(a + n'),\, \omega)$ and $\Sigma, \lambda' \vdash \mathsf{read}(\ell) \Downarrow_e (n',\, \tau' \triangleright \alpha(a' + n''),\, \omega')$ by rule Read, where $\Sigma, \lambda \vdash \ell \Downarrow_e (a + n',\, \tau,\, \omega)$ and $\Sigma', \lambda' \vdash \ell \Downarrow_e$

$(a' + n'', \tau', \omega')$. By rule READ-T, $\Psi, \phi \vdash \ell : \bot$. Hence $a + n' = a' + n''$ by Lemma 4.2 with hypotheses (I) and (II). By an application of induction, $\tau \equiv \tau'$ and $\omega \doteq \omega'$, thus $\tau \triangleright \alpha(a + n') \equiv \tau' \triangleright \alpha(a' + n'')$ by MEM-CONCAT and MEM-ADDR.

$\boxed{\mathsf{output}(e)}$ Suppose that $\Sigma, \lambda \vdash \mathsf{output}(e) \Downarrow_e (0, \tau, \omega \triangleright n)$ and $\Sigma, \lambda' \vdash \mathsf{output}(e) \Downarrow_e (0, \tau,' \omega' \triangleright n')$ by rule OUTPUT, where $\Sigma, \lambda \vdash e \Downarrow_e (n, \tau, \omega)$ and $\Sigma', \lambda' \vdash e \Downarrow_e (n', \tau', \omega')$. By rule READ-T, $\Psi, \phi \vdash e : \bot$, thus $n = n'$ by Lemma 4.2 and hypotheses (I) and (II). By an application of induction, $\tau \equiv \tau'$ and $\omega \triangleright n \doteq \omega' \triangleright n'$ by OUT-CONCAT and OUT-VALUE. □

Recall that statements in Core Covert transform the program store Σ. The Memory Confidentiality lemma proves a strict confidentiality property for program state transformations. In effect, arrays of type γ cannot be tainted by information of a security class greater than γ. This is essentially the Bell LaPadula "no read up, no write down" model of confidentiality [32].

Lemma 4.4 (Memory Confidentiality). *Let dom*(Σ) = *dom*(Σ') = *dom*(λ) = *dom*(λ') = *dom*(Ψ) = *dom*(ϕ)*, with* (I) $\lambda(i) = \lambda'(i)$ *for all* i *such that* $\phi(i) \leq \gamma$*. Suppose further that* (II) $\Sigma(a + n) = \Sigma'(a + n)$ *for all* $a + n$ *such that* $\Psi(a + n) \leq \gamma$*. If*

1. $\Psi, \phi \vdash s$ valid*,*
2. $\Sigma, \lambda \vdash s \Downarrow_s (\Sigma'', \tau, \omega)$*, and*
3. $\Sigma', \lambda' \vdash s \Downarrow_s (\Sigma''', \tau', \omega')$*,*

then $\Sigma''(a + n) = \Sigma'''(a + n)$ *for all* $a + n$ *such that* $\Psi(a + n) \leq \gamma$*.*

Proof. By induction on the structure of the derivation of $\Sigma, \lambda \vdash s \Downarrow_s (\Sigma', \tau, \omega)$. The case for skip is trivial.

$\boxed{\ell := e}$ Suppose the evaluation under Σ and λ ends with

$$\frac{\Sigma, \lambda \vdash \ell \Downarrow_\ell (a_1 + n_1, \tau_1, \omega_1) \qquad \Sigma, \lambda \vdash e \Downarrow_e (n_1' \, \tau_1', \omega_1')}{\Sigma, \lambda \vdash \ell := e \Downarrow_s (\Sigma[a_1 + n_1/n_1'], \tau_1 \triangleright \tau_1' \triangleright \alpha(a_1 + n_1), \omega_1 \triangleright \omega_1')}$$

and the evaluation under Σ' and λ' ends with

$$\frac{\Sigma', \lambda' \vdash \ell \Downarrow_\ell (a_2 + n_2, \tau_2, \omega_2) \qquad \Sigma', \lambda' \vdash e \Downarrow_e (n_2' \, \tau_2', \omega_2')}{\Sigma', \lambda' \vdash \ell := e \Downarrow_s (\Sigma'[a_2 + n_2/n_2'], \tau_2 \triangleright \tau_2' \triangleright \alpha(a_2 + n_2), \omega_2 \triangleright \omega_2')}$$

and typing ends with an application of ASSIGN-T:

$$\frac{\begin{array}{c}\Psi, \phi \vdash \ell : \bot \\ \Psi, \phi \vdash e : \gamma' \\ \Psi(\ell) = \gamma'' \\ \gamma' \leq \gamma''\end{array}}{\Psi, \phi \vdash \ell := e \ \textsf{valid}}$$

First, note that since $\Psi, \phi \vdash \ell : \bot$, by Lemma 4.2 and hypotheses (I) and (II), $a_1 + n_1 = a_2 + n_2$. Next, there are two cases to consider:

1. $\gamma'' \not\leq \gamma$: Therefore $\Psi(a_1 + n_1) \not\leq \gamma$ and $\Psi(a_2 + n_2) \not\leq \gamma$, so the assignment does not update a location $\leq \gamma$ in memory. That is, for all $a + n$ such that $\Psi(a + n) \leq \gamma$, $\Sigma(a+n) = \Sigma[a_1+n_1/n_1'](a+n)$ and $\Sigma'(a+n) = \Sigma'[a_2+n_2/n_2'](a+n)$. Moreover, because $\Sigma(a + n) = \Sigma'(a + n)$ by hypothesis (II), $\Sigma[a_1 + n_1/n_1'](a + n) = \Sigma'[a_2 + n_2/n_2'](a + n)$.
2. $\gamma'' \leq \gamma$: By ASSIGN-T and transitivity, $\gamma' \leq \gamma$. Furthermore, by Lemma 4.2 and hypotheses (I) and (II), $n_1' = n_2'$. So in either evaluation, the same address is updated with the same value. That is, for all $a+n$ such that $\Psi(a+n) = \bot$, $\Sigma[a_1+n_1/n_1'](a+n) = \Sigma'[a_2 + n_2/n_2'](a + n)$ using hypothesis (II).

$\boxed{s_1; s_2}$ Trivial, by two applications of induction on a derivation ending with the SEQUENCE rule.

$\boxed{\textsf{iterate}\ e\ s}$ Suppose that the typing ends with an application of the rule ITERATE-T:

$$\frac{\begin{array}{c}\Psi, \phi \vdash e : \bot \\ \Psi, \phi \vdash s \ \textsf{valid}\end{array}}{\Psi, \phi \vdash \textsf{iterate}\ e\ s \ \textsf{valid}}$$

Since $\Psi, \phi \vdash e : \bot$, evaluation of e under Σ and λ must produce the same value as evaluation under Σ' and λ'. That is, $\Sigma, \lambda \vdash e \Downarrow_e (n, \tau, \omega)$ and $\Sigma', \lambda' \vdash e \Downarrow_e (n', \tau', \omega')$ implies that $n = n'$ by Lemma 4.2 with hypotheses (I) and (II). Hence there are two cases to consider:

1. $n = 0$. Then the evaluations under Σ and λ, and Σ' and λ' each end with an application of rule ITERATE (1). The conclusion for this case is trivial, because ITERATE (1) does not transform the store.
2. $n \neq 0$: Suppose that the evaluation under Σ and λ ends with an application of rule

ITERATE (2):

$$\frac{\begin{array}{c}\Sigma, \lambda \vdash e \Downarrow_e (n_1,\, \tau_1,\, \omega_1) \qquad n_1 \neq 0 \\ \Sigma, \lambda \vdash s \Downarrow_s (\Sigma_1,\, \tau_1',\, \omega_1') \\ \Sigma_1, \lambda \vdash \mathsf{iterate}\ e\ s \Downarrow_s (\Sigma'',\, \tau_1'',\, \omega_1'')\end{array}}{\Sigma, \lambda \vdash \mathsf{iterate}\ e\ s \Downarrow_s (\Sigma'',\, \tau_1 \triangleright \alpha(s) \triangleright \tau_1' \triangleright \tau_1'',\, \omega_1 \triangleright \omega_1' \triangleright \omega_1'')}$$

and the evaluation under Σ' and λ' also ends with an application of ITERATE (2):

$$\frac{\begin{array}{c}\Sigma', \lambda' \vdash e \Downarrow_e (n_2,\, \tau_2,\, \omega_2) \qquad n_2 \neq 0 \\ \Sigma', \lambda' \vdash s \Downarrow_s (\Sigma_2,\, \tau_2',\, \omega_2') \\ \Sigma_2, \lambda' \vdash \mathsf{iterate}\ e\ s \Downarrow_s (\Sigma''',\, \tau_2'',\, \omega_2'')\end{array}}{\Sigma', \lambda' \vdash \mathsf{iterate}\ e\ s \Downarrow_s (\Sigma''',\, \tau_2 \triangleright \alpha(s) \triangleright \tau_2' \triangleright \tau_2'',\, \omega_2 \triangleright \omega_2' \triangleright \omega_2'')}$$

By induction, $\Sigma_1(a+n) = \Sigma_2(a+n)$ for all $a+n$ such that $\Psi(a+n) \leq \gamma$. By another application of induction, $\Sigma''(a+n) = \Sigma'''(a+n)$ for all $a+n$ such that $\Psi(a+n) \leq \gamma$.

$\boxed{\mathsf{select}\ e\ \{s_1, \ldots, s_m\}}$ Suppose that the evaluation under Σ and λ ends with

$$\frac{\begin{array}{c}\Sigma, \lambda \vdash e \Downarrow_e (n_1,\, \tau_1,\, \omega_1) \\ n_1 \in \{1, \ldots, m\} \\ \Sigma, \lambda \vdash s_{n_1} \Downarrow_s (\Sigma'',\, \tau_1',\, \omega_1')\end{array}}{\Sigma', \lambda' \vdash \mathsf{select}\ e\ \{s_1, \ldots, s_m\} \Downarrow_s (\Sigma'',\, \tau_1 \triangleright \alpha(s_{n_1}) \triangleright \tau_1',\, \omega_1 \triangleright \omega_1')}$$

and the evaluation under Σ' and λ' ends with

$$\frac{\begin{array}{c}\Sigma', \lambda' \vdash e \Downarrow_e (n_2,\, \tau_2,\, \omega_2) \\ n_2 \in \{1, \ldots, m\} \\ \Sigma', \lambda' \vdash s_{n_2} \Downarrow_s (\Sigma''',\, \tau_2',\, \omega_2')\end{array}}{\Sigma', \lambda' \vdash \mathsf{select}\ e\ \{s_1, \ldots, s_m\} \Downarrow_s (\Sigma''',\, \tau_2 \triangleright \alpha(s_{n_2}) \triangleright \tau_2',\, \omega_2 \triangleright \omega_2')}$$

and the typing ends with an application of the rule SELECT-T:

$$\frac{\begin{array}{c}\Psi, \phi \vdash e : \bot \\ \Psi, \phi \vdash s_1\ \mathsf{valid} \quad \cdots \quad \Psi, \phi \vdash s_m\ \mathsf{valid}\end{array}}{\Psi, \phi \vdash \mathsf{select}\ e\ \{s_1, \ldots, s_m\}\ \mathsf{valid}}$$

By Lemma 4.2 with hypotheses (I) and (II) applied to the evaluation of e, $n_1 = n_2$. Then by an application of induction on the evaluation of s_{n_1} and s_{n_2} over Σ, λ and Σ', λ', respectively, $\Sigma''(a+n) = \Sigma'''(a+n)$ for all $a+n$ such that $\Psi(a+n) \leq \gamma$. □

The next lemma establishes that memory and output traces emitted during the evaluation of a statement are entirely determined by $\bot$ inputs.

Lemma 4.5 (Statement Noninterference)**.** *Let $dom(\Sigma) = dom(\Sigma') = dom(\lambda) = dom(\lambda') = dom(\Psi) = dom(\phi)$, with* (I) *$\lambda(i) = \lambda'(i)$ for all i such that $\phi(i) = \bot$. Suppose further that* (II) *$\Sigma(a+n) = \Sigma'(a+n)$ for all $a+n$ such that $\Psi(a+n) = \bot$. If*

1. $\Psi, \phi \vdash s$ valid,
2. $\Sigma, \lambda \vdash s \Downarrow_s (\Sigma'', \tau, \omega)$, *and*
3. $\Sigma', \lambda' \vdash s \Downarrow_s (\Sigma''', \tau', \omega')$,

then $\tau \equiv \tau'$ and $\omega \doteq \omega'$.

Proof. By induction on the structure of the derivation of $\Sigma, \lambda \vdash s \Downarrow_s (\Sigma', \tau, \omega)$. The case for skip is trivial.

$\boxed{\ell := e}$ Suppose the evaluation under Σ and λ ends with

$$\frac{\Sigma, \lambda \vdash \ell \Downarrow_\ell (a_1 + n_1,\, \tau_1,\, \omega_1) \qquad \Sigma, \lambda \vdash e \Downarrow_e (n_1'\, \tau_1',\, \omega_1')}{\Sigma, \lambda \vdash \ell := e \Downarrow_s (\Sigma[a_1 + n_1/n_1'],\, \tau_1 \triangleright \tau_1' \triangleright \alpha(a_1 + n_1),\, \omega_1 \triangleright \omega_1')}$$

and the evaluation under Σ' and λ' ends with

$$\frac{\Sigma', \lambda' \vdash \ell \Downarrow_\ell (a_2 + n_2,\, \tau_2,\, \omega_2) \qquad \Sigma', \lambda' \vdash e \Downarrow_e (n_2'\, \tau_2',\, \omega_2')}{\Sigma', \lambda' \vdash \ell := e \Downarrow_s (\Sigma'[a_2 + n_2/n_2'],\, \tau_2 \triangleright \tau_2' \triangleright \alpha(a_2 + n_2),\, \omega_2 \triangleright \omega_2')}$$

and typing ends with an application of Assign-T:

$$\frac{\begin{array}{c}\Psi, \phi \vdash \ell : \bot \\ \Psi, \phi \vdash e : \gamma \\ \Psi(\ell) = \gamma' \\ \gamma \leq \gamma'\end{array}}{\Psi, \phi \vdash \ell := e \ \text{valid}}$$

First, note that since $\Psi, \phi \vdash \ell : \bot$, $a_1 + n_1 = a_2 + n_2$ by Lemma 4.2 and hypotheses (I) and (II). By Lemma 4.3, $\tau_1 \equiv \tau_2$ and $\omega_1 \doteq \omega_2$, and similarly $\tau_1' \equiv \tau_2'$ and $\omega_1' \doteq \omega_2'$. Hence $\tau_1 \triangleright \tau_1' \triangleright \alpha(a_1 + n_1) \equiv \tau_\ell' \triangleright \tau_e' \triangleright \alpha(a_2 + n_2)$ by Mem-Addr and two applications of Mem-Concat, and $\omega_1 \triangleright \omega_1' \doteq \omega_2 \triangleright \omega_2'$ by Out-Concat.

$\boxed{s_1; s_2}$ Suppose the evaluation under Σ and λ ends with

$$\frac{\Sigma, \lambda \vdash s_1 \Downarrow_s (\Sigma_1,\, \tau_1,\, \omega_1) \qquad \Sigma_1, \lambda \vdash s_2 \Downarrow_s (\Sigma'',\, \tau_1',\, \omega_1')}{\Sigma, \lambda \vdash s_1; s_2 \Downarrow_s (\Sigma'',\, \tau_1 \triangleright \tau_1',\, \omega_1 \triangleright \omega_1')}$$

and the evaluation under Σ' and λ' ends with

$$\frac{\Sigma', \lambda \vdash s_1 \Downarrow_s (\Sigma_2,\, \tau_2,\, \omega_2) \qquad \Sigma_2, \lambda \vdash s_2 \Downarrow_s (\Sigma''',\, \tau_2',\, \omega_2')}{\Sigma', \lambda \vdash s_1; s_2 \Downarrow_s (\Sigma''',\, \tau_2 \triangleright \tau_2',\, \omega_2 \triangleright \omega_2')}$$

and typing ends with an application of SEQUENCE-T:

$$\frac{\begin{array}{c}\Psi, \phi \vdash s_1 \ \ \mathsf{valid}\\ \Psi, \phi \vdash s_2 \ \ \mathsf{valid}\end{array}}{\Psi, \phi \vdash s_1; s_2 \ \ \mathsf{valid}}$$

By induction on the evaluation of s_1 under Σ, λ and Σ', λ', $\tau_1 \equiv \tau_2$ and $\omega_1 \doteq \omega_2$. By Lemma 4.4 with hypotheses (I) and (II), $\Sigma_1(a + n) = \Sigma_2(a + n)$ for all a such that $\Psi(a + n) = \bot$. Hence by another application of induction to s_2, $\tau_1' \equiv \tau_2'$ and $\omega_1' \doteq \omega_2'$. Thus $\tau_1 \triangleright \tau_1' \equiv \tau_2 \triangleright \tau_2'$ by MEM-CONCAT and $\omega_1 \triangleright \omega_1' \doteq \omega_2 \triangleright \omega_2'$ by OUT-CONCAT.

$\boxed{\mathsf{iterate}\ e\ s}$ Suppose that the typing ends with an application of the rule ITERATE-T:

$$\frac{\begin{array}{c}\Psi, \phi \vdash e : \bot\\ \Psi, \phi \vdash s \ \ \mathsf{valid}\end{array}}{\Psi, \phi \vdash \mathsf{iterate}\ e\ s \ \ \mathsf{valid}}$$

Since $\Psi, \phi \vdash e : \bot$, evaluation of e under Σ and λ must produce the same value as evaluation under Σ' and λ'. That is, $\Sigma, \lambda \vdash e \Downarrow_e (n_1,\, \tau_1,\, \omega_1)$ and $\Sigma', \lambda' \vdash e \Downarrow_e (n_2,\, \tau_2,\, \omega_2)$ implies that $n_1 = n_2$ by Lemma 4.2 with hypotheses (I) and (II). Moreover, by Lemma 4.3 $\tau_1 \equiv \tau_2$ and $\omega_1 \doteq \omega_2$. There are two cases to consider:

1. $n = 0$. Then the evaluation under Σ and λ ends with an application of rule ITERATE (1):

$$\frac{\Sigma, \lambda \vdash e \Downarrow_e (n_1,\, \tau_1,\, \omega_1) \qquad n_1 = 0}{\Sigma, \lambda \vdash \mathsf{iterate}\ e\ s \Downarrow_s (\Sigma,\, \tau_1,\, \omega_1)}$$

   and likewise for Σ' and λ':

$$\frac{\Sigma', \lambda' \vdash e \Downarrow_e (n_2,\, \tau_2,\, \omega_2) \qquad n_2 = 0}{\Sigma', \lambda' \vdash \mathsf{iterate}\ e\ s \Downarrow_s (\Sigma',\, \tau_2,\, \omega_2)}$$

   It has already been established that $\tau_1 \equiv \tau_2$ and $\omega_1 \doteq \omega_2$.

2. $n \neq 0$: Then the evaluation under Σ and λ ends with an application of rule ITERATE (2):

$$\frac{\begin{array}{c}\Sigma, \lambda \vdash e \Downarrow_e (n_1,\, \tau_1,\, \omega_1) \qquad n_1 \neq 0\\ \Sigma, \lambda \vdash s \Downarrow_s (\Sigma_1,\, \tau_1',\, \omega_1')\\ \Sigma_1, \lambda \vdash \mathsf{iterate}\ e\ s \Downarrow_s (\Sigma'',\, \tau_1'',\, \omega_1'')\end{array}}{\Sigma, \lambda \vdash \mathsf{iterate}\ e\ s \Downarrow_s (\Sigma'',\, \tau_1 \triangleright \alpha(s) \triangleright \tau_1' \triangleright \tau_1'',\, \omega_1 \triangleright \omega_1' \triangleright \omega_1'')}$$

   and the evaluation under Σ' and λ' also ends with an application of ITERATE (2):

$$\frac{\begin{array}{c}\Sigma', \lambda' \vdash e \Downarrow_e (n_2,\, \tau_2,\, \omega_2) \qquad n_2 \neq 0 \\ \Sigma', \lambda' \vdash s \Downarrow_s (\Sigma_2,\, \tau_2',\, \omega_2') \\ \Sigma_2, \lambda' \vdash \textsf{iterate}\ e\ s \Downarrow_s (\Sigma''',\, \tau_2'',\, \omega_2'')\end{array}}{\Sigma', \lambda' \vdash \textsf{iterate}\ e\ s \Downarrow_s (\Sigma''',\, \tau_2 \triangleright \alpha(s) \triangleright \tau_2' \triangleright \tau_2'',\, \omega_2 \triangleright \omega_2' \triangleright \omega_2'')}$$

By Lemma 4.3 with hypotheses (I) and (II), $\tau_1 \equiv \tau_2$ and $\omega_1 \doteq \omega_2$. Then by induction on the evaluation of s, $\tau_1' \equiv \tau_2'$ and $\omega_1' \doteq \omega_2'$. By Lemma 4.4 with hypotheses (I) and (II), $\Sigma_1(a+n) = \Sigma_2(a+n)$ for all a such that $\Psi(a+n) = \bot$. Hence by induction on the evaluation of $\textsf{iterate}\ e\ s$ over Σ_1, λ and Σ_2, λ', it holds that $\tau_1'' \equiv \tau_2''$ and $\omega_1'' \doteq \omega_2''$. Finally, $\tau_1 \triangleright \alpha(s) \triangleright \tau_1' \triangleright \tau_1'' \equiv \tau_2 \triangleright \alpha(s) \triangleright \tau_2' \triangleright \tau_2''$ by repeated application of MEM-CONCAT, and likewise $\omega_1 \triangleright \omega_1' \triangleright \omega_1'' \doteq \omega_2 \triangleright \omega_2' \triangleright \omega_2''$ with OUT-CONCAT.

$\boxed{\textsf{select}\ e\ \{s_1, \ldots, s_m\}}$ Suppose that the evaluation under Σ and λ ends with

$$\frac{\begin{array}{c}\Sigma, \lambda \vdash e \Downarrow_e (n_1,\, \tau_1,\, \omega_1) \\ n_1 \in \{1, \ldots, m\} \\ \Sigma, \lambda \vdash s_{n_1} \Downarrow_s (\Sigma'',\, \tau_1',\, \omega_1')\end{array}}{\Sigma, \lambda \vdash \textsf{select}\ e\ \{s_1, \ldots, s_m\} \Downarrow_s (\Sigma'',\, \tau_1 \triangleright \alpha(s_{n_1}) \triangleright \tau_1',\, \omega_1 \triangleright \omega_1')}$$

and the evaluation under Σ' and λ' ends with

$$\frac{\begin{array}{c}\Sigma', \lambda' \vdash e \Downarrow_e (n_2,\, \tau_2,\, \omega_2) \\ n_2 \in \{1, \ldots, m\} \\ \Sigma', \lambda' \vdash s_{n_2} \Downarrow_s (\Sigma''',\, \tau_2',\, \omega_2')\end{array}}{\Sigma', \lambda' \vdash \textsf{select}\ e\ \{s_1, \ldots, s_m\} \Downarrow_s (\Sigma''',\, \tau_2 \triangleright \alpha(s_{n_2}) \triangleright \tau_2',\, \omega_2 \triangleright \omega_2')}$$

and the typing ends with an application of the rule SELECT-T:

$$\frac{\begin{array}{c}\Psi, \phi \vdash e : \bot \\ \Psi, \phi \vdash s_1\ \textsf{valid} \quad \cdots \quad \Psi, \phi \vdash s_m\ \textsf{valid}\end{array}}{\Psi, \phi \vdash \textsf{select}\ e\ \{s_1, \ldots, s_m\}\ \textsf{valid}}$$

By Lemma 4.3 with hypotheses (I) and (II) applied to the evaluation of e, $\tau_1 \equiv \tau_2$ and $\omega_1 \doteq \omega_2$. By Lemma 4.2 with hypotheses (I) and (II) applied to the evaluation of e, $n_1 = n_2$. Hence by induction applied to the evaluation of s_{n_1} and s_{n_2} over Σ, λ and Σ', λ', respectively, $\tau_1' \equiv \tau_2'$ and $\omega_1' \doteq \omega_2'$. Thus $\tau_1 \triangleright \alpha(s_{n_1}) \triangleright \tau_1' \equiv \tau_2 \triangleright \alpha(s_{n_2}) \triangleright \tau_2'$ by MEM-STMT and MEM-CONCAT. Similarly, $\omega_1 \triangleright \omega_1' \doteq \omega_2 \triangleright \omega_2'$ by rule OUT-CONCAT. □

The last lemma establishes that non-$\bot$ inputs do not affect whether or not a program terminates.

Lemma 4.6 (Termination). *Let $dom(\Sigma) = dom(\Sigma') = dom(\lambda) = dom(\lambda') = dom(\Psi) = dom(\phi)$, with $\lambda(i) = \lambda'(i)$ for all i such that $\phi(i) = \bot$, and $\Sigma(a+n) = \Sigma'(a+n)$ for all $a+n$ such that $\Psi(a+n) = \bot$. Given a statement s such that*

1. $\Psi, \phi \vdash s\ \ \mathsf{valid}$*, and*
2. $\Sigma, \lambda \vdash s \Downarrow_s (\Sigma'',\ \tau,\ \omega)$*,*

there exist Σ''', τ', and ω' such that $\Sigma', \lambda' \vdash s \Downarrow_s (\Sigma''',\ \tau',\ \omega')$.

Proof. By induction on the structure of the derivation of $\Sigma, \lambda \vdash s \Downarrow_s (\Sigma',\ \tau,\ \omega)$. The only interesting cases are iterate and select.

$\boxed{\mathsf{iterate}\ e\ s}$ Suppose that the typing ends with an application of the rule Iterate-T:

$$\frac{\begin{array}{c}\Psi, \phi \vdash e : \bot \\ \Psi, \phi \vdash s\ \ \mathsf{valid}\end{array}}{\Psi, \phi \vdash \mathsf{iterate}\ e\ s\ \ \mathsf{valid}}$$

There are two cases to consider:

1. The evaluation under Σ and λ ends with an application of rule Iterate (1):

$$\frac{\Sigma, \lambda \vdash e \Downarrow_e (n,\ \tau,\ \omega) \qquad n = 0}{\Sigma, \lambda \vdash \mathsf{iterate}\ e\ s \Downarrow_s (\Sigma,\ \tau,\ \omega)}$$

   Since $\Psi, \phi \vdash e : \bot$, evaluation of e under Σ' and λ' must produce the same value as evaluation under Σ and λ by Lemma 4.2. That is, $\Sigma', \lambda' \vdash e \Downarrow_e (n,\ \tau',\ \omega')$ where $n = 0$. Hence by rule Iterate (1), $\Sigma', \lambda' \vdash \mathsf{iterate}\ e\ s \Downarrow_s (\Sigma',\ \tau',\ \omega')$.

2. The evaluation under Σ and λ ends with an application of rule Iterate (2):

$$\frac{\begin{array}{c}\Sigma, \lambda \vdash e \Downarrow_e (n,\ \tau_1,\ \omega_1) \qquad n \neq 0 \\ \Sigma, \lambda \vdash s \Downarrow_s (\Sigma_1,\ \tau_1',\ \omega_1') \\ \Sigma_1, \lambda \vdash \mathsf{iterate}\ e\ s \Downarrow_s (\Sigma'',\ \tau_1'',\ \omega_1'')\end{array}}{\Sigma, \lambda \vdash \mathsf{iterate}\ e\ s \Downarrow_s (\Sigma'',\ \tau_1 \rhd \alpha(s) \rhd \tau_1' \rhd \tau_1'',\ \omega_1 \rhd \omega_1' \rhd \omega_1'')}$$

   Since $\Psi, \phi \vdash e : \bot$, evaluation of e under Σ' and λ' must produce the same value as evaluation under Σ and λ by Lemma 4.2. That is, $\Sigma', \lambda' \vdash e \Downarrow_e (n,\ \tau_2,\ \omega_2)$ for some τ_2 and ω_2. Then by induction, there exist Σ_2, τ_2', and ω_2' such that $\Sigma', \lambda' \vdash s \Downarrow_s (\Sigma_2,\ \tau_2',\ \omega_2')$. By another application of induction, there exist Σ''', τ_2'', and ω_2'' such that $\Sigma_2, \lambda' \vdash$

$\textsf{iterate}\ e\ s \Downarrow_s (\Sigma''', \tau_2'', \omega_2'')$. Thus by rule Iterate (2),

$$\Sigma', \lambda' \vdash \textsf{iterate}\ e\ s \Downarrow_s (\Sigma''', \tau_2 \triangleright \alpha(s) \triangleright \tau_2' \triangleright \tau_2'', \omega_2 \triangleright \omega_2' \triangleright \omega_2'').$$

$\boxed{\textsf{select}\ e\ \{s_1, \ldots, s_m\}}$ Suppose that the evaluation under Σ and λ ends with

$$\frac{\begin{array}{c}\Sigma, \lambda \vdash e \Downarrow_e (n, \tau_1, \omega_1) \\ n \in \{1, \ldots, m\} \\ \Sigma, \lambda \vdash s_n \Downarrow_s (\Sigma'', \tau_1', \omega_1')\end{array}}{\Sigma, \lambda \vdash \textsf{select}\ e\ \{s_1, \ldots, s_m\} \Downarrow_s (\Sigma'', \tau_1 \triangleright \alpha(s_n) \triangleright \tau_1', \omega_1 \triangleright \omega_1')}$$

and the typing ends with an application of the rule Select-T:

$$\frac{\begin{array}{c}\Psi, \phi \vdash e : \bot \\ \Psi, \phi \vdash s_1\ \textsf{valid} \quad \cdots \quad \Psi, \phi \vdash s_m\ \textsf{valid}\end{array}}{\Psi, \phi \vdash \textsf{select}\ e\ \{s_1, \ldots, s_m\}\ \textsf{valid}}$$

Since $\Psi, \phi \vdash e : \bot$, evaluation of e under Σ' and λ' must produce the same value as evaluation under Σ and λ by Lemma 4.2. That is, $\Sigma', \lambda' \vdash e \Downarrow_e (n, \tau_2, \omega_2)$ for some τ_2 and ω_2. Then by induction, there exist Σ''', τ_2', and ω_2' such that $\Sigma', \lambda' \vdash s_n \Downarrow_s (\Sigma''', \tau_2', \omega_2')$. Thus by rule Select,

$$\Sigma', \lambda' \vdash \textsf{select}\ e\ \{s_1, \ldots, s_m\} \Downarrow_s (\Sigma''', \tau_2 \triangleright \alpha(s_{n_2}) \triangleright \tau_2', \omega_2 \triangleright \omega_2'). \qquad \square$$

Theorem 4.7 (Termination-Sensitive Noninterference). *Let* $dom(\Sigma) = dom(\Sigma') = dom(\lambda) = dom(\lambda') = dom(\Psi) = dom(\phi)$, *with* $\lambda(i) = \lambda'(i)$ *for all* i *such that* $\phi(i) = \bot$, *and* $\Sigma(a+n) = \Sigma'(a+n)$ *for all* $a+n$ *such that* $\Psi(a+n) = \bot$. *Given a* Core Covert *program* Π *such that* $\Psi, \phi \vdash \Pi\ \textsf{valid}$,

1. *if* $\Sigma, \lambda \vdash \Pi \Downarrow_\Pi (\tau, \omega)$ *and* $\Sigma', \lambda' \vdash \Pi \Downarrow_\Pi (\tau', \omega')$, *then* $\tau \equiv \tau'$ *and* $\omega \doteq \omega'$ *(non-*$\bot$ *inputs noninterfere with memory traces and outputs), and*
2. $\Sigma, \lambda \vdash \Pi \Downarrow_\Pi (\tau, \omega)$ *if and only if* $\Sigma', \lambda' \vdash \Pi \Downarrow_\Pi (\tau', \omega')$ (Core Covert *is termination-sensitive*).

Proof. By Lemma 4.5 and Lemma 4.6, respectively. $\square$

```
main {
    $a_{res} + 0 := 0$;
    $a_{sz} + 0 := i_1$;
    iterate (read($a_{sz} + 0$)) (
        $a_{sz} + 0 :=$ read($a_{sz} + 0$) $- 1$;
        select (read($a_1$ + read($a_{sz} + 0$)) $\neq$ read($a_2$ + read($a_{sz} + 0$))) {
            $a_{res} + 0 := 1$;  $a_{sz} := 0$,
            skip
        }
    )
}
```

Listing 4.1: A Core Covert implementation of `memcmp()` which does not type check

4.3 Example: Typing `memcmp()`

The standard C/C++ implementation of `memcmp()` leaks information through its control flow (Recall Example 1.1). Listing 4.1 shows the corresponding implementation in Core Covert. Additional parentheses have been added to clarify the syntax structure. Suppose that there are two security classes, $\bot$ and $\top$, such that $\bot \leq \top$ and $\top \not\leq \bot$. The two buffers are located in arrays a_1 and a_2. There are also two variables, i.e., arrays of length one: a_{sz} stores the size of the two buffers (the sizes are assumed equal), and a_{res} records the result. The result is 0 if the buffers are identical, and 1 otherwise. The buffer size is read from a program input i_1, and the buffer contents are assumed to be present in memory when the program begins.

Each loop iteration decrements the value stored in a_{sz}, which also determines the index for the next pair of read operations. On a given iteration, if the two integers read from corresponding indices in the buffers differ, then $a_{res} + 0$ is set to 1, and the index $a_{sz} + 0$ is set to 0, thus terminating the loop immediately. Otherwise, the loop continues to iterate until a difference is found, or the beginning of both buffers is reached.

It is assumed that $\Phi(a_{res}) = \top$, $\Phi(a_{sz}) = \bot$, $\Phi(a_1) = \top$, and $\Phi(a_2) = \top$, and that $\phi(i_1) = \bot$. Does this program type check? Fortunately, it is not necessary to build the entire type derivation tree. Note that $a_1 + \mathsf{read}(a_{sz} + 0)$ and $a_2 + \mathsf{read}(a_{sz} + 0)$ trivially have type $\top$ because $\Phi(a_1) = \Phi(a_2) = \top$. Hence by the Simple Security Theorem (Theorem 4.1), the

```
1  main {
2     $a_{res} + 0 := 0$;
3     $a_{sz} + 0 := i_1$;
4     iterate (read($a_{sz} + 0$)) (
5        $a_{sz} + 0 :=$ read($a_{sz} + 0$) $- 1$;
6        $a_{res} + 0 :=$ (read($a_1 +$ read($a_{sz} + 0$)) $\neq$ read($a_2 +$ read($a_{sz} + 0$))) | read($a_{res} + 0$)
7     )
8  }
```

Listing 4.2: A Core Covert implementation of `memcmp()` which type checks

condition expression for the select statement must have type $\top$. By rule SELECT-T (the only rule which can type select), the program in Listing 4.1 cannot type check.

Is it possible to compose a Core Covert program which is semantically equivalent to Listing 4.1, but which does type check? A valid solution must avoid any situation where a branch condition expression must be assigned type $\top$. One possible solution is given in Listing 4.2. All of the assumptions about the initial context Σ, λ and memory layout remain unchanged. What is different is that the loop does not short-circuit as soon as a difference between the two buffers is detected. Rather, the assignment to $a_{res} + 0$ in Line 6 simply updates the result value to 1 when two corresponding integers differ. The "|" operator is a bitwise OR, as in C and C++. Thus once the value at $a_{res} + 0$ is updated to 1 (i.e., the 0th bit is set), that bit cannot be unset by the bitwise OR. For brevity, the type derivation tree for Listing 4.2 has been omitted. It is easy to verify that noninterference is never violated by applying the Simple Security Theorem to each read() and each assignment, and the condition for the iterate statement.

One noteworthy aspect of these two implementations is that neither emits the result stored in $a_{res} + 0$ to the output trace. Rather, this value just remains in memory when the program terminates. This is because $\Psi(a_{res} + 0) = \top$, and thus the expression output($a_{res} + 0$) would not be typable by rule OUTPUT-T. If it were typable, then Core Covert would not satisfy noninterference. This topic of downgrading or disclosing the result of a computation is addressed in the next chapter.

Chapter 5

Covert C++

Covert C++ is a practical realization of the Core Covert language, introduced in the prior chapter. The Covert C++ security-type system specifies an information-flow policy which roughly enforces two categories of constraints. First, the information flow-policy prevents explicit flows from downgrading secret data into public data. Second, it precludes implicit flows of secret data through program control flow and memory access patterns (e.g., via pointer subscripting). The former constraint obviates storage-channel leaks, while the latter closes termination-channel leaks and defeats side-channel attacks against the memory hierarchy, including page fault and cache-based attacks. The information-flow control which enforces the policy is the type checking algorithm built into any C++17-compliant compiler, including GCC, Clang, and the Microsoft Visual C++ compiler (MSVC).

This chapter describes the design and implementation of the Covert C++ security-type system. In particular, a formal description of the type system is presented, along with an informal proof that Covert C++ enforces noninterference. The chapter closes with a case study involving an SGX enclave program whose security goal is to protect (i.e., not leak) a digital rights management (DRM) key. For the reader who is unfamiliar with C++ (esp. C++11 and newer), Table 5.1 provides a reference of the C++ keywords and metafunctions used throughout the chapter (and also in later chapters).

C++ Keyword	**Description**
`decltype( expression )`	Takes an expression and returns its type. Evaluated at compile time.
`template < parameter list > declaration`	Declares a `struct`, `class`, function or variable to be a template, thus allowing it to be parameterized by one or more types and/or compile-time constants.
`typename name`	Indicates that the following identifier (name) is the name of a type. Commonly used to declare template type parameters (the `class` keyword can also be used to declare template type parameters)
`constexpr declaration`	Declares that a variable or function may be computed at compile time.
`operator op`	Used to indicate the name of an overloaded operator, e.g., `operator+` or `std::string::operator+=`.
`reinterpret_cast < type > ( expression )`	Treats the value of the given expression as though it has the given type by reinterpreting the underlying bit pattern. A compile-time cast, i.e., it does not compile to any CPU instructions.

STL Metafunction	**Description**
`template <class T> std::declval`	Converts a type `T` to a reference type `T &&`. Can only be used in unevaluated environments (e.g., in an argument to `decltype`) to create a pseudo lvalue/rvalue object of type `T`.
`template <class T1, class T2> std::is_convertible`	Evaluates to `true` when the first template type argument is implicitly convertible to the second template type argument.
`template <class T, T v> std::integral_constant`	Represents a compile-time constant `v` of type `T`.
`template <bool b, class T> std::enable_if_t`	Evaluates to the type `T` if `b` evaluates to `true`; otherwise it is undefined. Commonly used with SFINAE to enable conditional compilation.

Table 5.1: C++ keywords and metafunctions

5.1 Type System Design

Covert C++ augments the C++ type system by pairing data types with security labels. A *security label* is the representation of a security class (e.g., public, secret) in a Covert C++ program. These labels may constrain the behavior of sensitive data. There are two security classes: low (public) and high (secret). In C++:

```
enum SLabel { L, H };
```

These labels form a binary priority lattice in which `L` is strictly less than `H`. A data type can be associated with a security label by instantiating the `SE` template container for that data type and security label. For example, `SE<char, L>` is the type of a low character, and `SE<int, H>` for a high integer. The `SE` container does not accept references, nor does it accept types that are cv-qualified at the top level. Hence a `const` lvalue reference to a low `int` must be expressed as `const SE<int, L> &`.

Security typing for pointers is more complicated. Pointers may require multiple security labels: one for the pointer itself, and one additional security label for each pointee. For example, `int**` requires three security labels. The meaning of a security-typed pointer is also more nuanced. A pointer `p` of type `SE<int*, L, H>` can be thought of as a low pointer to a high integer, but this is not always the case. The typing rules introduced later in this section allow one pointer to alias another pointer with lower security labels. Hence `p` could be an alias of a pointer `q` of type `SE<int*, L, L>`. By permitting this kind of *security-upgrade* aliasing, Covert C++ allows functions and algorithms to specify an upper bound on the security classes of their parameters.

To clarify the Covert C++ security-type system, some additional terminology is required. The number of security labels associated with a given type `T` is referred to as `T`'s *type depth*. A type is *primitive* if it belongs to any of the following categories: arithmetic (`int`, `char`, etc.), `enum`, non-function pointer, or pointer to any non-`SE` type, at any level. The type depth of a primitive type is is one, plus the number of pointers in the type. The type depth of an

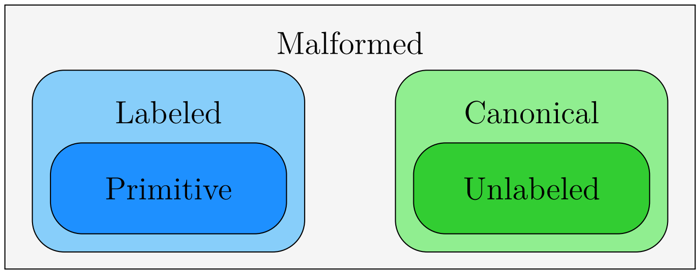

Figure 5.1: Covert C++ type relationships

`SE` type is zero, hence an `SE` type cannot be nested immediately inside of another `SE` type (although a pointer to an `SE` type can be nested within an `SE` type). A *labeled* type is any type with non-zero type depth (i.e., it may be assigned a security label). An *unlabeled* type is a non-`SE` type with zero type depth. An unlabeled type `T` can become a labeled type by specializing the type depth metafunction for `T`, and assigning to it an appropriate number of security labels. A type is *canonical* if it is unlabeled, or it is of the form `SE<T, S1,...,SN>` where `T` is a labeled type with type depth equal to `N`.

All other types are *malformed*. Some examples of malformed types are `SE<char, H>*` and `SE<SE<int, L>*, H>`. Malformed types must be allowed to exist in Covert C++. Consider the case when an array decays to a pointer. The type `int[8]` decays to `int*`. Similarly, `SE<int, H>[8]` will decay to `SE<int, H>*`. There is no way to override or circumvent this behavior in C++, so it must be allowed. Malformed types can also materialize when template parameters of template classes or functions are instantiated with `SE` types. Types which are either malformed or canonical, but not unlabeled, are collectively referred to as *covert* types. That is, a covert type is either a (possibly malformed) `SE` type or a pointer to an `SE` type. Figure 5.1 illustrates these type relationships.

Informally, Covert C++'s information-flow policy enforces the following rules (with several simplifications):

1. Labeled types can be implicitly converted into low covert types, and vice-versa.
    (a) Low covert types can be implicitly converted into high covert types.

   (b) High covert types cannot be implicitly converted into low covert types.

2. Operators which perform a computation over several operands (e.g., `+`, `|`, etc.) produce a result with a security label that is the least upper bound (LUB) of the operands' respective security labels.
3. Taking the address (`&`) of an lvalue of type `SE<T, [...]>` produces a pointer of type `SE<T*, L, [...]>`.
4. High covert-typed pointers cannot be used to access memory (e.g., via pointer indirection or subscripting). A pointer of type `SE<T*, L, [...]>` may be used to access memory; if it is used to read memory, then the result has type `SE<T, [...]>`, or `T` alone if `T` is an unlabeled type.

Although this summary oversimplifies Covert C++ quite a bit, these four basic rules capture the fundamental security features of the language.

More formally, Figures 5.2 and 5.3 show the precise Covert C++ type conversion rules and typing rules, respectively, expressed in terms of the type relationships depicted in Figure 5.1. Types γ are tuples $(T, (S_1, \ldots, S_n))$, and there are two projection functions $\mathbf{proj}_L$ and $\mathbf{proj}_R$ such that $T = \mathbf{proj}_L\langle\gamma\rangle$ and $(S_1, \ldots, S_n) = \mathbf{proj}_R\langle\gamma\rangle$. The left and right projections are referred to as the *inner type* and *security labels* of γ, respectively. A function with a parenthesized argument list is an ordinary function, and a function with an angle-enclosed argument list is a metafunction—computed at compile time—defined over types. Metafunctions in the `std` namespace are part of the C++ STL.

The `decltype` keyword was introduced in C++11 [88]. It takes an expression, and returns the type of that expression if it were to be evaluated. The `std::declval` function is an undefined function intended only for use in unevaluated contexts. `std::declval<T>()` has type `T &&`, thus in an unevaluated context it effectively produces a reference to a pseudo object/value of type `T` [88].

The SLabels (1, 2) rules define the Covert C++ metafunction `is_se_convertible`, which ensures that when a value of type γ is converted into a value of type γ', no security

T	$:=$	A labeled type	e	$:=$	An expression
S	$:=$	A Covert C++ security label	γ	$:=$	A canonical type
			μ	$:=$	A malformed type

$$\textsc{SLabels (1)}\ \frac{\begin{array}{c}(S_1,\ldots,S_n)=\mathbf{proj}_R\langle\gamma\rangle\\(S'_1,\ldots,S'_m)=\mathbf{proj}_R\langle\gamma'\rangle\\n\le m\\\forall i\in\{1,\ldots,n\}.\ S_i\le S'_i\end{array}}{\texttt{is\_se\_convertible}\langle\gamma,\ \gamma'\rangle}$$

$$\textsc{SLabels (2)}\ \frac{\begin{array}{c}(S_1,\ldots,S_n)=\mathbf{proj}_R\langle\gamma\rangle\\(S'_1,\ldots,S'_m)=\mathbf{proj}_R\langle\gamma'\rangle\\n> m\\\forall i\in\{1,\ldots,m\}.\ S_i\le S'_i\\\forall j\in\{m+1,\ldots,n\}.\ S_j=L\end{array}}{\texttt{is\_se\_convertible}\langle\gamma,\ \gamma'\rangle}$$

$$\textsc{SE To SE}\ \frac{\begin{array}{c}T=\mathbf{proj}_L\langle\gamma\rangle\\\texttt{std::is\_convertible}\langle T,\ T'\rangle\end{array}\quad\begin{array}{c}T'=\mathbf{proj}_L\langle\gamma'\rangle\\\texttt{is\_se\_convertible}\langle\gamma,\ \gamma'\rangle\end{array}}{\texttt{std::is\_convertible}\langle\gamma,\ \gamma'\rangle}$$

$$\textsc{To SE}\ \frac{T=\mathbf{proj}_L\langle\gamma\rangle}{\texttt{std::is\_convertible}\langle T,\ \gamma\rangle}\qquad\textsc{To Canonical}\ \frac{\begin{array}{c}\gamma'=\texttt{canonicalize}\langle\mu\rangle\\\texttt{std::is\_convertible}\langle\gamma',\ \gamma\rangle\end{array}}{\texttt{std::is\_convertible}\langle\mu,\ \gamma\rangle}$$

$$\textsc{From SE}\ \frac{\begin{array}{c}T=\mathbf{proj}_L\langle\gamma\rangle\\(S_1,\ldots,S_n)=\mathbf{proj}_R\langle\gamma\rangle\\\forall i\in\{1,\ldots,n\}.\ S_i=L\end{array}}{\texttt{std::is\_convertible}\langle\gamma,\ T\rangle}\qquad\textsc{From Canonical}\ \frac{\begin{array}{c}\gamma'=\texttt{canonicalize}\langle\mu\rangle\\\texttt{std::is\_convertible}\langle\gamma,\ \gamma'\rangle\end{array}}{\texttt{std::is\_convertible}\langle\gamma,\ \mu\rangle}$$

Figure 5.2: Covert C++ type conversion rules.

label is downgraded from H to L. Security labels may be upgraded from L to H. Moreover, SLabels (2) checks that no H security label is lost when reducing the number of security labels during a type cast. For example, it would prevent `SE<int*, L, H>` from being cast to `SE<uintptr_t, L>`.

The `std::is_convertible` metafunction tests whether there exists an implicit conversion sequence (possibly including a user-defined conversion operator) from its first argument to its second argument. The SE To SE rule allows γ to be implicitly converted (hence: converted) to γ' if their inner types are convertible and their security labels are convertible. The To SE rule allows any labeled type to be converted to a canonical type with any sequence of security labels of correct length. The From SE rule allows any canonical type with only L security labels to be converted to its inner type. These rules collectively define the information-flow policy for explicit flows that arise when data is copied from one (possibly temporary) object to another.

The To Canonical and From Canonical rules allow any malformed type to be converted to a canonical type, and vice-versa. The `canonicalize` metafunction is quite complex, and thus difficult to capture with inference rules. In essence, `canonicalize` performs two distinct operations. First, it pulls outlying pointers into the `SE` container, assigning a L security label to each such pointer. Thus `SE<int, H>*` would become `SE<int*, L, H>`. Second, `canonicalize` flattens nested `SE` containers, for example transforming `SE<SE<int, H>*, L>` into `SE<int*, L, H>`.

Label Cast defines the typing of the `se_label_cast()` function. This rule is similar to the SE To SE rule, except that Label Cast does allow security labels to be downgraded. Also note that SE To SE defined an implicit conversion, whereas `se_label_cast()` must be called explicitly by the user, and all of the new security labels must be specified in the template arguments. The intent is to force the developer to consciously declare which security labels are being downgraded. Although it may seem counter-productive to have a rule which downgrades security labels, Section 5.6 discusses some scenarios where `se_label_cast()` can be necessary and helpful, when used with caution.

The C++ language defines four named cast operators: `static_cast`, `reinterpret_cast`, `const_cast`, and `dynamic_cast`. C++ does not allow these functions to be overloaded, therefore Covert C++ defines its own `se_static_cast`, and likewise for the other named casts. The Named Cast inference rule for each cast is identical, and it is macro-defined for each of the four C++ named casts. The argument to `decltype` is the named cast from a pseudo value of type T to T'. If this cast is valid in C++, then `decltype` will return the type of the unevaluated expression, T', and the rule will resolve (assuming the security labels are also convertible). If the `[name]_cast` is not valid from T to T', then `se_[name]_cast` is not valid from γ to γ'.

The typing rules for the arithmetic and logical operators are more straightforward. The LUB over the lattice of security levels is denoted by $\sqcup$, such that

$$S_1 \sqcup S_2 = L \iff S_1 = L \wedge S_2 = L$$

$$
\begin{array}{lcl}
op_{UA} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{~} \mid \texttt{++} \mid \texttt{-} \\
op_{AA} & ::= & \texttt{+=} \mid \texttt{-=} \mid \texttt{*=} \mid \texttt{/=} \mid \texttt{\%=} \mid \texttt{^=} \mid \texttt{\&=} \mid \texttt{|=} \mid \texttt{<<=} \mid \texttt{>>=} \\
op_{BA} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{^} \mid \texttt{\&} \mid \texttt{|} \mid \texttt{<<} \mid \texttt{>>} \mid \texttt{<} \mid \texttt{>} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=} \\
op_{PA} & ::= & \texttt{+} \mid \texttt{-}
\end{array}
$$

$$
\textsc{Label Cast}\ \frac{\begin{array}{c} e : \gamma \qquad T = \mathbf{proj}_L\langle\gamma\rangle \\ T' = \mathbf{proj}_L\langle\gamma'\rangle \\ \texttt{std::is\_convertible}\langle T,\, T'\rangle \end{array}}{\texttt{se\_label\_cast}\langle\gamma'\rangle(e) : \gamma'}
\qquad
\textsc{Named Cast}\ \frac{\begin{array}{c} e : \gamma \quad \texttt{is\_se\_convertible}\langle\gamma,\, \gamma'\rangle \\ T = \mathbf{proj}_L\langle\gamma\rangle \qquad T' = \mathbf{proj}_L\langle\gamma'\rangle \\ T' = \texttt{decltype([name]\_cast}\langle T'\rangle\texttt{(} \\ \qquad \texttt{std::declval}\langle T\rangle\texttt{))} \end{array}}{\texttt{se\_[name]\_cast}\langle\gamma'\rangle(e) : \gamma'}
$$

$$
\textsc{Unary Logic Op}\ \frac{\begin{array}{c} e : \gamma \quad T = \mathbf{proj}_L\langle\gamma\rangle \quad (S_1, \ldots, S_n) = \mathbf{proj}_R\langle\gamma\rangle \\ \texttt{bool} = \texttt{decltype(!std::declval}\langle T\rangle\texttt{)} \end{array}}{\texttt{!}e : (\texttt{bool},\, (S_1))}
$$

$$
\textsc{Unary Arith Op}\ \frac{\begin{array}{c} e : \gamma \quad T = \mathbf{proj}_L\langle\gamma\rangle \quad (S_1, \ldots, S_n) = \mathbf{proj}_R\langle\gamma\rangle \\ T' = \texttt{decltype(}op_{UA}\ \texttt{std::declval}\langle T\rangle\texttt{)} \end{array}}{op_{UA}\ e : (T',\, (S_1, \ldots, S_n))}
$$

$$
\textsc{Arith Assign}\ \frac{\begin{array}{c} e_1 : \gamma_1 \qquad T_1 = \mathbf{proj}_L\langle\gamma_1\rangle \qquad (S^1_1, \ldots, S^1_n) = \mathbf{proj}_R\langle\gamma_1\rangle \\ e_2 : \gamma_2 \qquad T_2 = \mathbf{proj}_L\langle\gamma_2\rangle \qquad (S^2_1) = \mathbf{proj}_R\langle\gamma_2\rangle \\ T'_1 = \texttt{decltype(std::declval}\langle T_1\rangle\ op_{AA}\ \texttt{std::declval}\langle T_2\rangle\texttt{)} \quad S^1_1 \geq S^2_1 \end{array}}{e_1\ op_{AA}\ e_2 : (T'_1,\, (S^1_1, \ldots, S^1_n))}
$$

$$
\textsc{Pointer Arith (1)}\ \frac{\begin{array}{c} e_1 : \gamma_1 \quad T_1 = \mathbf{proj}_L\langle\gamma_1\rangle \quad (S^1_1, S^1_2, \ldots, S^1_n) = \mathbf{proj}_R\langle\gamma_1\rangle \\ e_2 : \gamma_2 \quad T_2 = \mathbf{proj}_L\langle\gamma_2\rangle \quad (S^2_1) = \mathbf{proj}_R\langle\gamma_2\rangle \quad S_1 = S^1_1 \sqcup S^2_1 \\ T = \texttt{decltype(std::declval}\langle T_1\rangle\ op_{PA}\ \texttt{std::declval}\langle T_2\rangle\texttt{)} \end{array}}{e_1\ op_{PA}\ e_2 : (T,\, (S_1, S^1_2 \ldots, S^1_n))}
$$

$$
\textsc{Pointer Arith (2)}\ \frac{\begin{array}{c} e_1 : \gamma_1 \quad T_1 = \mathbf{proj}_L\langle\gamma_1\rangle \quad (S^1_1) = \mathbf{proj}_R\langle\gamma_1\rangle \quad S_1 = S^1_1 \sqcup S^2_1 \\ e_2 : \gamma_2 \quad T_2 = \mathbf{proj}_L\langle\gamma_2\rangle \quad (S^2_1, S^2_2, \ldots, S^2_m) = \mathbf{proj}_R\langle\gamma_2\rangle \\ T = \texttt{decltype(std::declval}\langle T_1\rangle\ op_{PA}\ \texttt{std::declval}\langle T_2\rangle\texttt{)} \end{array}}{e_1\ op_{PA}\ e_2 : (T,\, (S_1, S^2_2 \ldots, S^2_n))}
$$

$$
\textsc{Binary Arith}\ \frac{\begin{array}{c} e_1 : \gamma_1 \qquad T_1 = \mathbf{proj}_L\langle\gamma_1\rangle \qquad (S^1) = \mathbf{proj}_R\langle\gamma_1\rangle \\ e_2 : \gamma_2 \qquad T_2 = \mathbf{proj}_L\langle\gamma_2\rangle \qquad (S^2) = \mathbf{proj}_R\langle\gamma_2\rangle \\ T = \texttt{decltype(std::declval}\langle T_1\rangle\ op_{BA}\ \texttt{std::declval}\langle T_2\rangle\texttt{)} \quad S = S^1 \sqcup S^2 \end{array}}{e_1\ op_{BA}\ e_2 : (T,\, (S))}
$$

$$
\textsc{Indirection}\ \frac{\begin{array}{c} e : \gamma \qquad T_{ptr} = \mathbf{proj}_L\langle\gamma\rangle \\ (S_1, S_2, \ldots, S_n) = \mathbf{proj}_R\langle\gamma\rangle \\ S_1 \neq H \\ T = \texttt{decltype(} \\ \qquad \texttt{*std::declval}\langle T_{ptr}\rangle\texttt{)} \end{array}}{\texttt{*}e : (T,\, (S_2, \ldots, S_n))}
\qquad
\textsc{Subscript}\ \frac{\begin{array}{c} e : \gamma \qquad e' : \gamma' \\ T_{ptr} = \mathbf{proj}_L\langle\gamma\rangle \qquad T_{idx} = \mathbf{proj}_L\langle\gamma'\rangle \\ (S_1, S_2, \ldots, S_n) = \mathbf{proj}_R\langle\gamma\rangle \quad S_1 \neq H \\ (S^i) = \mathbf{proj}_R\langle\gamma\rangle \qquad S^i \neq H \\ T = \texttt{decltype(std::declval}\langle T_{ptr}\rangle \\ \qquad \texttt{[std::declval}\langle T_{idx}\rangle\texttt{])} \end{array}}{e\texttt{[}e'\texttt{]} : (T,\, (S_2, \ldots, S_n))}
$$

$$
\textsc{Arrow}\ \frac{\begin{array}{c} e : \gamma \qquad m : \gamma' \\ (S_1, S_2, \ldots, S_n) = \mathbf{proj}_R\langle\gamma\rangle \\ S_1 \neq H \end{array}}{e\texttt{->}m : \gamma'}
\qquad
\textsc{Address-Of}\ \frac{\begin{array}{c} e : \gamma \qquad T = \mathbf{proj}_L\langle\gamma\rangle \\ (S_1, \ldots, S_n) = \mathbf{proj}_R\langle\gamma\rangle \end{array}}{\texttt{\&}e : (T*,\, (L, S_1, \ldots, S_n))}
$$

Figure 5.3: Covert C++ typing rules.

Hence the Pointer Arith (1, 2) rules and the Binary Arith rule dictate that operations which combine security-typed data propagate H security labels. Note that pointer arithmetic operations preserve the security label(s) of the pointee type.

Unary Arith Op uses `decltype` and `std::declval` to simulate the given unary arithmetic operation on a pseudo member of the inner type. If that evaluation succeeds, then the result obtains the inner type of the Covert C++ expression, and the security labels remain unchanged.

The use of `decltype` allows Covert C++ to be more lazy with its inference rule definitions. For instance, if T is a pointer type, and op_{UA} is the unary minus (negation) operator, then the expression argument to `decltype` is not well-formed in C++. This is because integer negation is invalid when applied to an expression of pointer type. When `decltype` fails to resolve the type of the given expression—perhaps because the expression is not well-typed—the type substitution fails, and thus the typing rule cannot be applied. Hence without `decltype`, each inference rule would need to explicitly list all of the acceptable types T, or T_1 and T_2 for the binary operators. Not only would this be extremely inconvenient, it would also mean that the Covert C++ typing rules would need to be updated as new C++ standard revisions adjust the C++ typing rules.

The Indirection, Subscript, and Arrow rules define the respective typing rules for operations which access memory by operating on pointers. All of these rules mandate that pointers with H as their front-most security label cannot be used to access memory. Moreover, a pointer subscript value cannot be H. The Address-Of rule simply prepends an L label to the result of the `&` operation, which returns the address in memory where the given variable resides. The variable's address is treated as public because it does not make sense to treat the location of a variable—whether on the stack, heap, or in static memory—as a program secret.

These inference rules entirely characterize the Covert C++ type system. What this dissertation does not present is an accompanying formal model of Covert C++ semantics.

The reason is that the C++ language is extraordinarily complex—the C++17 standard language specification [90] comprises 1,622 pages. Although other works [122, 126] have successfully modeled various subsets of the C++ language, the amount of effort required to build a truly accurate model of Covert C++ sufficient to capture the noninterference property would likely exceed that of any prior work. This was the rationale behind the simplified formulation of Core Covert presented in Chapter 4. Moreover, a formal proof that Covert C++ enforces noninterference would not even guarantee that this property would be preserved during compilation, a topic that is addressed later in Chapter 6. Nonetheless, several illustrative type inference rules for control-flow statements and expressions can be derived from the type inference rules in Figure 5.3.

C++ features a small set of control flow statements, collectively referred to as *selection* and *iteration* statements. Section 9/4 of the language standard specifies constraints on the branch/termination conditions for these statements:

> "The value of a *condition* that is an initialized declaration in a statement other than a `switch` statement is the value of the declared variable contextually converted to `bool` . . . . If that conversion is ill-formed, the program is ill-formed. The value of a *condition* that is an initialized declaration in a `switch` statement is the value of the declared variable if it has integral or enumeration type, or of that variable implicitly converted to integral or enumeration type otherwise. The value of a *condition* that is an expression is the value of the expression, contextually converted to `bool` for statements other than `switch`; if that conversion is ill-formed, the program is ill-formed." [90]

In brief, a selection/iteration statement is only well-formed when its branch/termination condition is contextually convertible to `bool`, or an integral or enumeration type, depending on the selection/iteration statement. By the FROM SE rule, an H value cannot be contextually converted to a value of one of these types. Thus an H-labeled expression cannot be the branch condition for a well-formed selection statement. This well-formedness criteria for selection statements can be expressed as a pair of type inference rules:

$$\textsc{Select-T'}\ \frac{e : \gamma \qquad (S_1, \ldots, S_n) = \mathbf{proj}_R\langle\gamma\rangle \qquad S_1 \neq H}{\texttt{select}\ C\ \{s_1, \ldots\ s_m\}}$$

$$\textsc{Iterate-T'}\ \frac{e : \gamma \qquad (S_1, \ldots, S_n) = \mathbf{proj}_R\langle\gamma\rangle \qquad S_1 \neq H}{\texttt{iterate}\ e\ \{s\}}$$

where `select` is one of `switch` or `if`/`else if`, and the s_i are the conditional branches. The `iterate` statement is one of `while`, `do while`, or `for`. Notice the similarity between these rules and their Core Covert counterparts, Select-T and Iterate-T respectively, given in Figure 4.3.

There are also three operators in C++ which have branching behaviors. Just as with the select and iterate statements, the `&&`, `||`, and `?:` operators require a contextual conversion to `bool` for their first argument, and the remaining operand(s) may or may not be evaluated, depending on the value of the first operand. Semantic inference rules can also be derived for these operators:

$$\textsc{Ternary-T'}\ \frac{\begin{array}{cc} e_1 : \gamma & T = \mathbf{proj}_L\langle\gamma\rangle \\ (S_1, \ldots, S_n) = \mathbf{proj}_R\langle\gamma\rangle & S_1 \neq H \\ \multicolumn{2}{c}{\gamma' = \texttt{decltype(std::declval}\langle T\rangle\ \texttt{?}\ e_2\ \texttt{:}\ e_3\texttt{)}} \end{array}}{(e_1\texttt{?}\ e_2\ \texttt{:}\ e_3) : \gamma'}$$

$$\textsc{Binary Logic-T}\ \frac{\begin{array}{ccc} e_1 : \gamma_1 & (S^1_1, \ldots, S^1_n) = \mathbf{proj}_R\langle\gamma_1\rangle & S^1_1 \neq H \\ e_2 : \gamma_2 & (S^2_1, \ldots, S^2_n) = \mathbf{proj}_R\langle\gamma_2\rangle & S^2_1 \neq H \\ \multicolumn{3}{c}{T_1 = \mathbf{proj}_L\langle\gamma_1\rangle \qquad T_2 = \mathbf{proj}_L\langle\gamma_2\rangle} \\ \multicolumn{3}{c}{\texttt{bool} = \texttt{decltype(std::declval}\langle T_1\rangle\ op_{BL}} \\ \multicolumn{3}{r}{\texttt{std::declval}\langle T_2\rangle\texttt{)}} \end{array}}{e_1\ op_{BL}\ e_2 : \texttt{bool}}$$

where op_{BL} is one of `&&` or `||`. Note once again the similarity between Ternary-T' and rule Ternary-T from the previous chapter. The Binary Logic-T rule is more restrictive than it needs to be. Since a control flow decision is only made for e_1, it should follow that the constraint $S^2_1 \neq H$ is unnecessary. However, the `&&` and `||` operators are not overloaded by Covert C++ because it is impossible to overload them while maintaining their short-circuiting behavior[1]. The C++ standard requires that both operands must be convertible to `bool`, hence both S_1^1 and S_1^2 must not be H.

5.1.1 Relationship to Core Covert

Chapter 4 introduced the Core Covert language, which had the objective of being detailed enough to capture the key features of Covert C++, while remaining simple enough to be formally verifiable. Before presenting the informal proof that Covert C++ enforces noninterference, it would be helpful to first examine the correspondence between Core Covert and Covert C++.

The correspondence between type inference rules for select, iterate, and for ternary expressions was noted above. Each of these constructs must make a branching decision, and each of their type inference rules requires that the branch condition must be public, i.e., $\bot$ in Core Covert or L in Covert C++.

Pointers in Covert C++ correspond to locations in Core Covert. Just as an `SE` pointer to a primitive type must have two security labels—one for the pointee, and one for the pointer itself—each location must have two associated security classes in order to be used in well-typed expressions and statements. For example, the Covert C++ type `SE<int*, L, H>` corresponds to a location ℓ such that $\Psi, \phi \vdash \ell : \bot$ and $\Psi(\ell) = \top$, given the typing context Ψ, ϕ. There is, however, no notion of a pointer to a pointer, etc. in Core Covert.

Pointer arithmetic corresponds to location arithmetic, i.e., the ADDRESS and ADDRESS-T rules. Pointer indirection and subscripting is covered by the read(ℓ) expression and assignment $\ell := e$ statements, together with location arithmetic. The typing rules are also analogous. The READ-T and ASSIGN-T rules in Core Covert mandate that the location being read from or written to must be $\bot$. The INDIRECTION and SUBSCRIPT rules in Covert C++ enforce the same constraint on pointers. There is no analog to the `->` operator in Core Covert, because there is no notion of a `struct` or `class` in Core Covert.

[1]The technical reason for this is that functions in C++ use call-by-value semantics, which requires that all arguments be evaluated before the function call is made.

5.2 Covert C++ Enforces Noninterference

With some additional requirements pertaining to safety, it is possible to formulate an argument that Covert C++ prevents explicit flows between objects that would violate the information-flow policy, e.g., by downgrading the security class of some information.

It is not expected that the developer will always adhere to the following requirements. Part of the allure of C++ is that it allows experienced programmers to wander outside the confines of the type system—at their own risk. Covert C++ keeps with this tradition.

Requirement 1 (Type/Memory Safety)**.** Type and memory safety cannot be violated, e.g., by reading/writing beyond the bounds of a buffer, using a pointer after it has been freed, etc.

This first requirement is purposefully vague. Type and memory safety in a weakly-typed language such as C++ is complicated and nuanced. A formal and complete definition of the necessary requirements to ensure safety in C++ has been given by DeLozier et al. [63]. Any violation of Requirement 1 can easily subvert the protections facilitated by Covert C++. For example:

```
1 SE<int, H> secret = 42, arr[8];
2 SE<int, L> val;
3 arr[8] = secret;
```

If the procedure stack is organized such that `val` is located immediately above `arr`, then in Line 3 the value of the high variable `secret` will be written to the low variable `val`. This is an example of an explicit flow which circumvents the security typing provided by Covert C++. The C++ language itself cannot prevent or detect this behavior. Another example:

```
1 SE<int, H> arr[8];
2 void *ptr = arr;
3 SE<int, L> val = *static_cast<int *>(ptr);
```

The cast in Line 2 is valid in C++. First, `arr` is allowed to decay to a pointer of type `SE<int, H>*`. C++ allows any prvalue pointer to be implicitly cast to an identically cv-qualified `void` pointer, hence this completes the cast [90].

Requirement 2 (Safe Pointer-to-Pointer Casts)**.** Pointer-to-pointer casts (including both explicit and implicit casts) with a source expression of covert type must only cast between compatible pointee types, or cast up or down a class hierarchy.

This requirement is partially covered by Requirement 1, but only for non-covert types. Covert named casts must also follow this rule. The C99 standard states that two types are *compatible* if they are identical modulo const/volatile/restricted (cvr) qualifiers and type aliases, or if they are pointers to compatible types, or if they are array types of the same size and with compatible elements [87]. Hence casts between compatible types can generally be considered safe. Note that Requirement 1 additionally prevents any kind of cast from a covert pointer type to a (possibly) covert `void` pointer type, because the `void` type is only compatible with itself.

Requirement 3 (Restricted Built-in Casts)**.** C++ built-in named casts must not have a source expression of covert type.

This requirement prevents labels from being cast away by the built-in C++ named casts. For instance, `reinterpret_cast` can be used to cast an lvalue of any type to a reference to any other (non-`void`) object type [90]. Thus

```
SE<int, H> secret = 42;
int &non_secret = reinterpret_cast<int &>(secret);
```

is valid C++ code. Therefore any covert type could be cast directly into a reference to any simple type. Given Requirement 1, it may be possible to omit Requirement 3. For instance, the reinterpret cast in the example above is obviated by type safety. Yet given the informal nature of this discussion, the complexity of the C++ named casts, and the fact that Covert C++ provides its own named casts, it is safer to simply restrict the use of built-in casts. This is a kind of defense-in-depth approach, which is occasionally useful when a system's properties cannot be fully formally verified.

Requirement 4 (Primitive Types Only)**.** The set of labeled types must be identical to the set of primitive types.

Requirement 4 effectively precludes overloading of the type depth metafunction, thus preventing non-primitive classes from being typed by Covert C++. For example, it is safe to add the `iterator` type of `std::array<T>` to Covert C++, because this iterator behaves (almost) identically to a pointer. However, it is not safe to add the `iterator` to a `std::map<T>` (a kind of binary search tree), because the increment and decrement operators make control flow decisions that depend on key values. If the key values are sensitive, then they might be leaked.

The final requirement forbids the one kind of security class downgrade that is explicitly provided by Covert C++.

Requirement 5 (Restricted Label Casts)**.** The `se_label_cast()` function must not be used to downgrade security labels.

Requirements 2, 4, and 5 could have been automatically enforced by the Covert C++ type system. In many practical applications this might have been too restrictive for the developer, hence these behaviors are allowed for Covert C++ programs. However, the refactoring toolchain (Section 5.7) includes a Clang-based syntax checking tool which can warn the user if any of Requirements 2–4 is violated.

With all of these requirements satisfied, Covert C++ enforces an information-flow policy analogous to the Simple Security Theorem, which was formally proved for Core Covert (Theorem 4.1). That is, if an expression e can be typed as L, then none of the information which determines the value of e is typed as H. The argument is simple. Requirements 1–4 guarantee type and memory safety among both covert and non-covert types. Thus the type inference rules in Figure 5.3 are unequivocally respected by the program. The only inference rule which allows labels to be downgraded is Label Cast, but this behavior is disallowed by Requirement 5. With this information-flow property satisfied, Covert C++ also has the property of noninterference.

Theorem 5.1 (Covert C++ Enforces Termination-Sensitive Noninterference)**.** *If the safety requirements are satisfied, a well-typed Covert C++ program has the property of termination-sensitive noninterference against adversary $\simeq_{Adv}$ with perfect observational granularity.*

Proof. First, consider explicit flows to storage channels (i.e., outputs). The C and C++ languages have no built-in I/O. Instead, they rely on their own standard libraries and system libraries to provide this functionality. All of these library functions are parameterized by simple types (recall Figure 5.1). Covert C++ mandates that a covert type can only be converted to a non-covert type if all of its labels are L, by rule From SE. Thus H data cannot accidentally leak into one of these interfaces. The only storage channel which does not require a function call is shared memory. If the shared memory is L, then a transfer of H data to the shared memory would be an explicit flow leak, which has been obviated by simple security. If the shared memory is not L, then it is, by definition, a secret storage channel, not covered by $\simeq_{Adv}$.

The next step is to demonstrate the absence of implicit flows arising from H data, which leak sensitive information through the program trace. This can happen in one of two ways: program control flow (instruction fetches), or stack/heap/static memory accesses.

Consider memory reads and writes. The only Covert C++ typing rules which regulate memory accesses are Indirection, Subscript, and Arrow. These typing rules dictate that H pointers and subscript values cannot be used to access memory, and thus cannot leak the H data that determined their values.

Next, consider control flow. The derived typing rules Select-T', Iterate-T', Ternary-T', and Binary Logic-T govern the type checking for flow-of-control statements and operators in Covert C++. All of them dictate that H data cannot be used to determine a branch. Another subtle control flow feature of C++ is dynamic dispatch. When a virtual method is invoked on a pointer to an object using the `->` operator, the callee is looked up in the object's virtual member table (vtable). This is effectively a kind of branching operation, where the branch ultimately depends on the value of the pointer, because the pointer determines which

```cpp
template <typename T> struct type_depth :
    std::integral_constant<int, is_primitive<T>> {};
template <typename T> struct type_depth<T*> :
    std::integral_constant<int,
        type_depth<std::remove_cv_t<T>>::value + 1> {};
```

Listing 5.1: Type depth definition

vtable must be used, and thus which function to call. The ARROW rule prevents an H pointer from being used for dynamic dispatch.

The argument has thus far demonstrated that Covert C++ satisfies the weaker termination-insensitive noninterference property against $\simeq_{Adv}$. Termination-sensitive noninterference follows from the control flow result: because H data cannot influence program control flow, H data cannot cause the program to diverge. That is, a program cannot get "stuck" or loop forever. □

5.3 Type System Implementation

The Covert C++ security-type system is implemented entirely in header files as an interface library. The implementation consists of four parts: helper metafunctions, operator overloading, the `SE` class definition, and explicit cast operations.

Listing 5.1 shows one Covert C++ metafunction which was mentioned earlier in Section 5.1. This metafunction assigns to each labeled type the correct number of security labels for that type. For a pointer, it is one plus the type depth of the pointed-to type. For a non-pointer primitive type, the type depth is one. By default, all other types have a type depth of zero. This behavior can be extended by specializing the `type_depth` template in program code:

```cpp
using VecIt = std::vector<int>::iterator;
struct type_depth<VecIt>
  : std::integral_constant<unsigned, 1> {};
```

This specialization declares `VecIt` as a labeled type with one security label. Just as with pointers and other primitive types, `VecIt` can participate in overload resolution with the

Covert C++ operators and their typing rules. For instance:

```
std::vector<int> v = ...;
SE<VecIt, L> I1 = v.begin();
SE<VecIt, H> I2 = v.begin();
std::cout << *I1; // Allowed by Covert C++
std::cout << *I2; // Compiler error!
```

To demonstrate why the first access operation is allowed while the second is not, Covert C++'s pointer indirection implementation is given in Listing 5.2. Here, the `*` operator is being overloaded for all objects of type `SE<T, Ss...>`. C++ allows nearly every operator to be overloaded in this manner [90]. This implementation corresponds to the INDIRECTION rule in Figure 5.3. Notice the similarity between the template arguments and the structure of the inference rule. This is intentional, so as to match the implementation of the type system to its formal specification as accurately as possible.

The `_M_val` is the class field containing the lone private member of type `T`. The `ConstructSE_t` metafunction takes a labeled type `U` and a list of labels `Ls`, and produces type `SE<U, Ls...>`, assuming that the number of labels is correct for `U`. It also correctly handles the cases where `U` is a reference and/or is cv-qualified at the top level (recall that `SE` cannot encapsulate these types). For instance, the result of a pointer indirection is always an lvalue reference type `R &`. Hence, if `RetSs` is `H` and `RetT` is `int &`, then `Ret` will correctly resolve to the type `SE<int, H> &`.

The STL metafunction `std::enable_if_t` is defined as `void` if the given Boolean condition evaluates to `true`, otherwise it is undefined. If `Head<Labels>` is high when this overloaded operator is being considered, the undefinedness of `std::enable_if_t` does not itself cause a compiler error because C++ has a feature called SFINAE, or "Substitution Failure Is Not An Error" [10]. When a template function is being considered for overload resolution, the compiler first attempts to infer its template arguments. If any of these arguments fails to resolve to a type or a constant value, then the compiler does not immediately issue an error. This behavior characterizes SFINAE. Instead, the compiler will continue to search for

```
template <
  typename Labels = LabelList<Ss...>,
  typename = std::enable_if_t<Head<Labels> != H>,
  typename RetSs = Tail<Labels>,
  typename RetT = decltype(*std::declval<T &>()),
  typename Ret = ConstructSE_t<RetT, RetSs>>
inline Ret operator*() {
  return reinterpret_cast<Ret>(*_M_val);
}
```

Listing 5.2: Covert C++ indirection implementation

a candidate function whose type will resolve correctly. An error will only be issued when no viable function is found. When the compiler tried to resolve the typing for `*I2` in the example above, it first attempted to apply the definition in Listing 5.2. When that overload failed, the compiler attempted to fall back on the built-in indirection operator for pointers, which of course failed because `I2` has class type, not pointer type. Thus the compiler protests that the operation is invalid because `I2` is not a pointer.

This discussion may seem pedantic because for the indirection operator, the only observable difference with SFINAE is that the compiler emits a different error. Yet SFINAE is crucial to other aspects of the Covert C++ implementation. Unlike the indirection operator, Covert C++'s binary operators are not defined in a class; they are part of the global scope. For instance, the `+=` operator for `SE` types must compete with `std::string::operator+=` (string append) during overload resolution. Without SFINAE, any attempt to append two `std::string`s with `+=` would fail, because at least one type substitution would fail in the template arguments for Covert C++'s `+=` operator, triggering a compiler error. Hence SFINAE allows C++'s operators, as well as operators from other libraries, to coexist without conflicts.

Covert C++ has a total of 25 definitions in the style of Listing 5.2, some of which are macros that are expanded for similar operators or functions. These definitions yield 38 distinct overloaded operators, 7 implicit cast operators, and 7 explicit cast functions. The implementation in total comprises roughly 1,700 lines of code (LoC). The definitions range in complexity from just 4-5 LoC to 31 LoC. At the upper extreme, the macro which defines binary arithmetic operators has 17 clauses in its template argument list.

The addition of `SE` types and the associated casts and operator overloads substantially bloats a program's AST. Fortunately, this bloat does not translate to the compiled binary. All of the template arguments—the clauses in the inference rules—are entirely evaluated at compile time. All that remains of each overloaded operator function call after the templates have been processed, is the operation itself. And since the functions are visible to the compiler within each translation unit (because they are defined in header files), an optimizing compiler such as Clang or GCC will simply inline them. Although the AST may be much larger with the `SE` type wrappers, the binary should not be any larger.

To test whether or not the `SE` classes and operators actually bloat the compiled binaries, several small sample programs were selected and refactored to use the `SE` types in place of all primitive types. Each program was compiled by Clang at optimization level 3 (the highest supported by Clang), with and without the `SE` types. After stripping the symbols from each compiled binary, the binaries for each `SE`/non-`SE` program pair were compared. In all cases, the binaries of the sample programs with and without Covert C++ were identical down to the bit: their MD5 sums matched. Thus Covert C++ is truly a static analysis technique.

5.4 Validating Type System Correctness

One design goal of Covert C++ is to have the `SE<T,...>` types behave just like the primitive `T` types, the only exception being the constraints placed on values labeled with `H`. This goal is met in part with the help of the `decltype` keyword, as discussed in Section 5.1. C++ is a complex language, and hence additional measures are required to ensure that the design goals are met.

Covert C++ has been validated by a comprehensive test suite which exhaustively tests every operator and function against both the Covert C++ specification given in Figures 5.2

and 5.3, and the C++ standard specification [90]. The test suite was written using the LLVM Integrated Tester (lit) [9], the test framework used to validate LLVM and the Clang compiler. The Covert C++ language test suite comprises 72 files, each testing a specific Covert C++ definition against the Covert C++ specification and/or the C++ specification. These files are partitioned into PASS tests and FAIL tests.

PASS test files contain tests which are expected to compile and run without errors, and with correct computational results. For example, the indirection PASS file contains several tests, including this one:

```
SE<int *, L, L> xp = &x;
TEST(SE<int, L> &rx = *xp;) // CHECK: TEST
// CHECK-NEXT: SE<int*, L, L>: operator*
// CHECK-NEXT: END TEST
```

`TEST(com)` is a macro which runs the command `com`, emitting "TEST" and "END TEST" before and after the command is run. When Covert C++ is compiled with logging enabled, each overloaded operator and function emits a logging message with the type(s) of its argument(s) and/or the type of its return value. The "CHECK*" directives are processed by the FileCheck tool [5], which ensures that the text specified after "CHECK*:" appears in the program output, and in the specified order. For example, "CHECK-NEXT" asserts that the given content appears on the very next line after the output covered by the previous "CHECK*" directive. This particular test will fail if the given command fails to compile (e.g., due to a typing error), or perhaps a different `*` operator is resolved, in which case the logging information would not be emitted.

FAIL tests succeed only when the given file fails to compile, and all of the correct errors are emitted for each malformed Covert C++ statement. For example, the pointer indirection FAIL file contains this test:

```
SE<int*, H, H> hp;
SE<int, H> b = *hp; // expected-error \
  //{{indirection requires pointer operand}}
```

This particular test will succeed only if the statement `*hp` fails to type check, and with the specified error from the compiler referring to this specific line in the source code. For consistency, the Clang compiler is used for all of the tests.

Other than testing the specific typing rules given in Figure 5.3, the overloaded operators and casting functions are rigorously tested against the C++ standard. For instance, the C++17 standard [90] lists 11 clauses describing the behavior of `reinterpret_cast`. In general, one PASS test is constructed for each clause which describes conversions between types which `SE` can encapsulate. When a clause specifically disallows certain behavior, an appropriate FAIL test is constructed for that clause. In total, the core Covert C++ language test suite contains 170 individual PASS tests and 94 individual FAIL tests. It also contains 102 assertion tests to evaluate the metafunctions, such as `is_se_convertible`, etc.

5.5 Limitations

The most striking drawback of Covert C++ is that it unequivocally forbids the use of high pointers for anything other than pointer arithmetic. However, this is only a restriction for *pure* Covert C++: the typing rules described in Figure 5.3. Chapter 8 addresses this problem by introducing oblivious types which can be used to access memory, regardless of whether the oblivious types itself is high or low.

The other noteworthy limitations of Covert C++ are the consequences of limitations inherited from C++. It was mentioned in Chapter 2 that C++'s template system was not designed with the explicit intent of supporting Turing-complete computation. Admittedly, the C++ template system certainly was not designed to have another type system superimposed on top of it, as with Covert C++. Therefore it is not surprising that there are several features of C++ that Covert C++'s security-type system cannot accommodate.

The `SE` template cannot wrap bit fields because a bit field type cannot be a template argument. This is disappointing because there could be many scenarios where some values in a bit field should be kept secret, and others need not be secret. One workaround is to have something like

```
struct {
  SE<int,L> low_bits;
  SE<int,H> high_bits;
};
```

and use macros with bitwise operators to simulate distinct bit field accesses within `low_bits` and `high_bits`, but this is obviously an inconvenience to the developer.

Additionally, the `SE` template wrappers can confuse a C++ compiler's overload resolution algorithm. Function overload resolution in C++ is complicated. The algorithm description runs over 20 pages in the C++ standard specification [90]. Consider the following example which requires overload resolution:

```
void foo(const char*);
void foo(bool);
char *str;
int main() { foo(str); }
```

Resolution for the call to `foo()` is non-trivial, since neither prototype is an exact match. A pointer to some `T` can be implicitly converted to adopt `const`/`volatile` qualifiers. It can also be implicitly converted to a `bool`. According to the C++17 standard, clause 16.3.3.2(4.1), "A conversion that does not convert a pointer, a pointer to member, or `std::nullptr_t` to bool is better than one that does" [90]. Hence the first `foo` should be selected. This clause from the C++17 specification is one of several which describe how viable functions for overload resolution should be ranked. Now consider what happens when the argument types are wrapped by the `SE` template:

```
void foo(SE<const char*, L, L>);
void foo(SE<bool, L>);
SE<char*, L, L> str;

int main() {
  foo(str);
}
```

The call to `foo()` with `str` cannot be compiled because the overload is ambiguous. Covert C++ allows both of the implicit conversions `SE<char*, L, L>` → `SE<const char*, L, L>` and `SE<char*, L, L>` → `SE<bool, L>`. However, the C++ overload viable function ranking algorithm assigns the same rank to both overloads of `foo`. Specifically, there is no clause to distinguish between conversions to instantiations of the same template, and which differ only in their template arguments. Thus the compiler cannot decide which `foo()` should be called.

Spectre-Style Attacks. Covert C++ unfortunately does not offer any solution for the recently discovered Spectre-style attacks on speculative execution side channels [96]. As discussed earlier in Section 5.2, Covert C++ only guarantees noninterference if certain assumptions have been satisfied, one of which is memory safety. Spectre attacks exploit the fact that even if a program satisfies this assumption, speculative execution may still allow an out-of-bounds read or write to occur. Covert C++ does, however, offer protection against the more recently documented NetSpectre AVX-based covert channel attack [139], which exploits conditional execution of AVX2 instructions. Because Covert C++ prevents secret data from influencing control flow, a decision to execute an AVX2 instruction cannot depend on secret data.

5.6 Usage Model

This section demonstrates how Covert C++ can be used in practice, and how it can interface with legacy libraries and APIs that are not security-typed.

5.6.1 Example: Secure `memcmp()`

Once again, recall the `memcmp()` example from Chapter 1. An equivalent Covert C++ implementation of the standard library `memcmp()` is shown at the beginning of Listing 5.3. This implementation is nearly identical to that from Listing 1.1, except that all of the

```
 1 SE<int, L> memcmp(SE<const uint8_t*, L, L> s1,
 2                   SE<const uint8_t*, L, L> s2,
 3                   SE<std::size_t, L> n) {
 4   while (n--) { // optimized implementation
 5     SE<int, L> diff = *s1++ - *s2++;
 6     if (diff) return diff;
 7   }
 8   return 0;
 9 }
10
11 SE<int, H> memcmp(SE<const uint8_t*, L, H> s1,
12                   SE<const uint8_t*, L, H> s2,
13                   SE<std::size_t, L> n) {
14   SE<int, H> res = 0;
15   while (n--) { // secure implementation
16     SE<int, H> diff = *s1++ - *s2++;
17     res = covert::ternary(
18         diff != 0 & res == 0, diff, res);
19   }
20   return res;
21 }
```

Listing 5.3: Optimized and secure implementations of `memcmp()` in Covert C++

primitive types have been replaced by low canonical types[2]. If the arguments `s1` and `s2` had instead been pointers to high data, then by the rules Unary Arith Op, Indirection, and Binary Arith, `diff` on Line 5 would require an `H` security label. Hence the contextual conversion to `bool` in Line 6 would fail, and the compiler would issue an error.

To construct a `memcmp()` implementation which can accept buffers containing `H` data, a different strategy must be adopted when writing the loop. Recall the second memory comparison program from the Core Covert chapter, given in Listing 4.2. That program achieved noninterference by fixing the number of loop iterations, and updating the result value during each iteration.

[2]The other difference is that the first two arguments are `const uint8_t*` instead of `const void*`. This is because `void` pointers point to a memory location containing a value whose type is unknown to the compiler. Since the type of the pointee is unknown, it is unclear how many labels should be assigned to it. Hence in Covert C++, `void` pointers are labeled such that only the pointer itself receives a label. So although `void` pointers are allowed in Covert C++, they are not necessarily as useful as they are in regular C++.

The second `memcmp()` implementation in Listing 5.3 follows the same strategy. This allows it to accept high buffers and still type check. The secure `memcmp()` does not break the `while` loop when a difference between the two buffers is discovered. Rather, the value `ret` is assigned a new non-zero value when a non-zero difference is found for the first time. A custom non-branching ternary function performs the update to `res` obliviously.

The `covert::ternary()` function allows the ternary condition to be high. If the condition is high, then the return value will also be high. The non-branching behavior is achieved using the `cmovz` x86 assembly instruction, whose behavior was discussed in Chapter 2. By employing these non-branching heuristics, the secure `memcmp()` implementation exhibits precisely the same semantic behavior as an ordinary `memcmp()`, except that this version is typable in Covert C++, and thus respects noninterference for its high inputs.

The Covert C++ implementation of `memcmp()` overloads the `memcmp` name with two definitions: one optimized, the other secure. This overloading is invisible to the developer. The compiler will automatically select the best implementation, depending on the types of the arguments. For instance, if both `s1` and `s2` point to low data, then the optimized `memcmp()` will be an exact match, hence the compiler will select this version. Otherwise, the From SE rule will not allow the argument labels to be downgraded, so the secure `memcmp()` will be the only valid candidate for overload resolution. If one buffer argument is high and the other is low, then the low argument can be implicitly upgraded via the SE To SE rule. This overloading scheme offers the best of both worlds: security when it is needed, and performance when it is not; and the developer is not required to consciously make this decision.

Even though the return value from a call to the secure `memcmp` is high, it can still be used as a branch condition. For instance:

```
if (se_label_cast<int, L>(memcmp(x, y, 128)) {
  ...
}
```

If either `x` or `y` points to high data, then the secure `memcmp()` will be called. By declassifying the result of this `memcmp` and using it to branch, the result may be leaked. However, the details of that computation will not leak. Depending on the security requirements of the particular application, it may be acceptable for an adversary to know whether or not the `memcmp()` operation succeeded, so long as she cannot infer the input values. This decision must be made by the developer, and must always be made explicit with an `se_label_cast()`. Although all secret data is high, it is not necessarily true that all high data should be kept secret.

5.6.2 STL compatibility

Unlike in libc, most of the C++ STL classes and definitions are entirely implemented in C++ header files. This is necessary because templatized functions cannot be compiled without knowledge of the template arguments. These functions are compiled on-the-fly when their template arguments are instantiated with new types. Hence it is not difficult to use STL templates with the `SE` type container:

```
std::forward_list<SE<int, H>> fl = {1, 2, 3};
fl.push_front(0);
fl.reverse();
```

Notice that all of the above operations should be secure, because none of them must make a control-flow or memory-access decision, based on a high value in one of the containers. When an attempt is made to perform a non-secure operation on high data, such as an equality test on forwardly-linked lists, the compiler emits an appropriate error message:

```
/include/c++/v1/forward_list:1696:13: error:
  no viable conversion from 'SE<bool, H>' to 'bool'
    if (!(*__ix == *__iy))
        ^~~~~~~~~~~~~~~~~
test.cpp:4:11: note: in instantiation of function
  template specialization 'std::__1::operator==' requested here
    eq = fl == fl2;
            ^
```

The C++ compiler has indicated that the `==` operator for forward lists is non-secure (vulnerable), moreover it reveals the precise location in the STL header `<forward_list>` where the leak would have occurred, had the code compiled successfully.

The Covert C++ toolchain also has a lit test suite for STL functions. The tests check whether the C++ compiler correctly signals which functions are secure and which are vulnerable, in a manner similar to that which was described in Section 5.4. The Clang compiler has so far been used to test 120 C++ STL functions. Clang identified 104 secure functions and 16 vulnerable functions. One false positive was noted. An STL function was implemented as follows:

```
template <...> bool fun(...) {
  ...
  return (E1 != E2);
}
```

With `E1` and `E2` instantiated as `SE<T, H>` types for some labeled type `T`, `E1 != E2` evaluates to a value with `SE<bool, H>` type, which cannot be converted to the `bool` return type. Even though this function is secure, the compiler rejected it.

When an STL function is not secure, the alternative is to either refrain from using it, or to write an alternative implementation which employs heuristics similar to those deployed in the `memcmp` example. With recent editions of C++ the STL has become quite large. Therefore it would be an unrealistic goal to reimplement every vulnerable STL function in Covert C++. However, Covert C++ implementations of the generic algorithms found in the STL algorithms library are under development. 20 of these algorithms have been implemented.

5.6.3 Legacy Code Compatibility

It is unrealistic to expect that an entire program will be written in Covert C++. Modern application programs usually consist of a relatively small body of original code, with numerous calls to external library functions, including standard library functions. Even though these functions will not explicitly accept `SE` arguments, many can still be used by Covert C++ code. If the library functions are themselves templates, then they may be instantiated with `SE` types, as with the STL functions discussed in the previous section. Otherwise, the developer

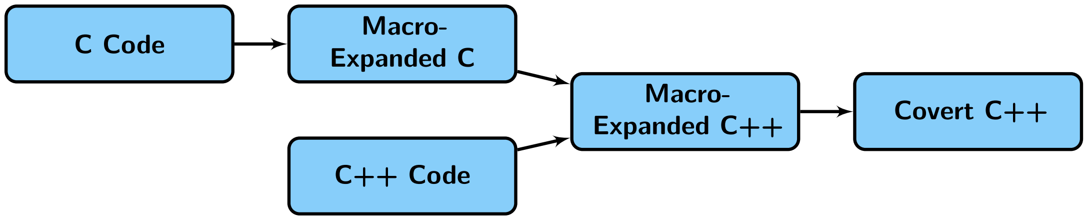

Figure 5.4: The Covert C++ refactoring pipeline

can use the NVT (Chapter 6) to determine whether or not a given library function is secure.

5.7 Refactoring Toolchain

Although it is not required to use Covert C++, the Covert C++ toolchain includes several tools which can be deployed to interactively refactor legacy codebases from C/C++ into Covert C++. The toolchain components are depicted in Figure 5.4. Optionally, the refactoring pipeline begins by expanding preprocessor macros, because it is notoriously difficult to perfectly refactor code containing or within macros. Also, C code must be converted into C++ code, because C is not a perfect subset of C++ [152]. Finally, an interactive tool based on Clang-Tidy [3] converts the C++ code into Covert C++ code. This tool can also warn the user of any potentially dangerous casts which may violate Covert C++'s safety requirements for noninterference, as described earlier in Section 5.2.

The refactoring pipeline has thus far been used to successfully refactor Amazon's implementation of TLS/SSL [137] (written in C, approx. 6,000 LoC) and the TinyXML2 [157] XML parsing tool (written in C++, approx. 2,000 LoC) into Covert C++.

5.8 Case Study 1: DRM on SGX

This brief case study demonstrates how to use the Covert C++ toolchain to refactor existing C++ codebases into Covert C++. The Intel SGX SDK [11, 15] includes some sample enclave applications, including an application for digital rights management (DRM). One typical

```
1 typedef struct _replay_protected_pay_load
2 {
3     sgx_mc_uuid_t mc;
4     uint32_t mc_value;
5     uint8_t secret[REPLAY_PROTECTED_SECRET_SIZE];
6     activity_log log;
7 }replay_protected_pay_load;
```

Listing 5.4: A structure containing an SGX enclave secret

use of DRM is to encrypt intellectual property, such as music and movies, so that it can only be used under certain circumstances. For instance, a DRM policy may specify that only a particular user application can open a certain file, or that a movie rental should expire after 72 hours. Intel SGX is an ideal platform for DRM because it provides an area of shielded execution where a cryptographic key can be stored and used in isolation. Even on an untrusted platform with a potentially malicious adversary, the adversary will not be able to view the contents of the key in plaintext.

However, SGX does little to protect enclave applications from storage channel and side channel leaks. This fact is admitted in the SGX User's Guide [15]. The noninterference guarantee provided by Covert C++ can close this security gap.

Listing 5.4 shows the definition of the structure which wraps the secret key. Specifically, the key is held in the array member `secret` in Line 5. Hence this member should be treated by the Covert C++ security-type system as a program secret (i.e., with a high security label). The developer does not need to manually refactor this code. He only needs to add a `SECRET` annotation to each declaration of a variable or member which should be treated as a program secret:

```
    uint8_t secret[REPLAY_PROTECTED_SECRET_SIZE] SECRET;
```

When the refactoring tool sees any declaration with the `SECRET` annotation, it assigns an `H` label to it. Otherwise it assigns an `L` label. The refactoring tool can be run as follows:

```
$ cpp2covert -checks=* -p . DRM_enclave/DRM_enclave.cpp
```

The `-checks=*` flag enables all refactoring checks. The `-p` flag instructs the refactoring tool to look for a JSON compilation database [8], which contains all of the information—flags, include directories, preprocessor definitions—used to compile the given source file. This allows the refactoring tool to be as precise as the compiler when parsing the given source(s).

Unfortunately, the output is a mess:

```
include/tlibc/string.h:108:22: warning: 'strncasecmp' declared with
                               primitive type 'int'
int    _TLIBC_CDECL_ strncasecmp(const char *, const char *, size_t);
~~~                  ^
SE<int, L>
include/tlibc/string.h:108:46: warning: Parameter declared with
                               primitive type 'const char *'
int    _TLIBC_CDECL_ strncasecmp(const char *, const char *, size_t);
                                 ~~~~~~~~~~~~^
                                 SE<const char *, L, L>
include/tlibc/string.h:108:60: warning: Parameter declared with
                               primitive type 'const char *'
int    _TLIBC_CDECL_ strncasecmp(const char *, const char *, size_t);
                                               ~~~~~~~~~~~~^
                                               SE<const char *, L, L>
include/tlibc/string.h:108:68: warning: Parameter declared with
                               primitive type 'size_t' (aka 'unsigned long')
int    _TLIBC_CDECL_ strncasecmp(const char *, const char *, size_t);
                                                             ~~~~~~^
                                                             SE<size_t, L>
```

The refactoring tool found several hundred declarations which are not in canonical form (because they are raw primitive types). Many of these declarations are in header files, and basically unrelated to the task of securing the DRM key.

When writing a Covert C++ program from scratch, it is considered good practice to label all of the primitive types either high or low. When dealing with legacy code, this principle can become too tedious. For larger programs, the refactoring tool provides a `-secret-only` flag, which causes the tool to only rewrite declarations that are annotated as `SECRET`:

```
$ cpp2covert -checks=* -p . -secret-only -fix DRM_enclave/DRM_enclave.cpp
DRM_enclave/DRM_enclave.cpp:54:13: warning: ’secret’ declared with
                                   primitive type ’uint8_t [32]’
    uint8_t secret[REPLAY_PROTECTED_SECRET_SIZE] SECRET;
    ~~~~~~~ ^
    SE<unsigned char, H>
DRM_enclave/DRM_enclave.cpp:54:5: note: FIX-IT applied suggested code
                                  changes
    uint8_t secret[REPLAY_PROTECTED_SECRET_SIZE] SECRET;
    ^
```

The `-fix` flag instructs the tool to automatically apply all of the suggested fixes. The `secret` member has been refactored as expected:

```
    SE<unsigned char, H> secret[REPLAY_PROTECTED_SECRET_SIZE];
```

Now `secret` is protected by Covert C++'s security typing. This means, for instance, that `secret` must be disclosed before being passed to any SGX SDK APIs:

```
    SE<uint8_t *, L, H> secret = data2seal.secret;
    ret = sgx_read_rand(
              se_label_cast<uint8_t *, L, L>(data2seal.secret),
              REPLAY_PROTECTED_SECRET_SIZE);
```

In this example, the `sgx_read_rand()` API is used to generate entropy bits to forge a new encryption key.

As the developer continues to follow `secret` and other program secrets through the program call graph, he will frequently encounter two kinds of problems. First, he must be careful when disclosing secret data (as above) to non-Covert C++ APIs. In this case, it is probably safe to trust this SGX API. For libraries which are not specially designed to prevent side-channel leaks, etc., it is safer to avoid those libraries. Or, the developer could use the NVT (the topic of the next chapter) to verify a questionable API call.

Second is the problem of label creep. If the developer designates too many inputs and variables as secret, then the propagation of secret information through the program via copies and computations could result in most of the program data becoming secret. Since the behavior of secret data is restricted, this could make it difficult to maintain or expand the program.

Chapter 6

Verifying Noninterference for Compiled Programs

Although Theorem 5.1 establishes that a program which is typable in Covert C++ has the property of noninterference at the source code level, a compiled Covert C++ program might not have the property of noninterference. The reason for this discrepancy is that optimizing compilers only guarantee preservation of semantics, not preservation of execution properties independent of semantics [68]. Consider again the secure implementation of `memcmp()` from Chapter 5. Unlike the optimized implementation of `memcmp()`, this definition should respect noninterference for the two buffers being compared, because the function is typable in Covert C++. Hence the values contained in each buffer do not alter the function's control flow or memory trace; the `while` loop always iterates `n` times.

However, a sufficiently smart compiler may be able to infer that as soon as `diff` is assigned a non-zero value, the remaining loop iterations cannot again alter the value of `diff`, and thus the remaining loop iterations could not affect the return value. Hence the compiler would short circuit the control flow because this yields faster code, without changing its semantic behavior. This is precisely the definition of an optimization. However, this optimization clearly invalidates the noninterference property that was established by Covert

C++'s information-flow security typing.

This gap between compiler correctness and security has been considered by D'Silva et al. [68]. The authors suggest several approaches for developing security-preserving compilers, and for testing whether existing compilers may preserve security. The latter approach entails fixing an adversary model, and developing tools which can compare an adversary's view of a optimized computation against her view of an unoptimized computation. The test suites should consist of "small pieces of code that represent security intent" [68]. The Covert C++ noninterference test suite follows a similar strategy.

6.1 The Noninterference Verification Tool (NVT)

This section introduces the Noninterference Verification Tool (NVT). Whereas LLVM lit [9] was used to validate Covert C++'s implementation against its specification (Section 5.4), the purpose of the NVT is to verify that the noninterference property is preserved by the compiler. The NVT operates by fuzzing all secret function inputs over many (thousands) of iterations, and monitoring the function's memory and output traces during execution. If any two combinations of secret input values to a function cause that function to yield inconsistent traces, then the NVT will report a test failure, and emit the values of those secret inputs which revealed the inconsistency. A test failure indicates that the target test module function is not safe to use for secret data; it does not satisfy the noninterference requirement.

In essence, the NVT's verification mechanism is a manifestation of the strong dependency principle (Definition 2.5): given a set of fixed low inputs, the NVT verifies that no output and no memory access is strongly dependent on any secret input (outputs and memory accesses are assumed to be public). If the number of possible values for the secret inputs exceeds the number of fuzz iterations, then the NVT only approximates the strong dependency principle. This technique of combining input fuzzing and dynamic memory-trace analysis is original to this dissertation—it has not been considered in any prior works.

```
for (i = 1; i <= 256; i++):
  if (NVT_test_init[i] exists) and
        (NVT_test_begin[i] exists):
    for (j = 0; j < fuzz_iterations; j++):
      reset test module heap
      fuzz_arg <- get fuzz_arg_sz bytes
      NVT_test_init[i](fuzz_arg, fuzz_arg_sz)
      // begin recording memory and output traces
      NVT_test_begin[i]()
      // stop recording memory and output traces
      if (memory_trace[j] != memory_trace[j-1] or
          output_trace[j] != output_trace[j-1]):
        exit(1); // report test failure
```

Listing 6.1: Pseudo-code description of the NVT

The NVT can be configured to accommodate an arbitrarily strong adversary model by allowing the user to adjust the granularity of the memory trace. For example, if the adversary is only allowed to use forced page eviction [171] then the granularity can be set to 12 bits, the number of bits required to address 4 KB of memory (the size of a small x86-64 page). Thus the NVT will record addresses with their 12 least significant bits masked away. Assuming an adversary who is able to use the PRIME+PROBE attack on 64-byte cache lines, the granularity would be set to 6 bits.

The operating procedure of the NVT is described in Listing 6.1. The NVT loads a *test module* built as a shared object or DLL, and searches for exported functions called `NVT_test_init*()` and `NVT_test_begin*()`, where `*` ranges from `1` to `256`. For each test, the initialization function is used to initialize program variables with the fuzzed secrets, and the second function runs the test. The test module together with other dynamically loaded modules (e.g., libc, libc++, etc.) called by the test module constitute the *test application*.

On each invocation of `NVT_test_begin*`, the NVT records a trace of the test application's execution consisting of tuples, each of which contains three pieces of information:

1. the memory address that was accessed,
2. the number of bytes that were read/written, and

3. the type of access: `r` (read), `w` (write), or `bb` (dynamic basic block).

The `r` and `w` tags trace memory accesses, while the `bb` tags trace process control flow. A *dynamic basic block* is a contiguous unit of dynamic execution, i.e., a sequence of instructions with a single point of entry and a single exit. Thus the sequence of dynamic basic blocks executed by a process is sufficient to characterize the control flow of the process.

When the granularity of the adversary is greater than 0—that is, when the adversary does not have a perfect view of the memory trace—the second piece of information is omitted. Instead, the trace records the sequence of masks that were accessed and the type of access on each mask. For example, if the adversary's granularity is configured to 6 bits and a process reads 4 bytes from address `0x0000007e`, then the NVT will record read accesses on masks `0x00000040` and `0x00000080` because this particular read covers those two masks.

The NVT is implemented as a client module for DynamoRIO [43, 46, 47], a framework for building dynamic analysis tools. DynamoRIO is an event-driven system, where clients register callbacks for selected events, such as when a new dynamic basic block is loaded, when a system call is made, or when the target program loads a new module.

The NVT transforms and instruments application modules (including the test module, libc, libc++, etc.) by registering dynamic basic block callbacks with DynamoRIO. When DynamoRIO is invoked with the NVT as its client, DynamoRIO commences execution of the target application (the `DynLoader`, described in Section 6.2). Each time DynamoRIO encounters a new dynamic basic block while the target application is running, it signals each of the NVT's basic block callbacks. After the NVT has been allowed to inspect and instrument each new dynamic basic block, DynamoRIO adds the (possibly) transformed block to a cache, and resumes execution. Any subsequent jump to that same untransformed block is then redirected to the transformed block in the cache.

6.1.1 Input Fuzzing

DynamoRIO clients can be extended with modules from the Dr. Memory framework [44, 45]. The NVT uses the Dr. Fuzz extension to iteratively fuzz the secret inputs. When the test module is loaded, the NVT registers all `NVT_test_begin*` functions with Dr. Fuzz. This involves two callbacks: a pre-call callback which "mutates" a new fuzzed input for the target function, and a post-call callback which decides whether to jump back to the pre-call, or to stop fuzzing and continue with regular execution. In the NVT, the post-call also checks whether the trace digests of the just-completed fuzz iteration match those of the previous fuzz iteration. If the traces match, it resets both of the trace digests and triggers the next fuzz iteration. Otherwise, the NVT reports a test failure, and dumps the secret input value sequences which yielded differing traces.

6.1.2 Recording the Memory Trace

For each loaded module and each dynamic basic block, the NVT must first perform an application-to-application transformation. X86 platforms commonly use the `rep`/`repne` family of instructions, which repeat a given operation on a string—ideally until the string's null terminating character is found, or a counter expires [58]. From the perspective of system software (including DynamoRIO) the `rep` instruction appears to execute atomically, though it may touch millions of memory locations. The application-to-application transformation searches the loaded module for each instance of any instruction in the `rep` family, and expands it into a semantically equivalent loop. For instance, if the `rep` is repeating a `mov` for the purpose of copying a string to another location in memory, the transformation will yield a replacement instruction sequence in a new dynamic basic block consisting of at least one `mov`, and an instruction to increment the memory address(es), and a branch with a test for the termination condition. The transformed loop will touch precisely the same addresses and in the same order as the `rep` instruction. It will also allow a dynamic analysis tool to instrument the now-isolated `mov` instruction to record each individual access.

After the application-to-application transformation is complete, the NVT traverses each dynamic basic block encountered during test application execution. It inserts inline instrumentation instructions to record the sequence of memory accesses made by the dynamic basic block into the NVT's memory trace buffer. At the very beginning of the dynamic basic block, the NVT inserts instrumentation to record the address of the block.

The NVT then iterates through the instructions in the dynamic basic block, looking for any instruction with operands that access memory. For each such operand, it inserts inline instrumentation to record the address being accessed, the number of bytes transferred, and the type of access (read or write). Additional processing is required for VSIB-addressed instructions [58]. Vector scaled index byte (VSIB) mode is an addressing mode for vector instructions such as `vpgather` and `vpscatter` which can represent an array of linear addresses [58]. The Intel architectural specifications and programming guides do not specify whether VSIB instructions access the array of addresses in serial or parallel order, or if the access sequence is nondeterministic[1]. The NVT's treatment of VSIB instructions is conservative. It simply records the VSIB address operands into the trace in the exact order in which they are given by the index vector.

At the end of the dynamic basic block, the NVT inserts a clean call to process the memory trace buffer. If the granularity of the adversary model not perfect, then all addresses are masked according to the granularity. If any particular access covers n masks where $n > 1$, that entry in the trace buffer is expanded into n entries, one for each mask that was touched. If the granularity is perfect, then no changes are made to the trace buffer. The trace buffer is then is appended to an MD5 hash digest which characterizes the memory trace, and then the memory trace buffer is reset.

The NVT can only record memory traces for single-threaded test application functions. This is not a limitation posed by DynamoRIO. It is a consequence of the definition of noninterference (Definition 2.7), from which the NVT algorithm (Listing 6.1) was derived.

[1]The performance analysis in Chapter 8 suggests that the `vpgather` instructions do not read from their operands in parallel.

6.1.3 Recording the Output Trace

The second requirement for noninterference is that the sequence of public outputs not vary with respect to the secret inputs. That is, no public output can strongly depend on any secret input. Similar to verifying memory-trace obliviousness, the strategy adopted by the NVT is to record a trace of program outputs in an MD5 hash digest. However, the problem of verifying classical noninterference for a compiled program is more nuanced than verifying memory-trace obliviousness. The reason is that there are potentially many ways in which a program can produce publicly-observable output, including but not limited to system calls, writing data to shared memory, and I/O.

On UNIX-like systems, it is typical for programs to emit output through I/O channels by means of a system call. On Linux the `SYS_write` system call writes a buffer to a file descriptor. Despite its name, a file descriptor is simply an identifier which can refer to a pipe to another process, a network socket, an I/O channel (e.g., `stdin`, `stdout`), or of course, a file. Hence a `SYS_write` is the preferred mechanism to emit output to a variety of storage channels.

In total, the Linux kernel supports several hundred system calls. In a sense, any of these system calls should be treated as a kind of channel through which information might be leaked. If sensitive information is passed as an argument to a system call, then an adversary with privileged access to the platform could read those parameters directly. Additionally, the mere fact that a particular system call was invoked could signal to the adversary that some particular path of execution was taken by the program. If that path was influenced by sensitive data, then this would violate noninterference.

To detect storage channel leaks, the NVT adopts a conservative approach. When the test application invokes a system call, the NVT intercepts the system call and checks its number against a whitelist. A system call is only included in the whitelist if the NVT knows how to record its arguments to the output trace. System calls which may introduce nondeterminism cannot be whitelisted.

One example of a whitelisted system call is `SYS_write`. `SYS_write` takes three parameters: (1) the file descriptor to which to write, (2) a pointer to the buffer to write, and (3) the size of the buffer. When the test application invokes `SYS_write`, the NVT intercepts the call and adds the descriptor number, buffer contents, and buffer size to the output trace digest. The need to record the buffer into the trace should be obvious. The need to record the file descriptor is less obvious. A call to `open()` a file on Linux will return a file descriptor—an integral identifier—referring to the requested file. The value of the file descriptor is not necessarily chosen deterministically by the Linux kernel. For instance, when a program opens `foo.txt`, then closes it, and then opens it again, a different file descriptor may refer to `foo.txt` the second time it is opened. In this case, the behavior of `open()` on account of the operating system is benign, so perhaps the file descriptor should not be recorded into the output trace. On the other hand, suppose that the test application is simultaneously managing several file descriptors as values in a lookup table. If the keys of that table are sensitive, and lookup is used to determine which descriptor should be written to, then the value of the file descriptor might leak sensitive information about the keys.

One corollary to this example is that the system call `SYS_open`, which implements `open()`, cannot be whitelisted because the file descriptor value is chosen nondeterministically. Hence a test application which calls `SYS_write` at any point must have the file descriptor passed in as a fixed input.

As argued earlier in Chapter 5, "outputs" to shared memory are monitored at the source code level by the Covert C++ type system. It is the responsibility of the developer to recognize and identify which variables, data structures, etc., are to be shared with another process, and to annotate all of these objects as public (i.e., with an `L` label). Hence, the Covert C++ type system will prevent program secrets from leaking into shared memory through explicit flows.

The NVT does not verify the absence of flows to shared memory in the compiled binary, for several reasons. First, the implementation would be complicated. The NVT would need to intercept system calls that map shared memory pages into a process, make note of which regions of memory are shared, and record a trace of any writes to the shared region(s). This last step would entail looking up the destination address of every write in a list of shared regions to determine if it is a write to shared memory, and thus should be added to the output trace. Currently, to keep the memory tracing fast, each memory access made by the test application is instrumented by a short sequence of inline instructions to record the address being accessed. The shared memory lookup procedure would instead require a clean call (new stack, registers, etc.) to be inserted before each memory access, and thus would dramatically slow down the tracing procedure, reducing the number of fuzz iterations which could feasibly be performed per test. Shared memory also introduces a source of nondeterminism if the test application is additionally reading from shared memory. As mentioned above, nondeterminism can be a source of false positives for NVT tests.

6.1.4 Managing Application Heap Memory

If any test application function allocates and uses heap memory (e.g., via a `new` expression or a direct call to `malloc()`), then there is no guarantee that libc or the operating system will allocate memory at precisely the same location for each fuzz iteration. In fact, this is almost never the case. Similar to the issue with `open()`, the various libc heap allocation routines allocate memory nondeterministically.

The NVT addresses this issue by intercepting calls to any libc routine which allocates memory, including `malloc()`, `aligned_alloc()`, `memalign()`, and `posix_memalign()`. Each of these calls by the test application is instead forwarded to the NVT's internally managed heap allocator, which allocates memory deterministically. Before each fuzz iteration, this heap is reset to a fixed state. Hence, if the test application makes a certain sequence of heap memory requests during fuzz iteration i, and also makes an identical sequence of heap memory requests during fuzz iteration j, the NVT heap allocator will guarantee that each pair of corresponding requests is satisfied identically.

```
#include "NVT.h"
NVT_TEST_MODULE;

SE<uint8_t, H> buf1[MAX_TEST_SIZE], buf2[MAX_TEST_SIZE];
SE<unsigned, L> buf_size;
SE<bool, H> ret;

NVT_EXPORT void NVT_test_init1(unsigned char *fuzz,
                               unsigned size) {
  buf_size = size / 2;
  unsigned char *src = fuzz;
  for (int i = 0; i < buf_size; ++i) { buf1[i] = src[i]; }
  src = fuzz + buf_size;
  for (int i = 0; i < buf_size; ++i) { buf2[i] = src[i]; }
}

NVT_EXPORT void NVT_test_begin1(void) {
  ret = memcmp(buf1, buf2, buf_size);
}
```

Listing 6.2: Using the NVT to verify `memcmp()`

6.2 Example: Verifying `memcmp()`

Recall the Covert C++ `memcmp()` example from Section 5.6.1. Listing 6.2 demonstrates how to set up an NVT test module to verify noninterference for `memcmp()`. The `NVT_TEST_MODULE` declaration exports a special symbol which informs the NVT that this is a test module which exports NVT tests. Each pair of `NVT_test_init*` and `NVT_test_begin*` define a distinct NVT test. Each test module can export up to 256 tests. The test module is loaded by a separate executable, the `DynLoader`, which simply calls each test exported by the test module in sequence.

The test module in Listing 6.2 exports one test for the secure `memcmp()` function defined in Listing 5.3. Note that the secure version will be called because at least one of the buffer arguments is secret. The test initialization function loads each input buffer with `size / 2` bytes of fuzzed data. The test begin function invokes `memcmp()` on the two buffers. Only the trace of the test begin function will be recorded. For 10,000 fuzz iterations, 8 bytes of fuzzed data per iteration (thus 4 bytes per buffer), and the strongest possible adversary model with perfect granularity, the NVT reports a pass for this test.

If the labels of `buf1` and `buf2` had instead been `L`, then the performance-optimized `memcmp()` would have been called instead. In this case the NVT reports a test failure, with the other test parameters unchanged.

The NVT can also be directed to emit the program traces to a log file. The traces for the secure and optimized `memcmp()` tests described above are given in Appendix A.

6.3 Results

As mentioned earlier in Section 5.6, the Covert C++ algorithms library currently supports 20 generic algorithms. These algorithms employ a combination of non-short-circuiting loops and non-branching ternary functions to achieve noninterference. Each of the Covert C++ algorithms has been tested over several STL containers, including `list`, `forward_list`, `array`, `vector`, and `deque`. Each test runs for 10,000 fuzz iterations. Depending on the algorithm, the container content and/or the algorithm parameters are fuzzed. For instance, the tests for `std::find()` fuzz both the container contents and the search query.

All existing Covert C++ algorithms which type check pass all NVT tests for any of the sequential containers.

Chapter 7

Secure Multi-Party Computation

This brief chapter generalizes the description of Covert C++ given in Chapter 5 to cover a proper subset of the decentralized label model proposed by Myers and Liskov [118, 120]. Rather than assigning security labels to data according to its sensitivity level, the decentralized label model assigns labels to data according to its ownership. The owner of some data is able to control how his or her data is used: how it is created, and how it is declassified. These owners are mutually distrusting. Owner A should not be able to observe owner B's data unless owner B explicitly allows this in his access control policy. The original model by Myers and Liskov also described more expressive access controls on data, including delegation. Those additional features are beyond the scope of this dissertation.

The problem addressed in this chapter is the secure multi-party computation (SMPC) problem which was introduced in Section 2.4. In SMPC, some group of n principals wish to compute a function $f(x_0, x_1, \ldots, x_{n_1})$ over their respective inputs x_i. Moreover, principal i must not be able to infer the value x_j for $i \neq j$ from any observable aspect of the computation (e.g., side channels). Some descriptions of SMPC go even further: no principal should be able to infer any of the other principals' inputs from f's output. One solution is differential privacy [69, 109, 130], a probabilistic technique which adds noise to the computational result. Again, this topic is beyond the scope of this dissertation.

7.1 A Generalized Label Model for Covert C++

The solution proposed in this chapter is to generalize the security labels in Covert C++ to accommodate an arbitrary lattice structure. The proper subset of the decentralized label model implemented in Covert C++ is referred to in this dissertation as the *generalized label model*. This model allows principals participating in a computation to be represented as points on a lattice. When a computation *joins* two principals' data (e.g., by arithmetic), the label assigned to the result is the least upper bound of the principals' labels on the lattice. Hence every value in a Covert C++ SMPC program keeps a sort of record of whose data influenced that value.

The description of Covert C++ in Chapter 5 was admittedly incomplete. Covert C++ is not defined in terms of the binary priority lattice consisting of `L` and `H`. Instead, all Covert C++ operators, metafunctions, and types are actually defined in terms of a generic `Lattice` template. Moreover, the `SE` type is actually a type synonym for the more generic `Covert` template:

```cpp
template <typename DataT, SLabel... Labels>
using SE = covert::Covert<SLabel, DataT, Labels...>;
```

while the `SLabel` type specializes the `Lattice` template:

```cpp
template <> struct Lattice<SLabel> {
  static constexpr auto bottom = L;
  static constexpr bool leq(SLabel l1, SLabel l2) {
    return l1 <= l2;
  }
  static constexpr SLabel join(SLabel l1, SLabel l2) {
    return (SLabel)(l1 | l2);
  }
};
```

The members `bottom` ($\bot$), `leq()` ($\leq$), and `join()` ($\sqcup$) are required for all lattices in Covert C++. Formally, these are the basic requirements for a bounded join-semilattice[1].

[1]There are also other algebraic requirements for a bounded join-semilattice, such as closure under $\sqcup$. However, these requirements are not possible to verify with the C++ type system. The developer must ensure that the other lattice requirements are met.

For example, Covert C++ consults the `Lattice` to determine how to label the sum of two `SE` types. Consider the typing of `x + y` where `x` has type `SE<int, L>` and `y` has type `SE<int, H>`. These types actually unfold to

`Covert<SLabel, int, L>` and `Covert<SLabel, int, H>`,

respectively. The Covert C++ overload for the `+` operator will examine the label types to ensure that they match. Assuming that they do match, it will then use the `join()` operator defined by `Lattice<SLabel>` to compute the label for the result. Similarly, `leq()` is used to implement the type conversions. The `bottom` value is used to enforce security constraints. For instance, given some label type `Label`, the FROM SE rule checks that every label in the source type is equal to `Lattice<Label>::bottom`. All of the `se_*()` functions, including `se_label_cast()`, `se_static_cast()`, etc., are also macro defined in terms of generic `covert_*()` functions. The `covert_label_cast()` function serves as Covert C++'s analog to declassification in the decentralized label model.

7.2 Case Study 2: Computing χ^2 with Multiple Parties

This case study demonstrates the application of Covert C++ to the SMPC problem. The problem description and dataset have been adapted from an example given in a Wikipedia article [168] which describes the computation of a χ^2 (chi-squared) statistic for a two-dimensional dataset. The dataset compares category of employment (white collar, blue collar, no collar) against neighborhood. A chi-squared statistic can be used to determine whether there is a significant difference between observed frequencies and expected frequencies in categorical data. The χ^2 statistic is computed as

$$\chi^2 = \sum_i \frac{(obs_i - exp_i)^2}{exp_i}$$

where the exp_i are the expected values for the corresponding observations obs_i.

Suppose that Alice, Bob, Charlie, and Dylan live in four different neighborhoods, A, B, C,

	A	B	C	D	Total
White collar	90	60	104	95	349
Blue collar	30	50	51	20	151
No collar	30	40	45	35	150
Total	150	150	200	150	650

Table 7.1: Neighborhood population by employment category

and D, respectively. Each principal polls the residents of his or her own neighborhood and records the results, shown in Table 7.1. The four principals would like to compute the χ^2 test statistic for the categorical data given in the table. However, Alice, Bob, Charlie and Dylan are mutually distrusting of one another. Each principal would prefer not to share his/her own input data with any other principal, but any principal participating in the computation may be allowed to view the output χ^2 test statistic.

To address this problem with Covert C++, a lattice must be constructed for the four principals. For brevity, Alice is principal A, Bob is B, etc. The lattice is implemented by assigning a bit to each individual. A is bit 0 (`0b0001`), B is bit 1 (`0b0010`), etc. The $\sqcup$ operation is defined as the bitwise OR of labels. Hence when A and B's data is combined, the result belongs to label `0b0011`, a kind of pseudo principal AB representing the combination of A and B. The ordering operator is defined so that

$$x \leq y \iff x \sqcup y = y.$$

The ordering relationship under $\leq$ for the lattice of SMPC labels is depicted in Figure 7.1. P and E are the bottom and top labels of the lattice, respectively. Data in security class P (public) has no restrictions placed on it. Data in security class E is influenced by every participant, and thus is the most restricted.

Similar to `SE`, the `MPC` type is a synonym for `Covert`, but with the SMPC lattice in place of the binary low/high lattice. The input data for each principal is stored in a vector which associates its elements with the principal who owns them:

```
    template<typename T, auto Principal>
```

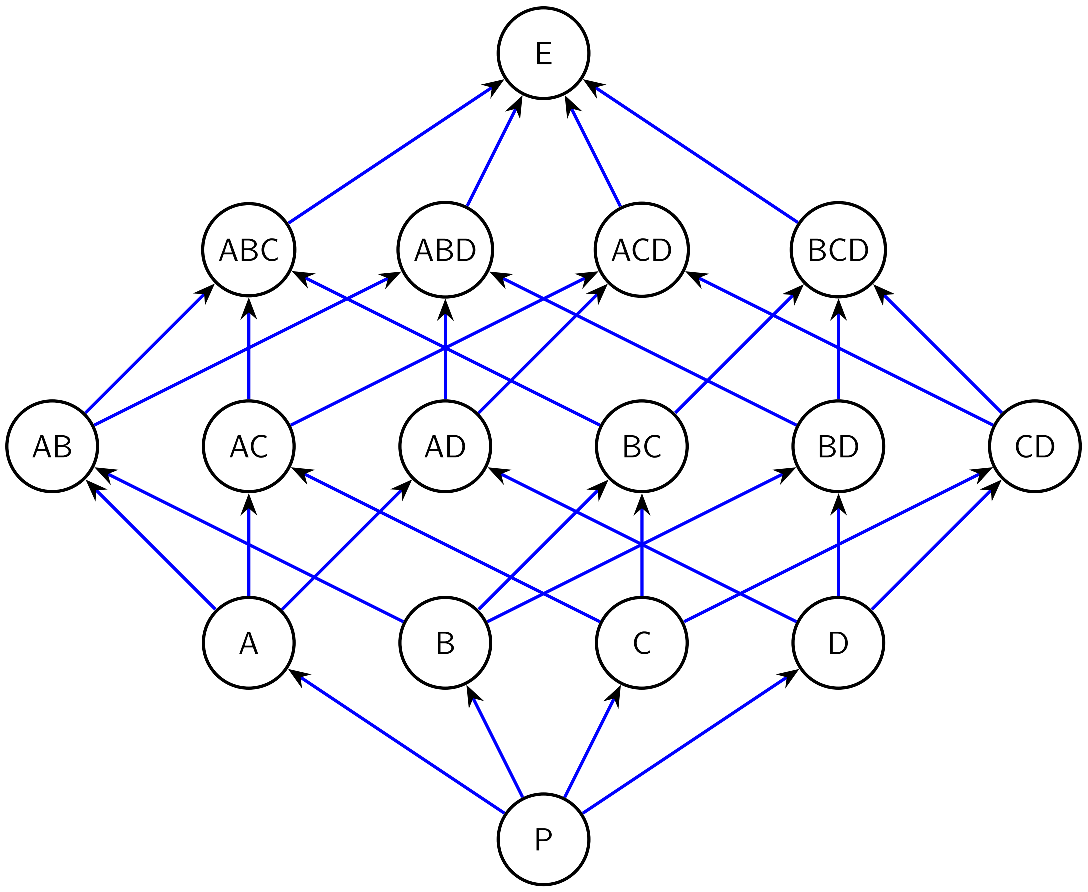

Figure 7.1: A lattice of security classes for SMPC

```
    using PVec = std::vector<MPC<T, Principal>>;
```

For simplicity, assume that the input data can be declared statically:

```
    const PVec<double, A> a = {90.0, 30.0, 30.0};
    const PVec<double, B> b = {60.0, 50.0, 40.0};
    const PVec<double, C> c = {104.0, 51.0, 45.0};
    const PVec<double, D> d = {95.0, 20.0, 35.0};
```

Furthermore, suppose that there is an output function templatized on the type of principal labels, and defined so that `output<P>(x)` outputs the value of `x` to all members of `P`—and only members of `P`.

The implementation of the Covert C++ χ^2 function, `chi2()`, is given in Appendix B. The `chi2()` function is variadic: it can accept any number of arguments (but it requires at least two). The implementation of `chi2()` makes extensive use of type inference, i.e., using the auto keyword. Hence the return type of chi2() is entirely inferred by the compiler from the types of the arguments. That is, the principals who contribute data to the computation. Because the data from each participant is combined (e.g., when the rows of the matrix are summed), the test statistic returned by chi2() will have a label equal to cumulative join of all of the participants.

If Alice, Bob, Charlie and Dylan all participate in the computation, the setup might look like this:

```
void test1() {
  auto result = chi2(a, b, c, d);
  output<E>(mpc_label_cast<P>(result));
}
```

By type inference, result must have type MPC<double, E>. Since the E label is restricted, the result value cannot be directly passed into a storage channel. Instead, it must be explicitly declassified. The mpc_label_cast() function is analogous to se_label_cast(), but for the MPC types. Both label casting functions are defined using the same macro. In test1() it is being used to downgrade the label of result to P (public), so that it can be output to everyone.

What if Bob does not participate in the computation? Then perhaps Bob should not be allowed to receive the output. A naïve implementation would hardcode this constraint, e.g., output<ACD>(...) or output<~B>(...). A more flexible approach would be to use type inference and the covert_traits interface:

```
void test2() {
  auto result = chi2(a, c, d);
  constexpr auto ResultLabel =
      covert_traits<decltype(result)>::label;
  output<ResultLabel>(mpc_label_cast<P>(result));
}
```

The covert_traits template defines a compile-time interface to covert types, similar to iterator_traits for iterators in the C++ STL. In test2() it is used with decltype to obtain the MPC label of the result. By type inference, this label is ACD. Hence the statistic will only be output to those three principals.

Although `test2()` is an improvement over `test1()`, the `output()` function does not perform any kind of validation on its input. For example, the principals might want to require that in order for a value to be output to any principal's channel, that principal must have participated in the computation. The revised `test3()` function introduces a guarded `g_output()` wrapper around `output()`, which accepts an additional boolean template parameter. If the parameter is false, then the compiler's function overloading will not consider it as a candidate, and thus the compiler will emit an error.

```
template <auto Principal, bool Guard, typename T,
          typename = std::enable_if_t<Guard>>
void g_output(const T &val) {
  output<Principal>(mpc_label_cast<P>(val));
}
void test3() {
  auto result = chi2(a, c, d);
  constexpr MPCLabel ResultLabel =
      covert_traits<decltype(result)>::label;
  using Ltc = Lattice<MPCLabel>;
  g_output<A, Ltc::leq(A, ResultLabel)>(result);
  g_output<B, Ltc::leq(B, ResultLabel)>(result); // error!
  g_output<C, Ltc::leq(C, ResultLabel)>(result);
  g_output<D, Ltc::leq(D, ResultLabel)>(result);
}
```

For each principal, the output guard ensures that he or she participated in the computation. Thus for Bob, the guard will fail.

A Note on Security. One benefit of using Covert C++ for SMPC is that the noninterference policy extends to the generalized label model. Although the informal noninterference proof presented in Chapter 5 was applied to the binary priority lattice, none of the details in that proof preclude the generalized label model. Furthermore, the lattice model for Core Covert in Chapter 4 was already generic. With generalized labels corresponding to individual users or groups of users, Covert C++ becomes a full realization of Goguen and Meseguer's original definition [76] of a noninterfering system.

Chapter 8

Covert C++ with Fast Memory-Trace Obliviousness

The greatest drawback of pure Covert C++, as presented in Chapter 5, is that it does not allow high pointers to be used to access (i.e., read/write) memory. This renders high pointers impotent; the sole purpose of pointers in C++ is to provide access to objects located in memory.

This chapter introduces libOblivious, a C/C++ software library to facilitate data-oblivious and memory-trace oblivious computation. In particular, libOblivious defines oblivious iterators, which have syntax and semantics similar to that of pointers and raw C++ iterators, except that memory accesses through oblivious iterators do not leak sensitive information. libOblivious is a stand-alone software library. It can be linked into other C or C++ codebases without Covert C++, though it was designed and implemented primarily for the purpose of facilitating fast memory-trace oblivious computation with Covert C++.

Section 8.1 describes the design and implementation of libOblivious. In particular, Sections 8.1.5 and 8.1.6 discuss the aspects of libOblivious that are novel to this dissertation—they have not been considered in prior works. Section 8.2 presents the performance evaluation for libOblivious. Since Covert C++ iterators and algorithms thinly wrap libOblivious iterators and algorithms (the wrappers are optimized away at compile time), these performance results apply equally to Covert C++. Section 8.3 demonstrates how libOblivious has been integrated into Covert C++. The last section, Section 8.4, demonstrates how libOblivious components can be used in concert to implement a memory-trace oblivious k-nearest neighbors algorithm in Covert C++.

8.1 libOblivious

libOblivious is composed of four components to facilitate memory-trace oblivious programming in C and C++: (1) oblivious primitives, such as oblivious copy and swap operations; (2) oblivious container types, e.g., vectors, linked lists, and deque-like structures; (3) oblivious iterators which can be used to obliviously read/write from/to the oblivious container; and (4) oblivious algorithms. The next four subsections provide an overview of these features in greater detail. The last two subsections describes the implementation details which make libOblivious fast, and also adhere to the conventions of the C++ standard as closely as possible.

8.1.1 Primitives

libOblivious provides four groups of oblivious primitives: oblivious copies, swaps, reads, and writes. An oblivious copy moves data from one of two sources, depending on a Boolean condition, to a single destination. The destination and sources could be any combination of memory locations or CPU registers. Semantically, an oblivious copy is like a ternary (`?:`) operator, except that the value of the Boolean condition does not leak through a side channel. An oblivious swap, like the oblivious copy, swaps the contents of two memory locations (or two registers) if the given Boolean condition is `true`, and it does not leak the value of that condition. An oblivious read/write from/to a region of memory does not leak the address within that region where the access was made. More complex algorithms such as those

C Prototype	Register Map	Implementation

`int64_t o_copy_i64(`
`  int cond,`
`  int64_t left,`
`  int64_t right);`
	`cond` ⇒ `ecx`
	`left` ⇒ `rdx`
	`right` ⇒ `r8`
		`mov rax, rdx`
		`test ecx, -1`
		`cmovz rax, r8`

`void o_swap_i64(`
`  int cond,`
`  int64_t *left,`
`  int64_t *right);`
	`cond` ⇒ `ecx`
	`left` ⇒ `rdx`
	`right` ⇒ `r8`
		`test ecx, -1`
		`mov r10, QWORD PTR [r8]`
		`mov r9, QWORD PTR [rdx]`
		`mov r11, r9`
		`cmovnz r9, r10`
		`cmovnz r10, r11`
		`mov QWORD PTR [rdx], r9`
		`mov QWORD PTR [r8], r10`

`int64_t o_read_i64(`
`  const void *src_base,`
`  size_t src_size,`
`  const int64_t *addr,`
`  bool base_aligned);`
	—	—

`void o_write_i64(`
`  void *dst_base,`
`  size_t dst_size,`
`  int64_t *addr,`
`  int64_t val,`
`  bool base_aligned);`
	—	—

Table 8.1: 64-bit libOblivious primitives

discussed in Section 8.1.4 can be built on top of these primitives.

Table 8.1 gives a summary of the 64-bit versions of these operations. The x86-64 assembly implementations for `o_copy_i64()` and `o_swap_i64()` are given in the table. Since this is the implementation for Windows, it uses Windows x86-64 procedure call conventions [17] and Microsoft macro assembler [16] (MASM) syntax (which itself uses Intel assembly syntax [13]). These operations utilize the x86 `cmov` family of instructions (as described in Chapter 2) to conditionally move data from one register to another—without requiring a branch operation.

The implementations of `o_read_i64()` and `o_write_i64()` are more complex, but both use the `vpgather` instructions in a manner similar to that which was described in Chapter 2. One difference is that 64-bit reads and writes use the `vpgatherqq` instruction, which reads four 64-bit words from four (possibly) non-contiguous addresses in memory. As demonstrated later in Section 8.2, this modification doubles the oblivious read and write throughput.

Each oblivious write actually performs exactly one read and exactly one write per mask because each mask in the given memory region(s) must be accessed the same number of times (and in the same order). This is trivial for the mask that contains the actual destination address for the write. For every other mask, a non-corrupting write must be performed. Hence for each mask that does not contain the destination address, an arbitrary value is read, and then written back (unmodified) to the same location. For the mask that does contain the destination address, the old value is read, and then the new value is written in its place.

For backwards compatibility with older CPUs that do not support AVX2, libOblivious can be configured to perform scalar oblivious reads and writes instead of the vectorized versions. The scalar implementation is up to 2x slower than the vectorized implementation.

The oblivious copy and swap operations are available in 8, 16, 32, 64, and 256-bit versions, along with generic `o_copy()` and `o_swap()` functions which copy or swap an arbitrary number of bytes in a single operation. Oblivious reads and writes are available in 32 and 64-bit versions and also have generic `o_read()` and `o_write()` functions. For non-contiguous data structures such as linked lists and deques, libOblivious also provides `o_read_list()` and `o_write_list()` operations which accept a list of memory regions from which to obliviously read or write data.

When the number of bytes to be copied, swapped, read, or written is large, libOblivious vectorizes these operations, dramatically increasing throughput without violating the property of memory-trace obliviousness. Details on these optimizations are given in Section 8.1.6.

Of the four components of libOblivious, this is the only component which provides an interface for C programs. Moreover, since the C language does not use name mangling, other languages with a C foreign function interface (e.g., Python) can use these libOblivious primitives.

When libOblivious is linked into C++ code, additional generic templatized primitives are exposed: `o_copy_T()`, `o_swap_T()`, `o_read_T()`, `o_write_T()`, `o_read_list_T()`, and `o_write_list_T()`. These are easier to use (and less error-prone) than the pure C APIs, because they automatically deduce the correct number of bytes to copy/swap/read/write from the types of the function arguments.

8.1.2 Containers

libOblivious defines several wrappers for C++ STL containers. The supported containers include:

- Arrays (`std::array`)
- Singly linked lists (`std::forward_list`)
- Doubly linked lists (`std::list`)
- Deques (`std::deque`)
- Queues (`std::queue`)
- Stacks (`std::stack`)
- Vectors (`std::vector`)

The oblivious containers are just type aliases for the STL containers, except that they use stateful heap allocators to maintain a record of which regions of the heap they own. More details on the stateful heap allocators are given in Section 8.1.5.

All of the supported containers are sequential, i.e., containers that can be accessed by sequential iteration or random access. Associative containers such as maps (e.g., red-black trees) and unordered maps (e.g., hash tables) are not supported because these structures use key-value lookup to access container elements in a manner which is not memory trace oblivious.

It is still possible to simulate associative containers with sequence containers. For example, a singly linked list can store a key-value pair in each node (e.g., using the `std::pair` template). Associative lookup can be performed using the `ofind_if()` algorithm (see Section 8.1.4),

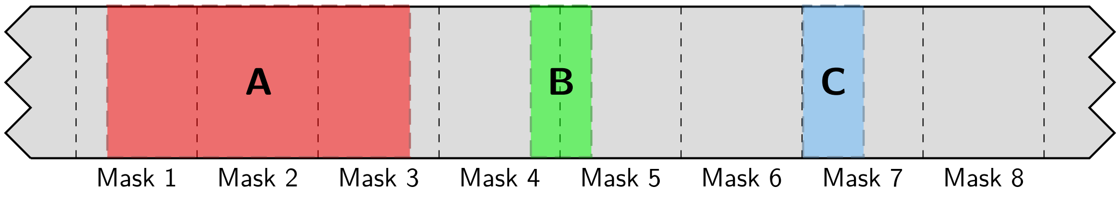

Figure 8.1: Objects in memory covering masks

where the given predicate returns `true` when a node's key matches a given query. Unlike ordinary associative lookup in `std::map` and `std::unordered_map`, this operation has $O(n)$ complexity, as opposed to $O(log(n))$ or $O(1)$ for the STL associative containers.

In fact, $O(n)$ complexity is the best that can be achieved for any search algorithm over an oblivious container, when the search criteria are secret. The following theorem establishes this fact. Some additional notation is required: $\mathcal{I}_{Secret}$ is the set of all possible sequences of secret inputs, and likewise $\mathcal{I}_{Public}$ for public input sequences. The *image* of a function $f : A \to B$ is defined as:

$$f^{\to} = \{f(x) \mid x \in A\},$$

and similarly for any subset $S \subseteq A$:

$$f^{\to}(S) = \{f(x) \mid x \in S\}.$$

In the following theorem, function f serves as the lookup function, mapping program input to the address of a container element. The term *access* in this context refers to any combination any combination of memory reads or writes on an object. An object in memory *covers* one or more masks (recall Definition 2.3) when any portion of the object's contiguous memory lies within those masks. For example, in Figure 8.1 object A covers masks 1, 2, and 3, B covers masks 4 and 5, and C covers mask 7.

Theorem 8.1 (Generalized Oblivious Lookup)**.** *Let C be a container with n elements. Let $f : \mathcal{I}_{Public} \to \mathcal{I}_{Secret} \to C$ be a function mapping program input to locations of elements in C. Let Π be a memory-trace oblivious program which uses f to locate and then access some element in C. Let $I_{Public} \in \mathcal{I}_{Public}$, and let $f_{Secret} = f\ I_{Public}$ (i.e., $f_{Secret} : \mathcal{I}_{Secret} \to C$).*

For all $I_{Secret} \in \mathcal{I}_{Secret}$, if $\mathcal{M}(\Pi, I_{Secret}, I_{Public}) \rightsquigarrow \tau$, then τ must have recorded at least $\Omega(\,|f^{\rightarrow}_{Secret}|\,)$ accesses on the masks covered by elements of C.

Proof. Let $\{e_1, \ldots, e_m\} = f^{\rightarrow}_{Secret}$, where $m = |f^{\rightarrow}_{Secret}|$. Partition $\mathcal{I}_{Secret}$ into $\mathcal{I}^1_{Secret}, \ldots, \mathcal{I}^m_{Secret}$ such that for all i from 1 to m, $f^{\rightarrow}_{Secret}(\mathcal{I}^i_{Secret}) = \{e_i\}$. Let $I_{Secret}, I'_{Secret} \in \mathcal{I}_{Secret}$ be arbitrary, and note that there exist $i, j \in \{1, \ldots, m\}$ such that $I_{Secret} \in \mathcal{I}^i_{Secret}$ and $I'_{Secret} \in \mathcal{I}^j_{Secret}$. Suppose

$$\mathcal{M}(\Pi, I_{Secret}, I_{Public}) \rightsquigarrow \tau \text{ and } \mathcal{M}(\Pi, I'_{Secret}, I_{Public}) \rightsquigarrow \tau'.$$

Since Π is memory-trace oblivious, $\tau \approx_{Adv} \tau'$. Let n be the granularity of $\approx_{Adv}$, let s be the size of each element in C (container elements must be of uniform size in C++). If $i = j$, then τ and τ' both recorded accesses on at least $\left\lceil \frac{s}{2^n} \right\rceil$ masks covered by e_i. If $i \neq j$, then τ and τ' both recorded accesses on at least $\left\lceil \frac{2 \cdot s}{2^n} \right\rceil$ masks covered by e_i and e_j.

In general, the traces emitted by Π over secret input sequences from k distinct subsets of the partition $\mathcal{I}^1_{Secret}, \ldots, \mathcal{I}^m_{Secret}$ must record accesses on at least $\left\lceil \frac{k \cdot s}{2^n} \right\rceil$ masks covered by k elements of C. Since any input in any of $\mathcal{I}^1_{Secret}, \ldots, \mathcal{I}^m_{Secret}$ can be given, each trace must record at least

$$\begin{aligned} \left\lceil \frac{m \cdot s}{2^n} \right\rceil &= \left\lceil \frac{|f^{\rightarrow}_{Secret}| \cdot s}{2^n} \right\rceil \\ &\in \Omega(\,|f^{\rightarrow}_{Secret}|\,) \end{aligned}$$

accesses on the masks covered by elements of C. □

Example 8.1. Consider the algorithm `ofind()`, an oblivious implementation of `std::find()`. The `ofind()` function takes as parameters two iterators into a container which define the range over which to search, and the value to find; it returns an iterator I to the first element in the container which matches the search query. Suppose that the search query value is secret, and that the iterators defining the range of the search cover only the first half of a vector of size n. If the caller then uses I to access the element that was looked up, Theorem 8.1 requires that $\Omega(n/2) = \Omega(n)$ additional elements in C must be accessed to maintain memory-trace obliviousness, and thus not leak the value of the secret query. △

Example 8.2. Consider the function `std::list::begin()`, which returns an iterator I to the first element of a given list. `std::list::begin()` is actually a degenerate lookup function whose image is a singleton set consisting of the first element of the list. By Theorem 8.1, if a caller uses I to access the first list element, then $\Omega(1)$ accesses must be performed. In fact, only the one access must be performed, hence `std::list::begin()` is invariably memory trace oblivious. △

Corollary 8.2 (Oblivious Lookup)**.** *Let C be a container with n elements. Let $f : \mathcal{I}_{Secret} \rightarrow C$ be a surjective function mapping program input to locations of elements in C. Let Π be a memory-trace oblivious program which uses f to locate and then access some element in C, and let $I_{Public} \in \mathcal{I}_{Public}$. For all $I_{Secret} \in \mathcal{I}_{Secret}$, if $\mathcal{M}(\Pi, I_{Secret}, I_{Public}) \rightsquigarrow \tau$, then τ must have recorded at least $\Omega(n)$ accesses on the masks covered by elements of C.*

Proof. This is simply a special case of Theorem 8.1 with $f^{\rightarrow}_{Secret} = C$. □

Example 8.3 (Oblivious Read)**.** Consider the primitive subscript operation on an array, e.g., `arr[42]`. Assuming that out-of-bounds accesses are disallowed, the subscript operation is a lookup operation which maps an integral index to an array element, returning a reference to that element. If the array has n elements, then by Corollary 8.2 $\Omega(n)$ accesses are required to access the returned reference, assuming the index is secret. △

On a particular concrete machine and with a particular adversary model, the argument in Example 8.3 can be made more precise. Recall the discussion about oblivious reads and writes from Chapter 2. For an adversary whose observational power $\sim_{Adv}$ has cache block granularity on x86-64, the strategy was to read one element (e.g., a 32-bit integer) from each cache block to obfuscate the access at a secret address or index. Recall that the proof of Theorem 8.1 constructed a precise lower bound on the number of accesses required for a memory-trace oblivious lookup: $\left\lceil \frac{n \cdot s}{g} \right\rceil$ when the lookup function's image is the entire container holding n elements. Thus to defeat $\sim_{Adv}$, a 32-bit oblivious read on an array of size n must make at least $\left\lceil \frac{n \cdot 4}{64} \right\rceil = \left\lceil \frac{n}{16} \right\rceil$ accesses. The oblivious memory access strategy employed by libOblivious for `o_read_32()` and `o_write_32()` performs precisely this many accesses for any array of size n.

8.1.3 Iterators

When operating on elements in a container, C++ conventions suggest the use of iterators, rather than pointers [154]. An iterator is typically implemented as a wrapper around a pointer, but with semantics more appropriate for its corresponding container. For instance, an iterator to a linked list may overload the `++` operator so that instead of incrementing the underlying pointer, the pointer is advanced to the next link in the list.

All iterators must define the indirection (`*`) operator, which is used to access the container element to which the iterator points [90]. Random access iterators, which operator over sequential containers that support random access operations, also provide a subscript (`[]`) operator. Random access iterators support all of the arithmetic operations (addition, pointer difference, etc.) that are supported by ordinary pointers.

libOblivious provides its own iterators, which are themselves wrappers around C++ STL iterators or ordinary pointers. The `O` template wraps an iterator or a pointer, and endows it with memory-trace oblivious read and write operations to the container into which the iterator points. The `O` template also must wrap a reference to the container, effectively binding each `O` iterator to a specific container. The reason for this design decision is discussed later in Section 8.1.5.

Listing 8.1 shows an example which uses an `O` iterator to obliviously read from a linked list element whose address was discovered by an oblivious find (`ofind_if()` is described in the next section). Failure to obliviously read from the iterator `i` could leak the value of `x` and/or the values of elements in `list`.

In general, `O` iterators conform to the C++ standard requirements for iterators, though there are two noteworthy exceptions. First, section 27.2.5 the C++17 standard states that

```
// return the first value less than ‘x‘ in ‘list‘, or -1 if
// there is no such value
int find_less_than(const olist<int> &list, int x) {
  auto finder = [x](int val) { return val < x; };
  O i = ofind_if(list.begin(), list.end(), finder, &list);
  if (i == list.end()) // value less than ‘x‘ not found
    return -1;
  else // return the value that was found
    return *i; // performs an oblivious read
}
```

Listing 8.1: Using an `O` iterator to obliviously read an `int`

the `reference` type member of a forward iterator (an input iterator that can be used in multipass algorithms) must be a reference to the value type of the corresponding container [90]. When a forward iterator is dereferenced using the `*` operator, the return type is `reference`. The `O` template uses the temporary proxy idiom [12], a topic covered in greater detail in Section 8.1.5. One feature of the temporary proxy idiom is that it uses a proxy object—rather than a reference—to access container elements. So the `*` and `[]` operators for `O` iterators do not conform to the `reference` type requirement for forward iterators.

Second, section 27.2.3 of the C++17 standard requires that an iterator `a` which satisfies the requirements for an input iterator (an iterator that can read from the pointed-to element), must define the `a->m` operation with semantics `(*a).m` [90]. The `O` template does not support the `->` operator.

8.1.4 Algorithms

The goal of the libOblivious algorithms library is to provide memory-trace oblivious implementations of all algorithms supported by the C++ STL algorithms library [4, 90]. These algorithms are built on top of the memory-trace oblivious primitives that were introduced in Section 8.1.1. Algorithms which use oblivious iterators (see: the `O` template, Section 8.1.3) can only be applied over the libOblivious containers (Section 8.1.2). All other oblivious algorithms can be applied to any sequential container in the C++ STL.

```
template <class InputIt, class UnaryPredicate,
          class ContainerT>
O<InputIt, ContainerT>
ofind_if(InputIt first, InputIt last,
    UnaryPredicate p, ContainerT *container) {
  InputIt ret = last;
  for (; first != last; ++first) {
    o_copy_T(ret, p(*first) & (ret == last), first, ret);
  }
  return {ret, container};
}
```

Listing 8.2: `ofind_if()` implementation

Listing 8.2 shows the implementation of `ofind_if()`, one of the foundational search algorithms of the libOblivious algorithms library. Its semantics correspond to those of `std::find_if()` [90]. The interface is identical to that of `std::find_if()`, except that `ofind_if()` also requires a reference to the container to be searched. This is necessary to construct the oblivious iterator return value, as discussed in Section 8.1.3. `o_copy_T()` is used to obliviously update the return value when the first match is found. Also noteworthy is the use of a bitwise AND (`&`) instead of the logical AND (`&&`) in the copy condition. This subtle distinction is important because the logical AND in C and C++ has short-circuiting behavior which may or may not compile into a branch: the second operand will only be evaluated if the first operand evaluates to `true`. Thus, if the first operand has been influenced by secret information, that information could be leaked. Finally, notice that the `for` loop does not break when a match is found. The algorithm always examines every single element in the container exactly once. Other algorithms such as `ofind()` and `oany_of()` are implemented on top of `ofind_if()`. Table 8.2 summarizes several of the algorithms exported by libOblivious.

8.1.5 Implementation

This section discusses the implementation of libOblivious in much more detail, especially the innovations that have not been presented in prior works.

Name	Description	Complexity
`osort`	Sort a container which support random access operations.	$O(n^2)$
`omax_element`	Find the greatest element in a container, and return an oblivious iterator to it. If several elements are equal to the greatest element, return an oblivious iterator to the first such element.	$O(n)$
`ofind_if`	Searches for an element in a container for which a given predicate P is valid. Return an oblivious iterator to the first such element found.	$O(n)$
`ofind`	Searches for an element in a container which is equal to a given element. Return an oblivious iterator to the first such element found in the container.	$O(n)$

Table 8.2: Some libOblivious algorithms

The libOblivious Heap Allocators. The libOblivious primitives are adequate for facilitating memory-trace oblivious computation on their own. However, the primitive APIs are neither elegant nor easy to use. The `o_write()` API, for example, has 7 parameters. libOblivious also exports oblivious iterators to make oblivious operations more accessible to the developer.

Many aspects of the oblivious iterator `O` template were non-trivial to implement, but none more so than the following. Since an iterator is really just a pointer to a container element, how can an oblivious iterator determine the entire range of memory covered by the container? On any given oblivious read or write, the oblivious iterator will need to know this range so that it can access each mask covered by the range. To make matters more difficult, a non-contiguous container such as a linked list may cover many non-contiguous memory regions. The oblivious iterator will somehow need to determine the base address and size of each region covered by the non-contiguous container. The proposed solution to this problem exploits a seldom-used feature of C++ [117]: user-defined heap allocators.

The C++ standard library has a category of containers called *allocator-aware containers* [90]. That is, containers which use a heap allocator to store values. Examples of allocator-aware containers include linked lists, maps, and vectors, but not statically-allocated or stack-allocated arrays. For any C++ STL allocator-aware container, C++ allows the default allocator to be substituted by a user-defined allocator, assuming the user-defined allocator conforms to the C++ standard's interface and behavioral requirements for a heap allocator [90].

The strategy adopted by libOblivious is to define an allocator which maintains bookkeeping information for all of the memory currently owned by its parent container. When an oblivious iterator performs a read or a write on a container, it queries the oblivious container's allocator to determine which memory region(s) must be accessed. Due to an unfortunate historical detail of C++, the implementation cannot be this simple.

The requirements for user-defined allocators changed substantially from C++03 to C++11 [85, 154]. In particular, prior to C++11 all allocators were required to be stateless. C++11 relaxed this requirement. However, the need to maintain backwards compatibility meant that the new interface requirements for stateful user-defined allocators could not be defined cleanly [117]. For example, allocator-aware containers have always been required to provide a `get_allocator()` member function which returns a copy-constructed replica of that container's internal heap allocator [90]. Because copy operations are often expensive, C++ accessor methods typically return a reference to the internal member. Since user-defined allocators were previously required to be stateless, copy operations were free; thus the return-by-value semantics of `get_allocator()` was justifiable.

Unfortunately, the `get_allocator()` function is the only member function exposed by C++ STL containers which allows access to the container's internal allocator. The return-by-value semantics poses two problems for the design. First, the heap memory ownership bookkeeping is maintained in a data structure which can potentially become quite large. Copying the entire structure on each oblivious read or write would be extraordinarily inefficient. Second, it may be possible for a container to allocate new memory after an oblivious iterator obtains a copy of the allocator, but before the oblivious iterator uses the copy to perform an oblivious read or write. In this scenario, the oblivious iterator would use stale information to determine which regions it should access.

A perfect solution to this problem could not be found. The compromise solution was to implement the heap allocator's internal state as a shared pointer [90, 154] (i.e., a reference-counted pointer) to the linked list containing the bookkeeping information. Thus when the oblivious iterator calls `get_allocator()` on the oblivious container to which it is bound, it receives a copy of the shared pointer, through which it can access the (fresh) bookkeeping information. The implementation takes care to ensure that the lifetime of the shared pointer copy does not extend beyond the current read or write operation, so as to not prolong the lifetime of the potentially large bookkeeping list after its parent container is destroyed.

The libOblivious heap allocator also strategically allocates memory to improve performance of oblivious reads and writes. Suppose that the underlying libc `malloc()` routine on a particular platform tends to scatter each successive heap allocation request across memory. In the worst case, consider a linked list where each node occupies 16 bytes, and each call to `malloc()` returns 16 bytes located in a mask not already occupied by any other node allocated in this list so far. If the linked list has n nodes, then an oblivious read will need to access all n nodes to access every mask covered by the container. However, if the nodes have all been allocated contiguously, then only $\left\lceil \frac{16 \cdot n}{mask\ size} \right\rceil$ nodes will need to be accessed. For an adversary who has observational power at cache block granularity, this optimization reduces the work load by as much as a factor of 4.

The libOblivious heap allocator has several implementations which are optimized for various data structures. The characteristics of each implementation are not particularly interesting. However, all implementations do share in common at least two attributes. First, they try to contiguously allocate the container elements as much as possible, with minimal internal and external fragmentation. Second, all memory regions covered by the allocators are guaranteed to be at least mask aligned.

The O Template. The `O` template wraps an iterator and a reference to the container into which the iterator points. As mentioned earlier, an oblivious iterator (an instance of the `O` template) behaves similarly to the iterator which it wraps, but with several important exceptions. The primary purpose of the oblivious iterator is to modify the read and write semantics of C++, thus replacing conventional memory accesses on containers with memory-trace oblivious accesses.

If `p` is a pointer into an array with element type `T`, then the default behavior in C++ for the indirection operation `*` is to return a reference to the pointed-to element (i.e., `T &`), and similarly for the subscript operator `[]`. Thus these operations behave predictably when used in expressions. The expression `T x = *p;` will assign the value of the reference returned by `*p` to the new variable `x`. Similarly, `*p = x;` will assign the value of the variable `x` to the object referenced by `*p`, assuming that `p` is a non-`const` pointer: invoking the indirection operator on a `const` pointer will return a `const` (immutable) reference.

The indirection and subscript operations can be overloaded in C++ [90, 154], which is also how these operations were implemented for pure Covert C++. The overload will typically behave similarly to the default operators: a dereference operation on an iterator will return a reference to the pointed-to element in the container. However the C++ standard does not mandate this as a requirement for all classes.

The temporary proxy idiom [12] exploits the flexibility of the indirection and subscript overloads to allow the developer to alter the manner in which container elements are accessed through an iterator. The basic mechanics of the idiom are as follows. Instead of returning a reference to the pointed-to element, the indirection/subscript operator can return a proxy object with one or both of the following operators overloaded:

```
operator T() { ... }
void operator=(const T &) { ... }
```

The first operator is the user-defined conversion operator [90, 154]. For a given class `U`, a user-defined conversion operator can specify either an implicit or explicit conversion to any other type `T`. Without the `explicit` keyword in the declaration, the conversion is assumed to be implicit. The second operator defines assignment. For example, when an object of the given class is assigned (via `=`) a value of type `T`, this operator will be called to perform the assignment.

The temporary proxy idiom uses the user-defined conversion operator to “read” from the pointed-to element, and it likewise uses the assignment operator to “write” to the pointed-to element. The process works as follows. Suppose that `i` is an iterator which points to elements of type `T`, and `i`’s class uses the temporary proxy idiom. In the statement `T x = *i;`, The indirection `*i` will construct and return a proxy object which, at minimum, knows the address of the pointed-to element. The proxy object and `x` do not share the same type, hence the compiler will search for a valid conversion sequence from the type of the proxy object to `T`. The proxy object’s user-defined conversion `operator T()` satisfies this requirement, and thus is selected to perform the conversion. The operator can simply return a copy of the pointed-to element, or it can do something more interesting.

The process for a write is similar. The statement `*i = x;` will invoke the proxy object’s assignment operator for a parameter of type `T`, thus allowing the user to customize the write behavior of the iterator. Note that the lifetime of the proxy object should not extend beyond the evaluation of the statement, which is why the idiom is called *temporary* proxy.

The oblivious read/write mechanism of the `O` template is implemented using the temporary proxy idiom. If `o` is an oblivious iterator, then the statement `T x = *o;` is evaluated as follows:

1. The evaluation of `*o` returns the proxy *oblivious accessor* object, which essentially wraps a reference to `o`.
2. Overload resolution for a conversion sequence from the type of the oblivious accessor to `T` selects the oblivious accessor’s implicit conversion operator, which performs the following sequence of operations:
    (a) Call `get_allocator()` on `o`’s container reference to retrieve a shared pointer to the container’s oblivious allocator.
    (b) Call `get_regions()` on the oblivious allocator, which returns a list of ⟨base address, size⟩ pairs of all heap memory regions covered by the container.
    (c) Call `o_read_list_T()` to obliviously read the value of type `T` pointed to by `o`

from the given list of regions.

3. The value returned by this last call is assigned to `x`.

The process is similar for an oblivious write through an oblivious iterator, except that the assignment operator on the oblivious accessor is called, and `o_write_list_T()` is used in step 2(c) above.

Unfortunately, the temporary proxy object does not always behave as expected. If a variable is declared with the C++ `auto` keyword in place of a specific type, as in `auto x = *o;`, then the C++ type inference rules will infer the type of `x` to be the type of the value being assigned to `x`—in this case, the proxy object itself. Hence instead of triggering a call to the proxy object's conversion operator, the compiler will use the proxy object's copy constructor to instantiate `x` as a copy of the proxy object. In certain situations, this may actually be what the developer would want. For example, the copy of the proxy object can act like a reference to the pointed-to element in the container, allowing repeated oblivious accesses to it, but it is more likely that the copy would be created accidentally.

One way to prevent these accidental copies would be to delete the proxy object's copy constructor. However, as of C++17 this fix is no longer workable due to guaranteed copy elision [65, 90].

A similar consequence of the temporary proxy idiom arises when calling template functions. Given a template function `foo()` with prototype

```
template <typename T> void foo(const T &arg);
```

the function call `foo(*o);` (where `o` is an oblivious iterator) will instantiate the template parameter `T` as the type of the proxy object. In this case, the lifetime of the temporary proxy object will be extended, and `arg` will be a `const` reference to the proxy object. Again, this is most likely not the behavior that the user would intend or expect.

It would be inconvenient for the user if he were always required to specify the element type when reading from an oblivious iterator. Therefore the oblivious iterator and the oblivious accessor both expose a member type `value_type`, which is the type of the elements in the container. For instance,

```
typename decltype(o)::value_type x = *o;
typename decltype(*o)::value_type x = *o;
```

The first statement uses the oblivious iterator's `value_type`. The second statement uses the oblivious accessor's `value_type`. Both statements are equivalent.

One other substantial limitation of the template proxy idiom is that it does not provide any strategy for similarly modifying the behavior of the member access (`->`) operator. The member access operator is typically used to implement wrappers around pointers, such as iterators and smart pointers. However, it is less customizable than the dereference and subscript operators. According to section 16.5.6 of the C++ standard, "An expression `x->m` is interpreted as `(x.operator->())->m` for a class object `x` of type `T` if `T::operator->()` exists and if the operator is selected as the best match function by the overload resolution mechanism" [90]. Hence any overload of the `->` operator must either return a raw pointer or another object with an overloaded `->` operator. There is no way to return a proxy object without inviting an infinite recursion.

The workaround is to first obliviously read an object from a container (i.e., create a copy of it on the stack) and then use the `x.m` member access to read a member.

Shallow versus Deep Copying. Most object-oriented or imperative programming languages support some kind of mechanism to copy an object from one location in memory to another. For instance, consider this definition of a string type in C:

```
struct MyString {
  size_t size;
  char *data;
};
```

If `s1` is a `MyString`, then the statement `MyString s2 = s1;` copies the value of `s1` to the memory occupied by `s2`. Note that the string pointed to by `s1.data` is not actually copied by this operation. Only the `s1.size` member and the `s1.data` pointer itself will be copied, 16 bytes in total on a 64-bit platform. Hence, after the copy `s1` and `s2` will share the same state: any modification to `s1`'s string will be visible through `s2`, and vice-versa.

In C++, this kind of copy is commonly referred to as a *shallow copy* [154]. Unlike C, C++ provides support for making *deep copies*, which completely copy the state of the source object. Deep copies are supported via overloading the copy constructor. For instance, a deep copy constructor for `MyString` might look like:

```
MyString(const MyString &other)
    : size(other.size), data(new char[size]) {
  memcpy(data, other.data, size);
}
```

This example creates a copy of a `MyString` by first copying its `size` member, and then instantiating its own `data` member with a pointer to an array of freshly allocated `char`s on the heap. Then it copies the string state to the fresh heap memory.

libOblivious primitives are only able to make shallow copies of objects. A deep copy requires knowledge of the meaning of an object, and what exactly constitutes its deeper state. Only the author of the `MyString` structure would know that the `size` member should refer to the number of characters pointed to by the data member; an algorithm cannot simply infer this. Hence all of the libOblivious C++ primitives statically check that the types of the arguments are all trivially copyable, meaning that they use the default copy constructor. That is, only shallow copies of the object can be made using copy semantics [90].

It is possible to support deep copies with libOblivious, though this would require a lot of extra work on account of the developer. One solution could be to support a static interface for obliviously (deep) copyable objects, for instance:

```
template <typename T> struct ODeepCopy {
  static T o_deep_copy(bool c, const T &left, const T
     &right);
};
```

If the developer wants one of his classes to be obliviously deep copyable, he can specialize the `ODeepCopy` struct for that class and define an appropriate `o_deep_copy()` function, recursively invoking `o_copy()`s or `o_deep_copy()`s as needed. The libOblivious C++ primitives could first check whether the class is trivially copyable and, if not, check whether the class specializes `ODeepCopy`. If so, the primitives would call the user-defined `o_deep_copy()` instead of `o_copy()`.

8.1.6 Optimizations

This section discusses a series of observations about the execution of libOblivious primitives, which highlight opportunities to improve their performance. These particular optimizations for data-oblivious or memory-trace oblivious programs have not been reported in related works. Section 8.2 demonstrates the observed performance benefits of some of these optimizations.

The first observation pertains to the `o_copy()` and `o_swap()` primitives, both of which move chunks of contiguous data in memory.

Observation 1. Loops which operate sequentially on pairs or tuples of data can often be optimized into parallel vectorized operations using single instruction, multiple data (SIMD) instructions.

The use of multimedia SIMD instructions in commercially available desktop, server, and laptop CPUs to achieve superword level parallelism (SLP) optimizations was first postulated by Larson and Amarisinghe [100]. This strategy has since been adopted by numerous optimizing compilers such as GCC and Clang.

Copy and swap operations sequentially operate on tuples of data in a manner which can be vectorized. However, since these operations are written in assembly (to prevent the compiler from using optimizations that would violate obliviousness) this also effectively disables all SLP optimizations. The solution is to manually vectorize the oblivious copy and swap operations when the number of bytes to copy or swap is sufficiently large enough to fill an AVX2 vector, specifically, 256 bytes [58]. The code which vectorizes memory-trace oblivious copy in libOblivious for 256 contiguous bytes is given in Listing 8.3.

The implementation of `o_copy_i256()` uses Intel's Streaming SIMD Extensions (SSE) compiler intrinsics [39], which are available on popular compilers such as GCC, Clang, and

```
void o_copy_i256(__m256i *dst, int cond,
                 const __m256i *left, const __m256i *right,
                 size_t offset) {
  const __m256i mask = _mm256_set1_epi32(!!cond - 1);
  const __m256i ltmp = _mm256_loadu_si256(left + offset);
  const __m256i rtmp = _mm256_loadu_si256(right + offset);
  const __m256i result =
      _mm256_blendv_epi8(ltmp, rtmp, mask);
  _mm256_storeu_si256(dst + offset, result);
}
```

Listing 8.3: `o_copy_i256()` implementation uses `vpblendvb`

MSVC for supported platforms. For example, the `_mm256_loadu_si256()` intrinsic compiles to an appropriately sized `vmov` instruction. The `o_copy_i256()` function unconditionally loads 256 bytes from both the `left` and `right` pointer operands. It then uses a vector blend operation—a kind of piece-wise ternary operation for vectors—to store either the `ltmp` or `rtmp` vector into the `result` vector, which is then written to the destination `dst` in memory.

The generic `o_copy()` and `o_swap` operations dynamically determine the vectorization depending on the number of bytes requested for copy or swap at runtime. The C++-only `o_copy_T()` and `o_swap_T()` operations are able to infer the number of bytes to copy/swap at compile time, since the number of bytes is the size of the object(s) to be copied or swapped. Thus the vectorization can be inlined, and possibly loop-unrolled, hence achieving even better performance without losing memory-trace obliviousness.

This optimization allows the memory-trace oblivious copy and swap operations to perform comparatively well against the analogous primitives in C++, as demonstrated later in Section 8.2. In fact, libstdc++ and libc++ both do not vectorize the `std::swap()` operation. Hence the oblivious `oswap()` operation is actually faster than `std::swap()` for larger objects.

Observation 2. When obliviously reading or writing an amount of data greater than or equal to the size of the mask, the read/write can be performed faster by an oblivious copy.

Section 8.1.1 described the basic approach for obliviously reading data from a memory region. When reading 4 bytes, the `vpgatherdd` instruction is used, and `vpgatherqq` is used

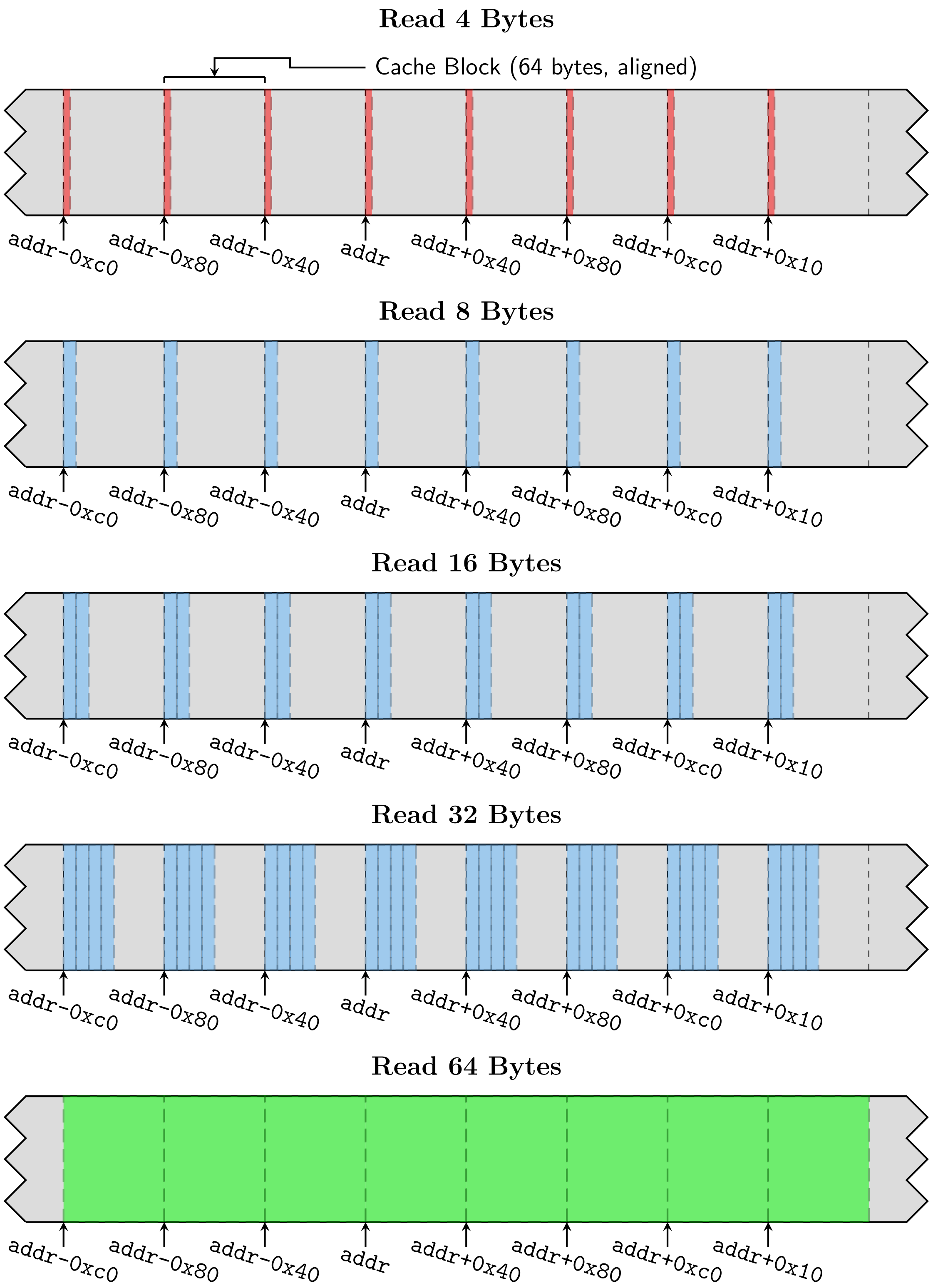

Figure 8.2: Obliviously reading n bytes from an array

to read 8 bytes, which increases throughput. To read more (contiguous) bytes, this process is repeated, reading 4 or 8 bytes at a time. Figure 8.2 illustrates this kind of memory striping pattern to read data. Reads shown in red use `vpgatherdd`, and reads in blue use `vpgatherqq`.

The striped reads offer reasonable throughput because they allow data to be read obliviously, without having to read from every single address within a memory region. As soon as the size of the read reaches the size of the mask, it becomes necessary to read from every address—otherwise the accesses on cache blocks would not be uniformly distributed. In this case, a more efficient strategy can be adopted.

In the last diagram in Figure 8.2, each green box indicates an oblivious copy. Thus, when an oblivious read is requested for n bytes where $n \geq \textit{mask size}$, the `o_read()` primitive iterates over the memory region in n-sized chunks, invoking `o_copy()` for n bytes on each chunk. The condition for `o_copy()` is false for every chunk except for the chunk whose beginning address matches the given `addr` parameter. Hence only the desired value is actually written to the destination. This optimization causes the throughput of `o_read()` to scale linearly with the size of the memory region.

Observation 3. If the memory region being read/written from/to fits entirely within a mask, then an oblivious read/write is not necessary.

This observation may sound trivial, but it can be used to improve performance in certain scenarios. For instance, the oblivious k-NN implementation in Section 8.4 uses an oblivious iterator to perform oblivious accesses on a direct address table containing the tally for each class (i.e., category for classification). If the number of classes is sufficiently small—specifically, less than 16—then the entire data structure may fit within the mask. The oblivious containers guarantee that this is the case, because all of the memory they allocate for data is at least aligned to the mask.

The oblivious read and write primitives inline a check to determine whether the target memory region is entirely contained within a mask. If so, the given address is simply read/written from/to through the pointer. Otherwise, a call is made into the libOblivious

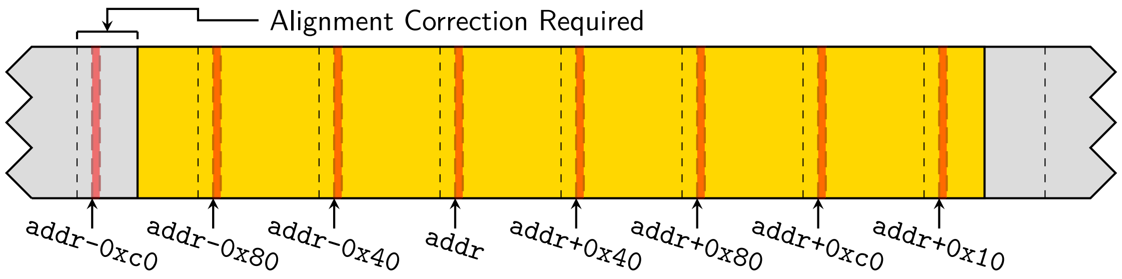

Figure 8.3: A correction is required when the memory region is not aligned to the mask.

DLL to perform an oblivious read or write. For a single operation, this optimization may make very little difference. In the k-NN example, where potentially many random accesses are performed in a loop, eliding repeated calls into a DLL can achieve a noticeable performance improvement.

The last optimization also pertains to alignment. The discussion thus far has not considered the case where either the beginning or the beginning or the end of a memory region is not aligned to a mask. Consider the example in Figure 8.3. The memory region from which to obliviously read (shaded in orange) is not aligned to the mask at either boundary. To ensure memory-trace obliviousness, these boundary masks must always be accessed.

The easiest solution is to treat the memory region as though it extends all the way to the beginning and end of all of the masks it covers. Computationally, this is as simple as computing the alignment correction at the beginning of the region (depicted in Figure 8.3), subtracting the correction from the base of the region, and adding the correction to the end of the region. The correction at the end of the region is computed in a similar fashion, and then added to the memory region's size.

Observation 4. If it is known at compile time that a memory region will be aligned to the mask, then the alignment check and adjustment can be elided at run time.

All of the oblivious read and write primitives have a `bool` parameter called `is_aligned`, which asserts that the given memory region will be aligned to the mask. The alignment check and adjustment described above are both inlined. Hence if an API call asserts `true` for `is_aligned`, the check will be entirely optimized away by the compiler. This is precisely what the `O` template does when it is instantiated for an oblivious container, because libOblivious containers always allocate memory with mask alignment.

8.2 Evaluation

This section describes a series of experiments to test the performance of libOblivious's primitives and algorithms. The results demonstrate that the libOblivious read and write primitives offer a substantial performance increase over the naïve solution described in Chapter 2. Other oblivious primitives compare favorably to their C++ and C++ STL counterparts, and in one case an oblivious primitive outperforms its STL counterpart. The algorithms generally perform well in comparison to the C++ STL, with some exceptions.

8.2.1 Test Setup

The tests were performed on a laptop machine with an off-the-shelf dual-core Intel SkyLake CPU, with a base frequency of 2.9 GHz. The CPU has a 32 KB L1 data cache, a 256 KB L2 cache (per core), and a 4 MB L3 cache (shared). It has 16 GB of 2133 MHz LPDDR3 RAM. All tests were performed on the host operating system, macOS 10.13. Individual tests were compiled by GCC version 8.1.0. All code was compiled and linked with libstdc++.

Before running each test, each region of memory required by the test is allocated and locked (i.e., using `mlock()`), to preclude paging. Each test is run 100,000 times for each value of each of the test parameter(s) (e.g., the size of the data structure being operated on). The reported result is an average of the 100,000 iterations. One exception to this rule is the sorting test, which is run for only 1,000 iterations. The result of the first iteration for each test is always discarded because the instruction and data caches are not yet hot, thus the first iteration is always slower.

For tests which require input parameter(s), such as an index from which to read or write, or a value to find in a container, the parameters are randomized for each test iteration.

8.2.2 Primitive Results

The four categories of libOblivious primitives are reads, writes, swaps, and ternary-like copies. For each primitive, its performance (in terms of execution time) is compared against the size of the data and/or the memory region over which the primitive is being applied.

Figure 8.4 shows the performance results for the libOblivious primitives. The top row of plots shows the performance of the `vpgatherdd`-based oblivious read and write primitives, `o_read_i32()` and `o_write_i32()`, respectively. Both scalar and vectorized versions of `o_read_i32()` substantially outperform the naïve solution. In general, the scalar solution offers a roughly 10-14x performance improvement. The vectorized solution performs best when applied to data already in the CPU's L1 and L2 caches. Beyond the L2 cache, performance gains against the scalar solution gradually taper off. In particular, the best improvements over the naïve solution were 32x when operating on data within the L1 cache, 21x on data in the L2 cache, and 11x within the L3 cache. This last result is slightly faster than the observation for the scalar oblivious read.

The best performance improvement that could have been achieved over the naïve solution with the vector or scalar solutions would have been 16x. This is because the vector and scalar solutions read just 4 bytes for every 64 bytes reads by the naïve solution. Yet beyond the L2 cache, the boost bottoms out at 10-11x. This discrepancy can most likely be attributed to cache misses. When the array is larger than 256 KB, the first memory access in a cache block is always more expensive than each subsequent access, because the first access is always an L2 cache miss. Hence for the naïve solution, each access in a cache block after the first is cheaper. For the scalar and vector solutions, every access should be an (expensive) L2 cache miss. Hence the 16x improvement beyond the L2 cache is almost certainly unattainable. The observed 10-11x improvement should be close to optimal.

The results for `o_write_i32()` are less favorable. As mentioned earlier in Section 8.1.5,

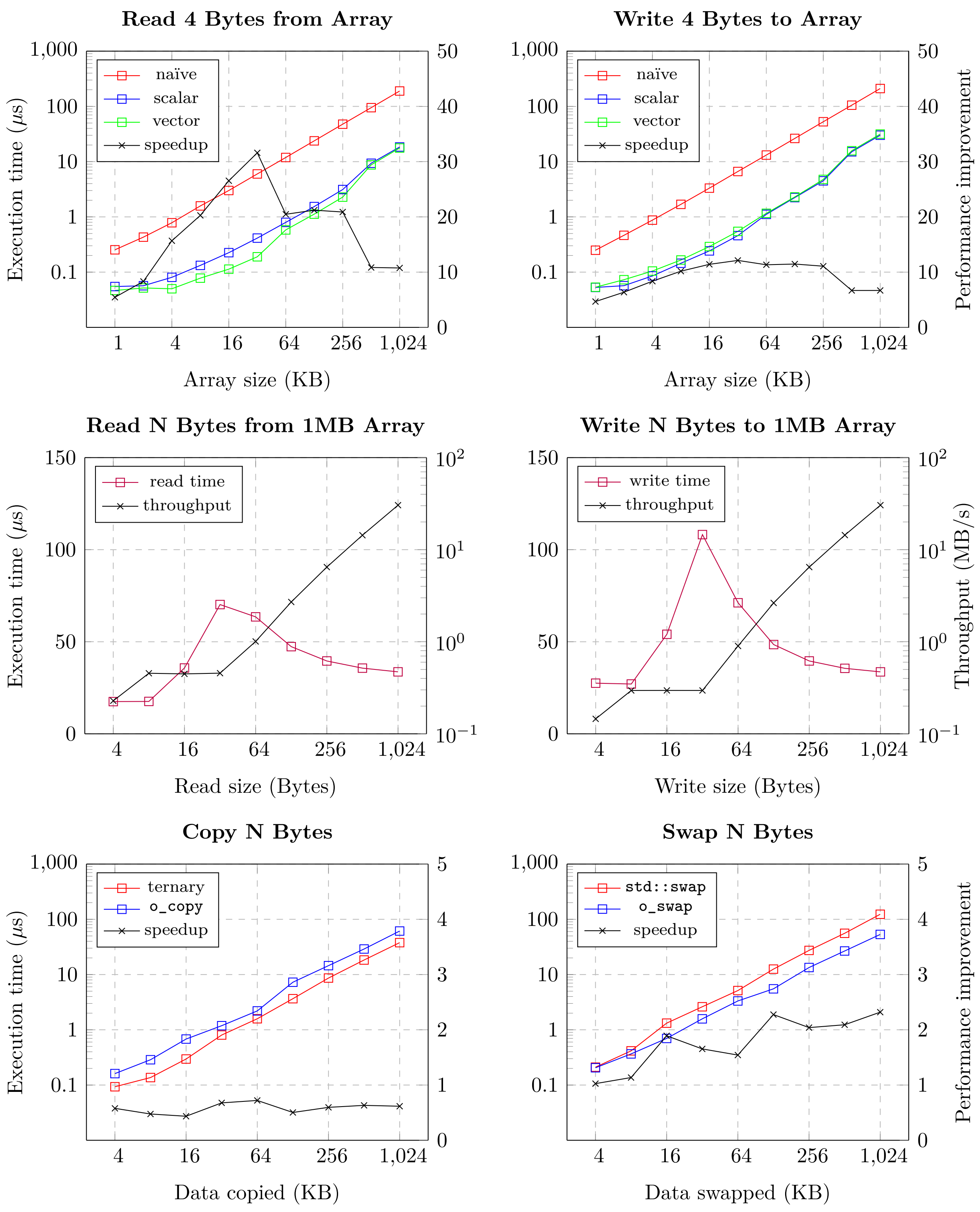

Figure 8.4: Performance results for libOblivious primitives

access to a newer Intel-based machine with AVX512 instructions was not available during the writing of this dissertation. The AVX512 architecture introduces the `vpscatter` family of instructions. These instructions could be used to drastically improve the performance of the oblivious write primitives, in the same manner as `vpgatherdd` does for oblivious reads. At best, a 12x performance improvement was observed in the L1 cache, 11x in the L2 cache, and 7x in the L3 cache.

The second row of plots in Figure 8.4 shows the performance of the `o_read()`/`o_write()` primitives against the number of bytes being read/written from an array of fixed size, 1 MB. Each plot also shows the throughput (number of MB read/written per second) of the operation. Throughput is at its worst when reading or writing only 4 bytes. In this case, `o_read_i32()` and `o_write_i32()` are being called by `o_read()` and `o_write()`, respectively. When the test parameter is increased to 8 bytes, `o_read()` instead calls `o_read_i64()`, and likewise for `o_write()`. These 64-bit primitives, as discussed in Section 8.1.5, instead use the `vpgatherqq` instruction, which gathers four 64-bit integers instead of eight 32-bit integers. The execution time is nearly identical, and thus the throughput is doubled.

Throughput remains steady until the parameter size reaches 64 bytes: the size of a cache block. At this point, `o_read()` and `o_write()` instead use `o_copy()`. The reasons for this were discussed in Section 8.1.5. Consequently, for reads and writes exceeding 64 bytes, the throughput increases linearly with respect to the number of bytes being read or written.

The bottom row in Figure 8.4 shows the results for the `o_copy()` and `o_swap()` primitives, respectively. The first plot compares the performance of `o_copy()` against the C++ ternary (`?:`) operator. As the number of bytes copied increases, the slowdown of `o_swap()` converges toward 1.5x. This result is predictable. When the statement

```
*d = c ? *x : *y;
```

is compiled (and the pointee type is sufficiently large), the compiler will produce a branch depending on `c`, which will invoke a `memcpy()` either from `*x` to `*d` or from `*y` to `*d`. If the libc `memcpy()` is optimized for AVX2, then it will repeatedly invoke vectorized `vmov` instructions to load from either `*x` or `*y`, and then store in `*d`. The analogous libOblivious operation would be

```
*d = o_copy_T(c, *x, *y);
```

which would similarly invoke `vmov` to load from both `*x` and `*y`, but then use `vpblendvb` to select the correct vector depending on `c`, and finally write that result to `*d` with a third `vmov`. So the oblivious copy is performing exactly three vectorized memory accesses for each two vectorized memory accesses performed by `memcpy()`, which explains the 1.5x slowdown.

The performance results for `o_swap()` are more impressive. The oblivious swap implementation actually outperforms the C++ STL's `std::swap()` implementation by up to 2.3x. This is because `std::swap()` is not optimized with AVX2 to vectorize swaps on sufficiently large parameters, at least as of libstdc++ 8.1.0. At a glance, this might seem like a surprising oversight. However, it is conventional in C and C++ to swap values in this manner only when the types of the values are sufficiently small, e.g., when they can fit into a general purpose register. When the values are larger, the developer is instead expected to swap pointers to these values, which is usually more efficient.

8.2.3 Algorithm Results

Each libOblivious algorithm was tested over one or more oblivious containers. The results are shown in Figure 8.5. The first row of plots compare the `ofind()` implementation against `std::find()` over a vector and over a linked list. The container is initialized with a sequence of integers $\{0, \ldots, n-1\}$, where n is the number of elements which can fit into the container, for the given size in bytes. Each test searches for a random element in the range $[0, n)$. Hence, on average the oblivious find would ideally perform twice as many memory accesses as the standard library find (because on average, the randomly chosen element will be located at the middle of the range). The results demonstrate a roughly 10-12x slowdown over the oblivious vector container, and 2.7x over the oblivious forward list. The reasons for this discrepancy are discussed in the next section.

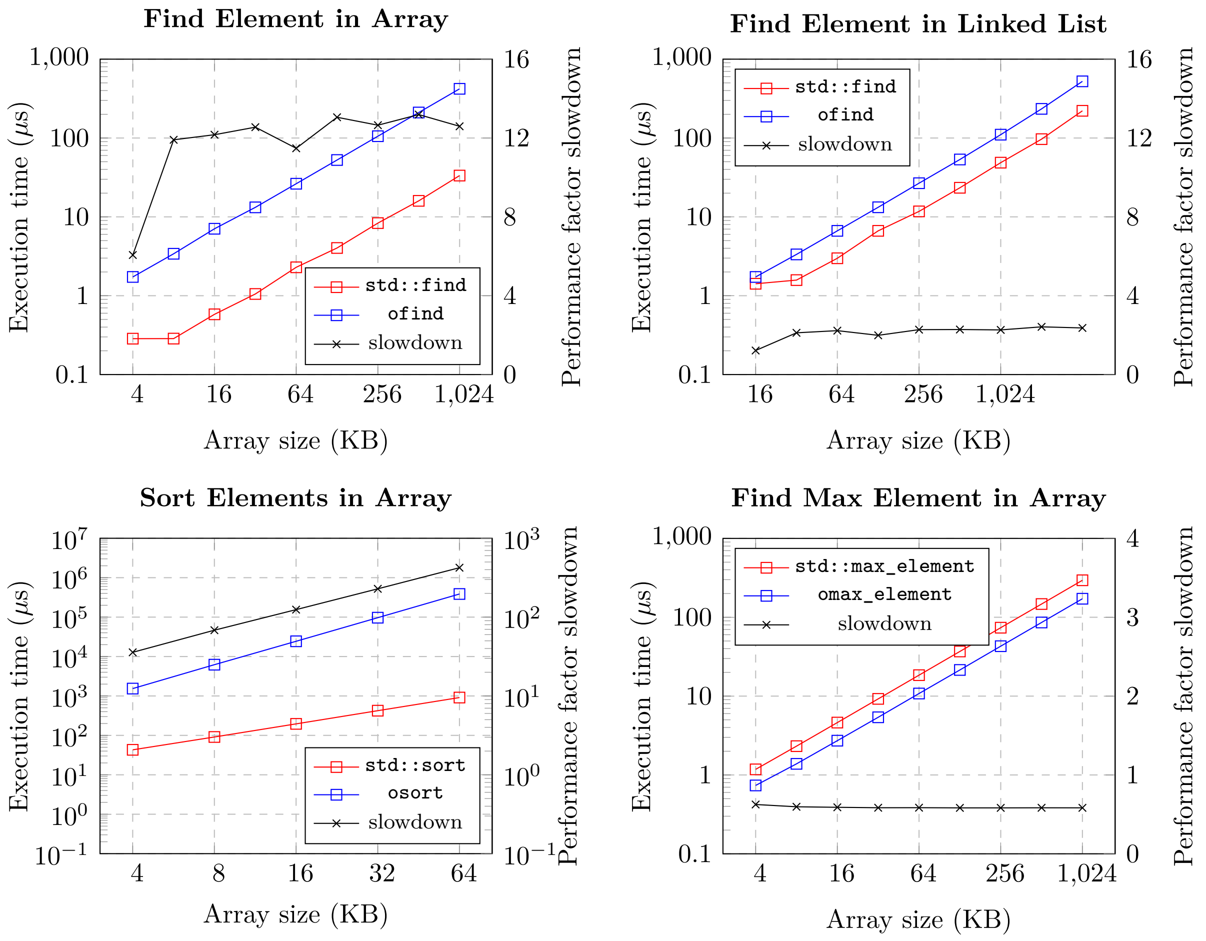

Figure 8.5: Performance results for libOblivious algorithms

The oblivious sort is the worst performing algorithm, when compared to its STL counterpart. Over an array of size 64 KB, `osort()` runs nearly three orders of magnitude slower, on average. And this discrepancy only increases as the size of the container increases. This is due to the asymptotic complexity of the oblivious sort, which is $O(n^2)$, as opposed to $O(n\,log(n))$ for `std::sort()`.

Of the three algorithms discussed in this section, only `omax_element()` actually outperforms its STL counterpart, improving performance by nearly a factor of two. However this may be related to a known optimization bug in GCC[1]. When both `omax_element()` and `std::max_element()` are compiled by Clang, a 3.4x slowdown is observed over a 1MB array. Because the entire container must always be searched to find the maximum element, an optimal result would have been no slowdown, or very little slowdown.

8.2.4 Discussion

With the exception of oblivious writes (due to lack of vector scatter support on the test machine), the oblivious primitives perform exceptionally well. However, the oblivious algorithms—which use those same primitives—do not always perform as well.

As discussed earlier in Section 8.1.5, the oblivious copy and swap primitives needed to be implemented in assembly so as to avoid compiler optimizations that would trigger a branch based on the (potentially) secret copy/swap condition. An unfortunate side effect of this approach is that the presence of inline assembly in any compiler basic block disables many optimizations for the rest of that basic block [18]. If the basic block is encountered infrequently, then the slowdown may be unnoticeable. If the basic block is part of a loop, then the slowdown will be multiplicative. In the case of the oblivious sort implementation, the inline assembly is actually nested within two loops.

This is precisely what has been observed during manual inspection of the compiled binaries of the oblivious algorithms. For example, compared to `std::find()`, the compiler output for

[1]bug reported here: `https://stackoverflow.com/questions/25622109/why-is-c-stdmax-element-so-slow`

`ofind` over a vector is more complex, and unnecessarily so. The performance discrepancy for `ofind` over a linked list is smaller, in part because the number of CPU cycles consumed by each iteration of a traversal over a linked list is partially dominated by reading the address of the next link from memory. There is still overhead attributable to the disabled optimizations, but it is less noticeable.

One solution to close the performance gap would be to manually write the loop in assembly, for each loop which contains an oblivious copy or swap primitive. Unfortunately it is not possible to do this in a manner which is entirely generic for each algorithm. For instance, `std::find()` is parameterized by the type of its iterator arguments. The compiler uses this type to determine which increment operator, dereference operator, and value equality comparison operator to use. Each of these is called within the main loop, and they are almost always inlined by the compiler. Thus, for any two invocations of `std::find()` which differ only in container type, iterator type, or value type, the compiler may produce a different binary.

It is not possible to manually write enough inlined assembly procedures to account for all combinations of type parameters, especially for every oblivious algorithm. It may be possible to write assembly procedures which call the dependent operators, instead of inlining them. However, this would likely produce code that would also perform poorly.

Of all the algorithms that were tested, only `osort()` demonstrated exceptionally poor performance. The existing implementation is a simple variation on bubble sort, but with oblivious swaps. A better solution would be to use something like Batcher's sorting network [30], which has complexity $O(n\,log^2\,n)$. This approach has been employed in [124] for memory-trace oblivious computation. Efficient, distributed oblivious sorting has also been demonstrated [176] using column sort [101].

8.3 Oblivious Iterators and Algorithms in Covert C++

Although libOblivious can be used as a stand-alone library to facilitate oblivious computation, this approach is not recommended. libOblivious can only guarantee memory-trace obliviousness when its interfaces are used correctly. With libOblivious alone, it is possible to

- pass a secret value as an argument to a libOblivious primitive which does not guarantee memory-trace obliviousness for that argument,
- run a non-oblivious algorithm over an oblivious container,
- run an oblivious algorithm over a non-oblivious container,
- or forget to use an oblivious iterator instead of a non-oblivious one, which could easily happen when using the `auto` keyword.

All of these pitfalls could result in a leak of sensitive information through a side channel. Moreover, libOblivious only makes security guarantees for its own components. The developer is responsible for ensuring the security of sensitive data throughout the rest of his program.

The solution proposed in this section is to integrate libOblivious into Covert C++. This accomplishes two security goals. First, Covert C++ augments the libOblivious interfaces with security typing. This ensures that secret values cannot accidentally be passed to non-secure parameters of libOblivious primitives and algorithms. Second, Covert C++ can verify that sensitive data is not being misused throughout the rest of the program.

The first notable feature of Covert C++ with libOblivious is security typing for oblivious iterators. Instances of the `O` template can be wrapped by the `SE` template, and thus endowed with security typing rules that are less restrictive than the typing rules for ordinary `SE` pointers. Henceforth, `SE` oblivious iterators (including pointers) are referred to as *oblivious* types.

The typing rules for memory access operations on oblivious types are shown in Figure 8.6, where Ω denotes an oblivious type. The SUBSCRIPT READ and SUBSCRIPT WRITE rules allow an oblivious type to be read/written from/to, regardless of the security labels of the subscript

T :=	A labeled type	e :=	An expression
S :=	A Covert C++ security label	γ :=	A canonical type
		Ω :=	An oblivious type

$$\textsc{Indirection Read}\ \frac{\begin{array}{c} e : \Omega \qquad T_\Omega = \mathbf{proj}_L\langle\Omega\rangle \qquad S_\Omega = \mathbf{proj}_R\langle\Omega\rangle \\ \tau = \texttt{std::iterator\_traits}\langle\Omega\rangle\texttt{::value\_type} \\ T = \mathbf{proj}_L\langle\tau\rangle \qquad T' = \mathbf{proj}_L\langle\tau'\rangle \\ \texttt{std::is\_convertible}\langle T, T'\rangle \\ \texttt{is\_se\_convertible}\langle\tau, \tau'\rangle \\ (S, \ldots) = \mathbf{proj}_R\langle\tau'\rangle \qquad S_\Omega \leq S \end{array}}{\mathit{implicit\ cast}\langle\tau'\rangle(\texttt{*}e) : \tau'}$$

$$\textsc{Indirection Write}\ \frac{\begin{array}{c} e : \Omega \qquad e' : \tau' \\ T_\Omega = \mathbf{proj}_L\langle\Omega\rangle \qquad S_\Omega = \mathbf{proj}_R\langle\Omega\rangle \\ \tau = \texttt{std::iterator\_traits}\langle\Omega\rangle\texttt{::value\_type} \\ T = \mathbf{proj}_L\langle\tau\rangle \qquad T' = \mathbf{proj}_L\langle\tau'\rangle \\ \texttt{std::is\_convertible}\langle T', T\rangle \\ \texttt{is\_se\_convertible}\langle\tau', \tau\rangle \\ (S, \ldots) = \mathbf{proj}_R\langle\tau\rangle \qquad S_\Omega \leq S \end{array}}{\texttt{*}e = e' : \tau}$$

$$\textsc{Subscript Read}\ \frac{\begin{array}{c} e : \Omega \qquad e' : \tau_{diff} \\ T_\Omega = \mathbf{proj}_L\langle\Omega\rangle \qquad S_\Omega = \mathbf{proj}_R\langle\Omega\rangle \\ T_{diff} = \mathbf{proj}_L\langle\tau_{diff}\rangle \qquad S_{diff} = \mathbf{proj}_R\langle\tau_{diff}\rangle \\ \tau = \texttt{std::iterator\_traits}\langle\Omega\rangle\texttt{::value\_type} \\ T = \mathbf{proj}_L\langle\tau\rangle \qquad T' = \mathbf{proj}_L\langle\tau'\rangle \\ \texttt{std::is\_convertible}\langle T, T'\rangle \\ \texttt{is\_se\_convertible}\langle\tau, \tau'\rangle \\ (S, \ldots) = \mathbf{proj}_R\langle\tau'\rangle \qquad S_\Omega \sqcup S_{diff} \leq S \end{array}}{\mathit{implicit\ cast}\langle\tau'\rangle(e\texttt{[}e'\texttt{]}) : \tau'}$$

$$\textsc{Subscript Write}\ \frac{\begin{array}{c} e : \Omega \qquad e' : \tau_{diff} \qquad e'' : \tau' \\ T_\Omega = \mathbf{proj}_L\langle\Omega\rangle \qquad S_\Omega = \mathbf{proj}_R\langle\Omega\rangle \\ T_{diff} = \mathbf{proj}_L\langle\tau_{diff}\rangle \qquad S_{diff} = \mathbf{proj}_R\langle\tau_{diff}\rangle \\ \tau = \texttt{std::iterator\_traits}\langle\Omega\rangle\texttt{::value\_type} \\ T = \mathbf{proj}_L\langle\tau\rangle \qquad T' = \mathbf{proj}_L\langle\tau'\rangle \\ \texttt{std::is\_convertible}\langle T', T\rangle \\ \texttt{is\_se\_convertible}\langle\tau', \tau\rangle \\ (S, \ldots) = \mathbf{proj}_R\langle\tau\rangle \qquad S_\Omega \sqcup S_{diff} \leq S \end{array}}{e\texttt{[}e'\texttt{]} = e'' : \tau}$$

Figure 8.6: Covert C++ oblivious typing rules.

and the oblivious type itself. The INDIRECTION READ and INDIRECTION WRITE rules are essentially special cases of their respective subscript rules, where the subscript value is equal to 0, and its security label is L.

Recall the discussion from Section 2.3.1 pertaining to implicit flows. The typing rules for pure Covert C++ given in Figure 5.3 and the derived typing rules for control flow statements did not allow any implicit flows of high data. For the first time in this dissertation, the rules in Figure 8.6 must consider implicit flows involving high data.

Consider the following variation on an example from Section 2.3.1:

```
    y = parr[n];
```

Here, `n` is some index and `parr` is some pointer into an array. There is an explicit flow from `parr[n]` into `y` (i.e., `parr[n]` $\Rightarrow_E$ `y`). There is also an implicit flow from the subscript value, `n` $\Rightarrow_I$ `y`. Since `parr` is pointing to some element in an array—not necessarily its base—`parr` is effectively transmitting some information (its value) into `y`. Hence `parr` $\Rightarrow_I$ `y`.

If `parr` were instead an oblivious type, then this assignment statement would be typed by the SUBSCRIPT READ rule. To ensure that the label(s) monotonically increase from the array element to `y`, the flow `parr[n]` $\Rightarrow_E$ `y` is checked by the `is_se_convertible`$\langle\tau, \tau'\rangle$ clause. S, S_Ω, and S_{diff} are the security labels of the destination object, covert type, and the subscript value, respectively. That is, `y`, `parr` and `n` in the example above. The clause $S_\Omega \sqcup S_{diff} \leq S$ ensures that the two implicit flows do not downgrade secret information flowing from the iterator and the subscript value (recall that $\sqcup$ is the least upper bound operator).

The implementation of oblivious types is non-trivial. It entails overloading the `SE` pointer operations for `O` with new operations that encode the additional typing clauses for implicit flows. Moreover, recall from Section 8.1.5 that oblivious iterators use the temporary proxy idiom to achieve the desired memory access behavior. The actual read/write from/to memory is performed by a proxy object that is returned when an oblivious iterator is dereferenced, and then the proxy object is either converted into the element type, or is assigned a value. Hence this proxy object must also be wrapped by a *covert oblivious accessor*. It is ultimately the responsibility of the covert oblivious accessor to ensure that the read or write is permissible.

In more detail, the procedure for type checking a subscripted memory access on an oblivious type is as follows:

1. (Determine the type of the covert oblivious accessor) The covert oblivious accessor is itself a template class, parameterized by the security label that characterizes the information that will flow through the accessor. This is the least upper bound of the security label of the oblivious type and the subscript value (i.e., $S_\Omega \sqcup S_{diff}$). If $S_\Omega \sqcup S_{diff} = L$, then there is no implicit flow of sensitive information, and an oblivious memory access is unnecessary. In this case, the covert oblivious accessor is a wrapper around a proxy object which performs an ordinary (non-oblivious) memory access.
2. (Determine whether the operation is a read or a write) The covert oblivious accessor also follows the template proxy idiom. Hence if the accessor is being read from, then its implicit conversion operator will be compiled. If it is being written to, its assignment operator will be compiled.
3. (Perform the type check) In either case, verify that the explicit flow is permissible.
   (a) If the operation is a read, verify that the label of the implicit flow, $S_\Omega \sqcup S_{diff}$, is no greater than the label of the destination type for the implicit conversion.
   (b) If the operation is a write, verify that the label of the implicit flow, $S_\Omega \sqcup S_{diff}$, is no greater than the label of the container element.

Covert C++ also wraps the libOblivious algorithms. This serves two purposes. First, it allows Covert C++'s type checking to prevent sensitive data from being used in an unsafe manner, e.g., as an argument to an algorithm which does not guarantee security for that particular argument. It also makes the algorithms more generic. The developer does not need to manually choose between the oblivious algorithm and the performance-optimized algorithm. By analyzing the types of the parameters, Covert C++ oblivious algorithms can automatically determine the best option, as with the Covert C++ `memcmp()` example in Section 5.6.

```
 1 template <class RandomIt, class Compare>
 2 void sort(RandomIt first, RandomIt last, Compare comp) {
 3   using comp_traits =
 4       covert_traits<decltype(comp(*first, *first))>;
 5   constexpr auto value_label = comp_traits::label;
 6   using label_type = typename comp_traits::label_type;
 7   constexpr auto bottom = Lattice<label_type>::bottom;
 8   if constexpr (value_label == bottom) {
 9     std::sort(first, last, comp);
10   } else {
11     oblivious::osort(first, last, [comp](const auto &x,
12                                          const auto &y) {
13       return covert_label_cast<label_type, bool, bottom>(
14           comp(x, y));
15     });
16   }
17 }
```

Listing 8.4: The Covert C++ `sort()` implementation

For example, Listing 8.4 shows the implementation of `covert::sort()`, whose usage is demonstrated in the case study presented later in this chapter. The `covert_traits` template was described in Chapter 7. In Line 4 it is used with `decltype` to deduce the traits for the type which is the result of comparing two values in the container, given the comparison functor `comp`. In Line 5 it provides the security label for the comparison result. If this label is the `bottom` label for the given lattice, the optimized C++ STL `sort()` will be used. Otherwise, libOblivious' oblivious sort will be called[2].

The `if constexpr` statement in Line 8 was introduced in C++17 [90]. It is similar to an ordinary `if`/`else`, except that it is evaluated at compile time. Hence for each instantiation of the `covert::sort()` template, only one of the two branches will be compiled.

All of the Covert C++ oblivious algorithms have been verified by the NVT in a manner similar to that which was described in Section 6.3. The only difference in the NVT tests for the oblivious algorithms is that the adversary's granularity is set to 6 bits, the binary

[2]The label of the comparison result must be downgraded in Line 13 because the oblivious sort function does not know how to handle high data, though it will do so securely

```
INPUT: $z$, $k$, $classes$, $d$, $T = \{(x_i, y_i)\}_{i=1,\ldots,n}$
  Find $T_{Near} = \{(x_i, y_i)\}_{i=1,\ldots,k}$ and $T_{Far} = \{(x_i, y_i)\}_{i=1,\ldots,n-k}$ such that:
    $T = T_{Near} \cup T_{Far}$ and $T_{Near} \cap T_{Far} = \emptyset$ // $T_{Near}$ and $T_{Far}$ partition $T$
    $\forall (x, y) \in T_{Near}, (x', y') \in T_{Far}.\ d(y, z) \leq d(y', z)$
  For $c$ in $classes$:
    Set $count[c] := 0$
  For all $(x, y) \in T_{Near}$:
    Set $count[x] = count[x] + 1$
  Find $c \in classes$ such that $\forall c' \in classes.\ count[c'] \leq count[c]$
OUTPUT: $c$
```

Listing 8.5: The k-nearest neighbors (k-NN)

logarithm of the size of an x86 cache block.

8.4 Case Study 3: k-Nearest Neighbors

One popular application of cloud computing is machine learning. An example of a simple, yet powerful classification algorithm is the k-nearest neighbors algorithm [60]. In brief, k-NN is a non-parametric classification algorithm where the program inputs consist of a training set and a test set. Each set has a series of data points, and each data point is characterized by a sequence of attributes. A generic implementation of k-NN would also accept as a parameter a measure of distance d between data points. For each data point z in the test set, k-NN finds its k nearest "neighbors" in the training set (using d). The class c assigned to z by k-NN is the mode of the classes of its k nearest neighbors. If the neighbors have no mode (e.g., two classes are equally represented among the k neighbors), then k-NN must use some heuristic (e.g., randomness) to choose the class for z.

Listing 8.5 describes the algorithm for a k-NN classifier in more detail. The k-NN classifier uses a training set T to classify a data point z. The training set consists of pairs (x_i, y_i), where each y_i is a data point, and each x_i is the classification (some element of $classes$) assigned to y_i. The algorithm proceeds in three steps. The first step is to partition T into T_{Near} and T_{Far}: the k data points nearest to z (according to d), and the remaining $n-k$ data

```
 1 template<typename It>
 2 It max_element(It first, It last)
 3 {
 4     if (first == last) return last;
 5
 6     It largest = first++;
 7     for (; first != last; ++first) {
 8         // leaks the ordering relationship
 9         // between `largest` and `first`
10         if (*largest < *first) {
11             largest = first;
12         }
13     }
14     return largest;
15 }
```

Listing 8.6: A leaky `max_element()` implementation

points, respectively. Next, the classes of the data points in T_{Near} are tallied. The classifier assigns to z the class which has won the highest tally among z's nearest neighbors.

The procedures in each of these three steps have the potential to leak data through a side channel. Typically, the first step would be implemented by sorting the data structure containing T and then selecting the first k elements. Common sorting algorithms such as quick sort, merge sort, and insertion sort all have control flow patterns which depend on the ordering relationships between container elements. If the class tallying is performed with any kind of dictionary-like data structure, as Listing 8.5 would suggest, then each key lookup would almost certainly leak the value of the key.

Finally, determining the class with the highest tally could leak any or all of the tallies. Consider Listing 8.6, which defines a function to determine the maximum element in a container. The relationship between the current element and the largest element found so far is leaked in Line 10, where the `<` relation between container elements influences a branch condition.

In general, it is difficult to write memory-trace oblivious algorithms from scratch. This is why it is essential to provide the developer with tools to facilitate certifiable memory-trace oblivious programming. With libOblivious's fast memory-trace oblivious primitives and Covert C++'s strict information-flow policy, it becomes possible to write machine learning algorithms that are verifiably secure. This case study demonstrates how to use Covert C++ to construct a simple k-NN implementation.

Each data point can be represented as a structure, wherein each field has a covert type:

```
struct KNN_Entry {
  SE<int, H> category;
  SE<unsigned int, L> num_attributes;
  SE<double *, L, H> attributes;
};
```

Note that the word "category" must be substituted in place of "class" in the source code, because `class` is a reserved keyword in C++. The covert types clearly delineate between public and secret data. The function which assigns a category to a data point has the following signature:

```
void classify_entry(SE<unsigned, L> k,
  SE<unsigned, L> num_categories, SE<KNN_Entry*, L> entry,
  SE<const KNN_Entry*, L> training_set,
  SE<unsigned int, L> training_set_size);
```

Hence the only secret inputs are the `category` and `attributes` fields of the `training_set` and `entry` (the data point to classify) parameters. The decision to designate only these two input fields as secret is both subjective and practical. The attributes and classification for each data point directly correspond to the raw data. Thus if the raw data must be kept secret, these inputs must be kept secret. On the other hand, the metadata does not necessarily need to be kept secret. In this example, the metadata includes the size of the training set and the number of attributes in each data point. The `k` parameter determines the number of nearest neighbors among the training set to poll for each member of the test set. `k` is neither secret data nor metadata, thus it is up to the developer to designate `k` as secret or public. For simplicity, this case study assumes that `k` is public. The total number of classes which can be assigned to a data point, `num_categories`, is also assumed to be public.

The body of the `classify_entry()` function follows the description of k-NN given in Listing 8.5. First, the training set data points are sorted by their proximity to the given test set data point: `entry`. The proximity is the Euclidean distance between two data points, according to their corresponding attributes. That is,

$$d(p, q) = \sqrt{\sum_{i=0}^{n-1} (p_i - q_i)^2}$$

where p_i is the i^{th} attribute of data point p, and n is the number of attributes per data point. The Euclidean distance can be computed in Covert C++ as follows:

```
auto Euclidean_distance =
  [](auto x, auto y, SE<std::size_t, L> len) {
    auto distance = (x[0] - y[0]) * (x[0] - y[0]);
    for (SE<std::size_t, L> i = 1; i < len; ++i) {
      distance += (x[i] - y[i]) * (x[i] - y[i]);
    }
    return covert::sqrt(distance);
  };
```

This lambda function uses type inference (via the `auto` keyword) to declare every variable except for `i` and `len`, which determine the loop termination condition, and therefore must be public. Together with the typing rules in Figure 5.3, type inference guarantees that any value returned by `Euclidean_distance()` will have a security label that is the least upper bound of the pointee labels for `x` and `y`. Hence if `Euclidean_distance()` is invoked for the `attributes` fields of two data points, then the return value will have type `SE<double, H>`.

The proximity between `entry` and each data point in the training set can then be computed:

```
const auto neighbors = new pair[training_set_size];
for (unsigned int i = 0; i < training_set_size; ++i) {
  neighbors[i] =
    {training_set + i,
     Euclidean_distance(training_set[i].attributes,
                        entry->attributes,
                        entry->num_attributes)};
}
```

where the `pair` type associates each data point in the test set with its distance from `entry`:

```
struct pair {
  SE<const KNN_Entry *, L> entry;
  SE<double, H> distance;
};
```

Note that if the `distance` field had instead been declared with type `SE<double, L>`, the assignment to `neighbors[i]` above would have failed to type check. The reason is that the value returned by the call to `Euclidean_distance()` is secret whenever one or both of its first two arguments points to an array of secret attributes.

Since these distances are derived from the secret attributes, they must not be leaked. The C++ STL's `std::sort()` algorithm is not oblivious, hence it may leak information about the values in the container being sorted. The Covert C++ library provides the `covert::sort()` function. When the container to be sorted contains high data, `covert::sort()` calls the `oblivious::osort()` algorithm provided by libOblivious. Otherwise, `covert::sort()` simply forwards the request to `std::sort()`. This decision is made at compile time.

```
auto cmp = [](const pair &p1, const pair &p2) {
  return p1.distance < p2.distance;
};
covert::sort(neighbors, neighbors + training_set_size, cmp);
```

The second step in Listing 8.5 is to tally the "votes" for each class among the k nearest neighbors. This computation can also leak information. Below, a direct address table implemented as an oblivious vector is indexed by class; it will be used to record the tally for each class.

```
using HVector = oblivious::ovector<SE<unsigned, H>>;
using HVectorIt = typename HVector::iterator;
HVector class_tally(num_categories);
```

Whenever one of the k nearest neighbors votes for its class, that class's tally is incremented in the table. This increment operation—a read followed by a write—could leak the address in memory where the tally is being updated, thus potentially revealing the class of that nearest neighbor. The `O` template from libOblivious can be used to perform this update obliviously:

```
const SE<O<HVectorIt, HVector>, L> optr{
  class_tally.begin(), &class_tally
};
for (SE<std::size_t, L> i = 0; i < k; ++i) {
  auto category = neighbors[i].entry->category;
  optr[category] = optr[category] + 1;
}
```

A subscript access on an oblivious type will not leak the value of the subscript argument, nor will it leak the value of the oblivious type itself. Thus the value of `category`, which is secret, will not be leaked. If `optr` had instead been a non-oblivious covert pointer, then the expression `optr[category]` would have failed to type check by rule SUBSCRIPT.

The third step in Listing 8.5 is to determine the class which received the highest tally. Just as `std::sort()` was not oblivious, `std::max_element()` is also not memory trace oblivious. The Covert C++ algorithms library provides a solution:

```
entry->category = covert::max_element(
    class_tally.begin(), class_tally.end(), &class_tally)
    - optr;
```

Like `covert::sort()`, `covert::max_element()` is optimized for performance when it is applied over a container with public data. Otherwise, it calls `oblivious::omax_element()`. It also uses type inference to determine the appropriate label for the return value—in this case, `H`.

The assignment to `entry->category` completes the algorithm. The implementation is listed in full in Appendix B.

Chapter 9

Related Work

Covert C++ is a language-based technique which simultaneously enforces classical noninterference and memory-trace obliviousness. The large body of work related to Covert C++ can be divided into one of three categories: language-based techniques, program transformation techniques, and enforcement at the architectural level. Some works combine aspects of two or three of these categories. The following three sections survey the related work in each category. Several of these techniques use some kind of oblivious RAM (ORAM) to obfuscate memory accesses. This section concludes with a brief discussion on related works on ORAM.

By far the greatest influence on the Covert C++ implementation has been Ironclad C++. Ironclad C++ [63] is a library-augmented subset of C++ which also utilizes templates and template metaprogramming. It places various constraints on C++ which make the language type safe and memory safe. For instance, Ironclad C++ mandates the use of bounded arrays, smart pointers, and bounded pointers, and restricts the use of void pointers. It also prevents use-after-free errors for stack and heap pointers. Osera et al. [126] derived a model of typing and semantics for Ironclad C++, and proved (by hand) that Ironclad C++ does indeed enforce type safety and memory safety. Their approach does not consider information flows, which is the objective of Covert C++. The name "Core Covert" is an homage to the Core Ironclad [126] verification effort.

The techniques used in Chapter 8 to implement memory-trace oblivious primitives are derived from recent work [124] published by Microsoft Research, and earlier by Rane et al. [133]. Ohrimenko et al. [124] used memory-trace oblivious primitives to implement several popular machine learning algorithms. They employed x86 instructions such as `cmov` and `vpgatherdd` to build oblivious primitives, upon which they constructed several popular machine learning algorithms. Their approach has two notable limitations. First, it requires the developer to be highly skilled at manually identifying potential side channel vulnerabilities in C/C++ code, and to know when and how to use the oblivious primitives to avoid leaks. Second, the verification of the memory-trace oblivious property is limited to an informal proof for a pseudocode description of each algorithm. Side-channel leaks can be subtle, and thus easily overlooked by an experienced developer. Covert C++ can catch many of these leaks automatically, and leaks which are not caught by Covert C++ can still be exposed by the NVT. This defense-in-depth strategy which couples two drastically different protection mechanisms has not been presented in contemporary related works.

Overall, Covert C++ and its broader toolchain improve on existing techniques and related works in the following ways:

- Covert C++ is the first language-based technique to implement security typing entirely within the bounds of a commonly used programming language, i.e., without requiring language extensions, compiler modifications, etc.
- By utilizing overload resolution and conditional compilation (a feature of template metaprogramming), Covert C++ can automatically optimize algorithms depending on the sensitivity of their parameters. This technique has not been considered in related works on security-typed languages.
- The kinds of application scenarios illustrated in the case studies are typically implemented using imperative industrial languages such as C and C++. Many related works that add security typing to existing languages have focused on functional languages. Two exceptions are JFlow [119] (for Java) and Obliv-C [175]. Yet both require additional

preprocessing transformations, and thus do not integrate as easily with existing libraries and frameworks (e.g., the SGX SDK [15]) as does Covert C++.

- Other works across each of the three aforementioned categories do not provide the end-to-end noninterference guarantees on x86-64 platforms that are provided jointly by Covert C++ and the NVT. The lone exception is SAFE [26], which uses a custom CPU architecture and domain-specific languages.

9.1 Language-based Techniques

Li and Zdancewic [102] described an embedding of security typing into the Haskell programming language. Similar to Covert C++, this security typing employs a customizable security lattice, with the intent of supporting multi-user applications wherein different users have different privileges. The information-flow policies themselves can be customized by the application designer. The most substantial limitations of this work are that it does not attempt to address side channels (this was not one of their stated goals), and that Haskell is not a commonly used language in industry. A precursor to Li and Zdancewic's work was Pottier and Simonet's work [131] on the Flow Caml language, an extension to Objective Caml (OCaml) which encodes security typing. Unlike the embedding in Haskell, Flow Caml code is not valid OCaml code, and hence cannot be processed by a standard OCaml compiler. Another limitation is that Flow Caml does not support OCaml objects [143].

Andrew C. Myers created JFlow [119], later known as Java Information Flow (JIF), which is the most influential precursor to Covert C++'s type system. JFlow embeds security typing into Java in much the same way that Covert C++ does for C++. Unlike Li and Zdancewic's work, JFlow can track both explicit and implicit flows, though the policy mechanisms are quite different from those of Covert C++. For instance, JFlow does not prevent sensitive data from influencing program control flow. Rather, it employs an approach where the program counter itself receives a security label. If this label is high, then all assignments are treated as high. One drawback of JFlow is that programs written in JFlow are not valid Java code; they must first be transformed by a custom compiler into native Java code.

Liu et al. [104] contributed the foundational work in programming language theory for constructing a programming language with the property of memory-trace obliviousness. Furthermore, their work exceeds memory-trace obliviousness to also cover classical termination-sensitive noninterference. Memory accesses are obfuscated using a maximal partition of ORAM banks over all secret program data structures. Unlike Covert C++, their formulation of a memory-trace oblivious language does allow selection statements (but not iteration statements) to have a secret conditional guard. They achieve this result by generating "dummy" execution sequences along the path(s) not taken by the selection statement.

ObliVM [106] is a domain-specific language for memory-trace oblivious computation, built on the theoretical foundation established by Liu et al. described above. The ObliVM-lang is a C++-like language with features such as structures, generics, loops, etc. Each variable in ObliVM-lang is annotated as either "public" or "secret." Memory-trace obliviousness with respect to the secret values is enforced by the type system. Programs written in ObliVM-lang are compiled into ObliVM-GC, a Java-based garbled circuit [147, 172] implementation. This allows a computation to run on an untrusted platform, while providing confidentiality guarantees on any platform which does not provide dynamic memory encryption, such as SGX.

Obliv-C [175] is an extension (i.e., strict superset) to the C language which adds a category of data called "obliv," similar to the "secret" annotation in ObliVM and `H` in Covert C++. As in ObliVM, typing in Obliv-C enforces memory-trace obliviousness for obliv-annotated values. The implementation is simply a wrapper around GCC which performs the oblivious type checking against a rule set. If no rules have been violated, it translates the Obliv-C code into ordinary C code, which is then compiled by GCC.

9.2 Program Transformation

Program transformation techniques can be used before or during the compilation process to analyze and modify control-flow patterns and memory accesses, for the purpose of achieving some degree of obliviousness.

Raccoon [133] is a kind of source code preprocessing tool which transforms a C or C++ program into a program that is memory-trace oblivious. Raccoon only requires the developer to annotate secret variables in the source code. The tool performs inter-procedural taint analysis on the secret variables to determine precisely where a secret may influence either control flow or a memory access. When secret data is determined to influence a branch toward one of several paths, Raccoon obfuscates the control flow. The obfuscation mechanism forces execution on all paths, while replacing memory writes along each path with an oblivious store operation, similar to the libOblivious `o_write()` primitive described in Chapter 8. The values in memory are only updated along the correct execution path. Raccoon uses software Path ORAM to obfuscate memory accesses which depend on program secrets.

Dr. SGX [40] is an intermediate representation (IR)-level transformation tool and library which obfuscates heap memory accesses to make an SGX enclave program data oblivious, at cache block granularity. Unlike Raccoon, Dr. SGX does not require any additional code annotation by the user. It obfuscates heap memory accesses by first producing a randomized heap memory layout, and then constantly re-randomizing this layout, subject to a configurable time window. Dr. SGX uses an LLVM IR pass to transform the input program, replacing ordinary heap memory accesses with calls to the Dr. SGX library. Each call consults the current iteration of the pseudo-random memory permutation function to obtain the correct randomized address. Although this solution requires less work on account of the developer, it also cannot distinguish between public and secret data. Thus every access is obfuscated, which may introduce unnecessary overhead.

SGX-Lapd [74] specifically targets the forced page eviction adversary model, which may observe enclave memory traces at page granularity, as in [171]. However, instead of promising full prevention of memory-based side-channel attacks, SGX-Lapd is simply a mitigation technique which makes it much more difficult for the adversary to infer enclave secrets from the memory trace. Specifically, SGX-Lapd consists of two components: (1) an untrusted kernel module which provides large (2 MB) pages to the enclave, and (2) a program transformation tool which inserts instrumentation code into the enclave program. Whenever enclave program control flow or data accesses cross a 4 KB-aligned boundary, the instrumentation code dynamically checks whether a forced 4 KB page fault has occurred. If so, this would likely indicate a malicious OS feeding 4 KB pages to the enclave and forcibly evicting them. Like Dr. SGX, SGX-Lapd uses an LLVM IR pass to perform the enclave code transformation. This technique increases the granularity of the adversary's visibility by 9 bits, hence it also increases the mask size by a factor of 2^9.

9.3 Architectural Approaches

SAFE [26] is an end-to-end information-flow approach, spanning from the operating system to the computer architecture level. The SAFE machine associates a security label "tag" with every word in the system state, including in memory, CPU registers, and the program counter. Hence the approach uses a kind of dynamic typing, which distinguishes it from the previously discussed works. SAFE enforces termination-insensitive noninterference between principals (users) for programs which may simultaneously handle data that is private to each user. The system is modeled at three layers of abstraction. The top-level "abstract" model specifies the information-flow policy, and the bottom "concrete machine" precisely describes the implementation. Noninterference was formally verified for the abstract model. The data refinement technique [62] was employed to prove that noninterference is preserved by the implementation. The total verification effort comprises around 15,000 lines of Coq script, including formal proofs.

The GhostRider [105] project features a co-designed assembly language, compiler, and hardware architecture for memory-trace oblivious computation. Static analysis of the assembly allows GhostRider to partition program data into one of three hardware RAMs depending on its security classification and usage: (unencrypted) RAM, (encrypted) ERAM, or ORAM. The usage scenario for GhostRider involves an untrusted host system communicating with a GhostRider co-processor, both of which share the same RAM/ERAM/ORAM banks. The authors simulated their design in software, and on an FPGA system.

9.4 ORAM

Oblivious RAM [78, 79, 150] (ORAM) is a technique for obfuscating memory access patterns, and can be used to defeat side-channel attacks. This technique is appealing because it operates "under the hood," and thus places no additional burden on the developer. However, it does incur a $\Omega(log(n))$ cost for each memory access, where n is the size of the entire program memory. Thus protecting all program secrets in a single ORAM has been has been shown to be inefficient [71, 149, 169].

ZeroTrace [136] is an Intel SGX enclave runtime which uses an ORAM-based memory controller to enforce data obliviousness. The ZeroTrace usage model assumes that a remote client needs to make remote queries on a large (e.g., $>$ 10 GB) dataset. These queries could include reads, writes, key-value lookups, etc. Within an SGX enclave, ZeroTrace uses a Path ORAM [150] library to look up the ORAM leaf containing the query. Within each ORAM leaf, the lookup can be performed by an untrusted third party, hence the final request is forwarded to a non-enclave fetch/store controller, which can execute with less overhead. ZeroTrace has demonstrated favorable performance for queries over large datasets.

Several works have specifically targeted the design of oblivious data structures using ORAM variations. Wang et al. [164] proposed a framework for created ORAM-based oblivious data structures. Their work reduces access time overhead on oblivious data structures from $O(n)$ to $O(log(n))$. However, the data structure storage scheme requires $O(n\,log(n))$ storage.

By specifically optimizing their implementation for each data structure, they are able to achieve a 10-15x speedup over naïve ORAM for moderately sized ORAMs. Keller and Scholl [93] also implemented optimized data structures over ORAMs. Their work specifically considers the problem of secure multi-party computation using oblivious data structures. Their analysis also compares the performance of data structures using Path ORAM vs. Tree ORAM, and also against naïve scalar accesses.

ORAM itself also does not protect against timing side-channel attacks, such as an attacker inferring a loop termination condition from the number of loop iterations. This is a problem which is addressed by Covert C++. For relatively small (e.g., < 100 MB) data structures, ORAM has been demonstrated to be significantly slower than the kind of memory scanning techniques employed by libOblivious and Covert C++ [133].

Chapter 10

Conclusion

This dissertation introduced Covert C++, a new language built on top of C++ to facilitate static information-flow analysis and noninterference. Covert C++'s security guarantees were verified both formally and informally, and furthermore were confirmed for compiled binaries by applying a novel dynamic analysis technique. A series of case studies described how Covert C++ can be harnessed to certify a variety of security-critical applications.

Section 10.1 summarizes the contributions of the dissertation, and argues that the presented body of work satisfies the thesis statement. Section 10.2 discusses some of the areas where Covert C++ can be extended to address other categories of security problems.

10.1 Discussion

The body of work detailed in this dissertation substantiates the thesis statement. Covert C++ uses the Turing-complete C++ template type system to implement an information-flow policy that restricts the propagation of sensitive data. This policy certifies a comprehensive form of noninterference which prevents sensitive information from being leaked through storage channels, termination channels and memory-based side channels. A novel dynamic analysis technique was designed and implemented to verify that an unmodified C++ compiler and linker are able to preserve this noninterference property—and with all optimizations enabled.

Due to the enormous complexity of the C++ language, it is infeasible to formally verify that Covert C++ can guarantee noninterference for every program which type checks. Nonetheless, formal typing rules and an informal proof were presented to argue that Covert C++ does enforce noninterference, subject to several assumptions about memory and type safety.

The Core Covert language is a distilled formulation of Covert C++ with built-in memory and type safety, and nearly identical security-typing rules. The goal of Core Covert is to capture the security-relevant features of Covert C++ as accurately as possible, while remaining simple enough to be amenable to formal verification. A formal proof was presented to verify the claim that Core Covert has the property of termination-sensitive noninterference. Because Core Covert's security typing corresponds to Covert C++'s security typing, this can be taken as further evidence that Covert C++'s typing is sound.

These are not the only novel contributions made by this dissertation. Several noteworthy observations were made in Chapter 5 about the consequences of embedding security typing into C++. First, the embedding effectively "trains" any ordinary C++17-compliant compiler to perform static information-flow analysis on C++ programs. It is worth emphasizing that the approach does not require any modifications to the compiler. Second, C++'s overload resolution can be harnessed to automatically optimize Covert C++ programs, depending on the security classes of function parameters. Third, because library template interfaces must be entirely defined in header files, Covert C++ can be used to statically analyze existing templatized code bases for side-channel vulnerabilities, including the C++ STL.

Chapter 7 demonstrated how Covert C++ can be extended to support arbitrary lattices of security classes, and thus become a framework for composing secure multi-party computations. Chapter 8 described the implementation of libOblivious, a software library to facilitate memory-trace oblivious computation. Covert C++ can use libOblivious iterators to improve performance when the adversary's view of the memory trace is imperfect (i.e., with granularity greater than 0).

Perhaps most significantly, this dissertation demonstrates that it is possible to superimpose an entirely new type system on top of the C++ language. There could be many applications of this approach, reaching well beyond information-flow analysis.

10.2 Future Work

One of the key contributions of this dissertation is that it demonstrates how to use a combination of operator overloading and template metaprogramming to superimpose a security-type system on top of C++'s type system. More broadly, many different kinds of type systems with various properties could be similarly adapted for C++, a topic which has not yet been investigated beyond Covert C++.

10.2.1 Dynamic Data Security

The Covert C++ security-type system as described in Chapter 5 is static in nature, because the Covert C++ information-flow control is a compile-time mechanism. As mentioned in Section 2.5, the original intent of the C++ template system was to provide type-constrained code generation. Hence `SE` types can also be used to generate runtime code, possibly to introduce additional dynamic protections for secret data.

For example, Intel SGX provides an encrypted region of memory called an enclave, with well-defined upper and lower bounds on the address space [59]. The access control policy of SGX hardware does not allow a non-enclave program to copy data directly into an SGX enclave. However, a program executing within an enclave can copy data to a region of memory outside of the enclave [112]. If the enclave program can unintentionally copy sensitive information in plaintext to an unencrypted location outside of the enclave, then this is an example of a storage-channel leak.

The only defense provided by Covert C++ against storage-channel leaks is the From SE rule, which prevents sensitive data from being implicitly downgraded into public data. Only public data can be emitted through a storage channel API, such as `printf()` or `cout`.

```
template <typename T, SLabel S>
struct SE {
  T val;
  ...
  SE<T, S>(SE<T, S> x) { // copy constructor
    if (!copy_guard<S>(this)) {
      exit(1);
    } else {
      val = x.val;
    }
  }
  ...
};
```

Listing 10.1: A guarded copy operation

However, this enforcement mechanism can be fragile. SGX enclave programs typically consist of both enclave and non-enclave code. Calls can be made back and forth between these two components through an interface provided by the SGX SDK [15]. Assuming that this interface is the only means by which data is exchanged between the two components, Covert C++ should catch any storage-channel leak when the program is type checked.

Suppose that some statically-allocated object `x` is labeled as `H` because the developer intends to use it to store sensitive data within the enclave. However, he has misconfigured his linker scripts, and `x` is actually linked into the non-enclave component. Hence each update to `x` will write data outside of the enclave. The program may type check, and assuming that only the enclave component accesses `x`, the SGX access controls will not trigger a memory fault. Therefore the developer may never notice the vulnerability.

No compile-time check can detect this kind of vulnerability because linking occurs after compilation. After compilation, many of the abstractions that were present at the source level (such as types) are lost in translation. One practical way to preclude this kind of vulnerability is with a runtime access control check.

To prevent sensitive information from leaking to a non-enclave component with Covert C++, the copy semantics of `SE` types can be adjusted as in Listing 10.1. Here, `copy_guard()`

(implementation omitted) is a template function specialized on its security label parameter, such that if the argument is `L`, it vacuously returns `true`. If the argument is `H`, it checks whether the destination address for the copy (i.e., `this`) is within the bounds of the enclave, and returns `false` if it is out of bounds.

These basic semantics can also be overloaded to automatically sanitize memory locations containing sensitive data after the data has either gone out of scope or been freed from the heap. This is a new solution to the data lifetime problem proposed by Garfinkel et al. [75]. The current best-practice solution was described by Wheeler: "If your application must handle passwords or non-public keys (such as session keys, private keys, or secret keys), try to hide them and overwrite them immediately after using them so they have minimal exposure" [167]. The naïve solution is to `memset()` the sensitive data to zero, which is insufficient as noted in CERT Secure Coding rule MSC06-C [140]. This rule observes that an optimizing compiler "could employ 'dead store removal' " to optimize away calls to `memset()`. For this reason, the ISO recently introduced `memset_s()` into the C11 standard, as a function which cannot be optimized away by the compiler [89]. Notably, the Intel® SGX Developer Guide recommends that "The enclave writer must use the `memset_s()` function to clear any variable that contained secret data" [2]. Each additional burden placed on the developer is a potential point of failure in a secure system.

C++ classes can also define destructors: functions which are automatically called when an object of a given class goes out of scope or is freed from the heap[1]. The implementation in Listing 10.1 could be augmented with the following destructor:

```
~SE<T, S>() { // destructor
  sanitize_if_not_low<S>(this);
}
```

The `sanitize_if_not_low()` function is a template function, specialized so that it does nothing if the security label parameter is `L`. Otherwise, it calls `memset_s()` to zero out the confidential data stored in the object.

[1]destructors can also be called manually, though this use of destructors is uncommon

One problem with this approach is that it violates a crucial assumption of Covert C++: the SE template does not define any custom constructors, copy/move constructors, copy/move assignment operators, or destructors for SE types. In this sense, SE types are *trivial* [90] in their fundamental semantics. Both of the runtime protection schemes proposed above would render the SE types non-trivial. For some operations, this could cause problems. For instance:

```
SE<int *, L, L> ptr = new SE<int, L>[16];
int *_ptr = ptr;
delete[] _ptr; // program crash?
```

Recall that the implicit cast in the second line is allowed by rule FROM SE. The problem is that new[] and delete[] are being invoked with types whose deletion semantics differ (because ptr has a custom destructor, and _ptr does not), and thus the behavior of delete[] is undefined [90]. One potential solution would be to create custom new and delete operators, but this could introduce compatibility issues.

10.2.2 Generalizing Covert C++

Attaching security labels to primitive data is hardly the only novelty of Covert C++. The work presented in this dissertation suggests that arbitrary compile-time information can be associated with program data, and this additional information can be used to (a) constrain the behavior of certain data, (b) imbue operators with new semantics, and/or (c) decorate the binary. It should not be necessary to write a new framework from scratch for each new problem domain that can be addressed by specifications for (a-c).

The solution is to use policy-based design [21] to make Covert C++ generic. For instance, the typing rules in Figure 5.3 could be interpreted as a special case of a more generic policy. The generic policy would provide a pluggable template, wherein the parameters determine what it means to combine different kinds of data through computation, and how data should behave given this additional compile-time information. The policy interface might be:

```
template <typename DataT, typename Combine, typename Cast>
struct DataPolicy { ... };
```

where `DataT` is the type of the additional compile-time information to associate with each object, `Combine` is a metafunction which defines what it means to combine data (e.g., via arithmetic), and `Cast` specifies the constraints on data transfers. For Covert C++, `Data` would be `SLabel`, `Combine` would be the `lub()` ($\sqcup$) operation, and `Cast` would be the `is_se_convertible` metafunction. For instance:

```
using CovertPolicy =
        DataPolicy<SLabel, lub, is_se_convertible>;
```

Some other template, call it `Data`, would implement the policy in a manner similar to `SE`. A use case might look something like this:

```
Data<CovertPolicy, int, List<H>> x;
Data<CovertSMPCPolicy, short, List<Alice>> y;
Data<CovertDynamicPolicy, int *, List<H, H>> z;
```

Here, `x` has Covert C++ typing, `y` has SMPC Covert C++ typing, and `z` has Covert C++ typing with the dynamic protections described in Section 10.2.1. Covert C++ could also use generalized information-flow and dynamic mechanisms to deploy Myers' decentralized label model [118, 120], which was implemented in JFlow [119].

10.2.3 Adding Path Discovery to the NVT

The NVT operates by fixing public data, and fuzzing secret data. If memory access patterns and outputs do not strongly depend on secret data, then the target function(s) satisfy noninterference. However this analysis is insufficient when public data can also affect a program's control flow. Testing for noninterference against only one fixed set of low inputs can leave some execution paths unexamined.

For example, consider Listing 10.2. Suppose that `arg1` and `arg2` are public and `secret` is (of course) secret. If `arg1` and `arg2` are fixed to values which do not satisfy `arg1 > arg2`, then the vulnerability will not be detected by the NVT.

```
int foo(int arg1, int arg2, int secret) {
  int res = 0;
  if (arg1 > arg2) {
    while (arg1 > secret) { // leak
      res += arg1--;
    }
  } else {
    res = arg2 + secret;
  }
  return res;
}
```

Listing 10.2: A function which violates noninterference

One solution would entail the addition of a preliminary “discovery” phase, in which all low inputs are fuzzed. Each time a set of low inputs exposes a new execution path, the NVT will note those low inputs. Then the second “noninterference” phase will fuzz the high inputs for each set of low inputs that was noted during the discovery phase.

10.2.4 Increasing the Granularity of the Adversary Model

Recent work in academia and in industry has begun to close the door on cache-based side-channel attacks (e.g, [49, 53, 67, 108, 125, 142]). In the near future, it may be possible to completely obviate cache-based attacks with a combination of architecture and compiler-based techniques. If and when this day comes, libOblivious—and thus Covert C++—can be configured to a granularity higher than 6 bits, assuming the x86 architecture.

The next layer in the memory hierarchy on x86 is RAM, where the basic unit of memory is the 4 KB page. Hence Covert C++ could assume page granularity instead of cache granularity to perform memory-trace oblivious computations. This change would reduce the number of memory accesses required to access large data structures by a factor of 2^6. Moreover, Covert C++ could be used in concert with SGX-Lapd [73] to take advantage of x86 large pages, of size 2 MB. Hence the work factor for memory-trace oblivious reads and writes would be further reduced by 2^9, or 2^{15} in total over the current implementation.

Appendix A

Sample NVT Traces

Below is the NVT's trace of the while loop of the safe memcmp function in Listing 5.3 for an input of size 4. The first column contains the memory address that was accessed, the second gives the size (in bytes) of the operand, and the third is the type of access (see Chapter 6). The loop behavior is identical for all inputs, iterating exactly 4 times.

```
0x7f3372309e40:  3, bb
0x7f337250b060:  1, r
0x7f337250b460:  1, r
0x7f3372309e40:  3, bb
0x7f337250b061:  1, r
0x7f337250b461:  1, r
0x7f3372309e40:  3, bb
0x7f337250b062:  1, r
0x7f337250b462:  1, r
0x7f3372309e40:  3, bb
0x7f337250b063:  1, r
0x7f337250b463:  1, r
```

This second example compares two complete traces of the optimized memcmp function in Listing 5.3. With the second set of fuzzed inputs, the loop runs for one additional iteration.

```
Fuzz Iteration 0:          Fuzz Iteration 1:
0x7f745b378c90:  1, bb     0x7f745b378c90:  1, bb
0x7ffd8a3a5a60:  8, w      0x7ffd8a3a5a60:  8, w
0x7f745b579fb8:  8, r      0x7f745b579fb8:  8, r
0x7f745b57a038:  8, r      0x7f745b57a038:  8, r
```

```
0x7f745b579ff0:  8, r     0x7f745b579ff0:  8, r
0x7f745b57a040:  8, r     0x7f745b57a040:  8, r
0x7f745b579fd0:  8, r     0x7f745b579fd0:  8, r
0x7f745b57a860:  4, r     0x7f745b57a860:  4, r
0x7ffd8a3a5a58:  8, w     0x7ffd8a3a5a58:  8, w
0x7f745b378900:  6, bb    0x7f745b378900:  6, bb
0x7f745b57a018:  8, r     0x7f745b57a018:  8, r
0x7f745b378cd0:  3, bb    0x7f745b378cd0:  3, bb
// loop begins here       // loop begins here
0x7f745b378cee:  3, bb    0x7f745b378cee:  3, bb
0x7f745b57a060:  1, r     0x7f745b57a060:  1, r
0x7f745b57a460:  1, r     0x7f745b57a460:  1, r
// loop ends here
0x7f745b378cf8:  1, bb    0x7f745b378ce0:  3, bb
0x7ffd8a3a5a58:  8, r     0x7f745b378cee:  3, bb
0x7f745b378cb3:  2, bb    0x7f745b57a061:  1, r
0x7f745b579fb0:  8, r     0x7f745b57a461:  1, r
                          // loop ends here
0x7f745b57a864:  1, w     0x7f745b378cf8:  1, bb
0x7ffd8a3a5a60:  8, r     0x7ffd8a3a5a58:  8, r
0x7ffd8a3a5a68:  8, r     0x7f745b378cb3:  2, bb
0x40080b:  5, bb          0x7f745b579fb0:  8, r
0x7ffd8a3a5a68:  8, w     0x7f745b57a864:  1, w
                          0x7ffd8a3a5a60:  8, r
                          0x7ffd8a3a5a68:  8, r
                          0x40080b:  5, bb
                          0x7ffd8a3a5a68:  8, w
```

Appendix B

Sample Code for Case Studies

B.1 Covert C++ χ^2 Implementation

```
// square a value
template <typename _T> static constexpr _T square(_T val) {
  return val * val;
}

template <typename _MatrixT, std::size_t... J>
auto chi2_impl(const _MatrixT &Obs, std::size_t rows,
               std::index_sequence<J...> cols) {
  using RetT = decltype((... + std::get<J>(Obs)[0]));

  auto SumColumn = [rows](const auto &col) -> auto {
    auto ret = col[0];
    for (int i = 1; i < rows; ++i) {
      ret += col[i];
    }
    return ret;
  };
  auto ColumnSums =
      std::tuple{SumColumn(std::get<J>(Obs))...};
  auto SumRow = [&Obs](std::size_t i) -> auto {
    return (... + std::get<J>(Obs)[i]);
  };
  std::vector<RetT> RowSums(rows);
  for (int i = 0; i < rows; ++i) {
    RowSums[i] = SumRow(i);
  }
  auto Total = (... + std::get<J>(ColumnSums));

  auto ExpColumn = [&](const auto &col, auto sum) -> auto {
    std::vector<RetT> exp(rows);
    for (int i = 0; i < rows; ++i) {
      exp[i] = sum * (RowSums[i] / Total);
    }
    return exp;
  };
  auto Exp =
      std::tuple{ExpColumn(std::get<J>(Obs),
          std::get<J>(ColumnSums))...};

  RetT ret = 0.0;
  for (int i = 0; i < rows; ++i) {
    ret += (... + ((square(std::get<J>(Obs)[i]
                     - std::get<J>(Exp)[i])) /
                   std::get<J>(Exp)[i]));
  }
  return ret;
}

template <typename _ArgT, typename... _ArgTs>
auto chi2(const _ArgT &arr, const _ArgTs &... arrs) {
  std::size_t size = arr.size();
  assert(size > 0 && "args␣must␣be␣non-empty");
  assert((... && (size == arrs.size()))
         && "args␣are␣not␣all␣the␣same␣size");
  return chi2_impl(
      std::forward_as_tuple<
          const _ArgT &, const _ArgTs &...
      >(arr, arrs...),
      size,
      std::make_index_sequence<1 + sizeof...(_ArgTs)>{});
}
```

B.2 Covert C++ k-NN Implementation

```
struct KNN_Entry {
  SE<int, H> category;
  SE<unsigned int, L> num_attributes;
  SE<double *, L, H> attributes;
};

// Implementation of k-NN with Covert C++ and libOblivious.
static void classify_entry(
    SE<unsigned, L> k,
    SE<unsigned, L> num_categories,
    SE<KNN_Entry *, L> entry,
    SE<const KNN_Entry *, L> training_set,
    SE<unsigned int, L> training_set_size)
{
  struct pair {
    SE<const KNN_Entry *, L> entry;
    SE<double, H> distance;
  };

  // lambda function to compute the Euclidean distance
  // between two data points
  auto Euclidean_distance =
    [](auto x, auto y, SE<std::size_t, L> len) {
      auto distance = (x[0] - y[0]) * (x[0] - y[0]);
      for (SE<std::size_t, L> i = 1; i < len; ++i) {
        distance += (x[i] - y[i]) * (x[i] - y[i]);
      }
      return covert::sqrt(distance);
    };

  // compute the Euclidean distance between the given data
  // point, and each data point in the training set
  const auto neighbors = new pair[training_set_size];
  for (unsigned int i = 0; i < training_set_size; ++i) {
    neighbors[i] =
      {training_set + i,
       Euclidean_distance(training_set[i].attributes,
                          entry->attributes,
                          entry->num_attributes)};
  }

  // sort the training set according to distance from the
  // given data point. the first k elements of the sorted
  // vector are the k nearest neighbors
  auto cmp = [](const pair &p1, const pair &p2) {
    return p1.distance < p2.distance;
  };
  covert::sort(se_static_cast<pair *, L>(neighbors),
                 neighbors + training_set_size, cmp);

  using HVector = oblivious::ovector<SE<unsigned, H>>;
  using HVectorIt = typename HVector::iterator;
  HVector class_tally(num_categories);
  const SE<O<HVectorIt, HVector>, L> optr{
    class_tally.begin(), &class_tally
  };

  // count the tally for each category among the
  // k nearest neighbors
  for (SE<std::size_t, L> i = 0; i < k; ++i) {
    auto category = neighbors[i].entry->category;
    optr[category] = optr[category] + 1;
  }

  // determine the "winner"
  entry->category = covert::max_element(
      class_tally.begin(), class_tally.end(), &class_tally)
      - optr;

  delete[] neighbors;
}
```

Bibliography

[1] Security on ARM TrustZone. [Online]. URL `https://www.arm.com/products/security-on-arm/trustzone`. Accessed: 9 June 2018.

[2] Intel® Software Guard Extensions (Intel® SGX) Developer Guide. [Online]. URL `https://software.intel.com/sites/default/files/managed/33/70/intel-sgx-developer-guide.pdf`. Accessed: 3 September 2018.

[3] Clang-Tidy. [Online]. URL `http://clang.llvm.org/extra/clang-tidy/`. Accessed: 9 April 2017.

[4] Algorithms library. [Online]. URL `https://en.cppreference.com/w/cpp/algorithm`. Accessed: 6 August 2018.

[5] FileCheck - Flexible pattern matching file verifier. [Online]. URL `https://llvm.org/docs/CommandGuide/FileCheck.html`. Accessed 7 February 2018.

[6] History of C++. [Online]. URL `http://en.cppreference.com/w/cpp/language/history`. Accessed: 22 August 2017.

[7] Intel® Software Guard Extensions (Intel® SGX). [Online]. URL `https://software.intel.com/en-us/sgx/details`. Accessed: 9 June 2018.

[8] JSON Compilation Database Format Specification. [Online]. URL `https://clang.llvm.org/docs/JSONCompilationDatabase.html`. Accessed: 7 September 2018.

[9] lit - LLVM Integrated Tester. [Online]. URL `https://llvm.org/docs/CommandGuide/lit.html`. Accessed: 8 February 2018.

[10] SFINAE. [Online]. URL `http://en.cppreference.com/w/cpp/language/sfinae`. Accessed: 6 February 2018.

[11] Intel SGX for Linux. [Online]. URL `https://github.com/intel/linux-sgx`. Accessed: 7 September 2018.

[12] Temporary proxy. [Online]. URL `https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Temporary_Proxy`. Accessed: 1 August 2018.

[13] Introduction to x64 assembly. [Online]. URL `https://software.intel.com/sites/default/files/m/d/4/1/d/8/Introduction_to_x64_Assembly.pdf`. Accessed: 5 August 2018.

[14] The Trusted Execution Environment : Delivering Enhanced Security at a Lower Cost to the Mobile Market. White paper, GlobalPlatform, February 2011. URL `http://www.globalplatform.org/documents/GlobalPlatform_TEE_White_Paper_Feb2011.pdf`. Accessed: 14 September 2017.

[15] *Intel® Software Guard Extensions SDK for Linux OS Developer Reference*, 2016. URL `https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf`. Revision 1.5.

[16] Microsoft macro assembler reference. [Online], November 2016. URL `https://docs.microsoft.com/en-us/cpp/assembler/masm/microsoft-macro-assembler-reference`. Accessed: 5 August 2018.

[17] Overview of x64 calling conventions. [Online], November 2016. URL `https://docs.microsoft.com/en-us/cpp/build/overview-of-x64-calling-conventions`. Accessed: 5 August 2018.

[18] DontUseInlineAsm. [Online], April 2016. URL `https://gcc.gnu.org/wiki/DontUseInlineAsm`. Accessed: 3 August 2018.

[19] M. A. N. Abrishamchi, A. H. Abdullah, A. David Cheok, and K. S. Bielawski. Side channel attacks on smart home systems: A short overview. In *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, pages 8144–8149, Oct 2017. doi: 10.1109/IECON.2017.8217429.

[20] C. M. Adams and H. Meijer. Security-related comments regarding McEliece's public-key cryptosystem. *IEEE Transactions on Information Theory*, 35(2):454–455, March 1989. ISSN 0018-9448. doi: 10.1109/18.32140.

[21] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-70431-5.

[22] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM. doi: 10.1145/800028.808479. URL `http://doi.acm.org/10.1145/800028.808479`.

[23] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. Innovative technology for CPU based attestation and sealing, August 2013. URL `https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing`.

[24] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *OLS*, 2009.

[25] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS

’08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-88312-8. doi: 10.1007/978-3-540-88313-5_22. URL http://dx.doi.org/10.1007/978-3-540-88313-5_22.

[26] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’14, pages 165–178, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535839. URL http://doi.acm.org/10.1145/2535838.2535839.

[27] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’14, pages 165–178, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535839. URL http://doi.acm.org/10.1145/2535838.2535839.

[28] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. *SIGPLAN Not.*, 49(1):165–178, January 2014. ISSN 0362-1340. doi: 10.1145/2578855.2535839. URL http://doi.acm.org/10.1145/2578855.2535839.

[29] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. Secure multiparty computation from SGX. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security*, pages 477–497, Cham, 2017. Springer International Publishing. ISBN 978-3-319-70972-7.

[30] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS ’68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM. doi: 10.1145/1468075.1468121. URL http://doi.acm.org/10.1145/1468075.1468121.

[31] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems*, 33(3):1–26, 2015. ISSN 07342071. doi: 10.1145/2799647. URL http://dl.acm.org/citation.cfm?doid=2818727.2799647.

[32] D Bell and Lj J LaPadula. Secure computer systems: Mathematical foundations. *Data Base*, 1(MTR-2547 Vol. I):513–523, 1973. doi: 10.3233/JCS-1996-42-308. URL http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=AD0770768%5Cnhttp://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=AD0770768.

[33] Steven M. Bellovin. Problem areas for the ip security protocols. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography*

- *Volume 6*, SSYM'96, pages 21–21, Berkeley, CA, USA, 1996. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1267569.1267590`.

[34] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6632 LNCS, pages 169–188, 2011. ISBN 9783642204647. doi: 10.1007/978-3-642-20465-4_11.

[35] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, 2005. URL `https://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[36] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, pages 513–525, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69528-8.

[37] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5628 LNCS, pages 325–343, 2009. ISBN 3642035485. doi: 10.1007/978-3-642-03549-4_20.

[38] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 37–51, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69053-5.

[39] Leonardo Borges. Program optimization through loop vectorization. [Online], January 2014. URL `https://software.intel.com/en-us/articles/program-optimization-through-loop-vectorization`. Accessed: 11 August 2018.

[40] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. DR.SGX: hardening SGX enclaves against cache attacks with data location randomization. *CoRR*, abs/1709.09917, 2017. URL `http://arxiv.org/abs/1709.09917`.

[41] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. Cryptology ePrint Archive, Report 2006/052, 2006. URL `https://eprint.iacr.org/2006/052`.

[42] Peter Bright. Intel's SGX blown wide open by, you guessed it, a speculative execution attack. [Online], August 2018. URL `https://arstechnica.com/gadgets/2018/08/intels-sgx-blown-wide-open-by-you-guessed-it-a-speculative-execution-attack/`. Accessed: 18 August 2018.

[43] Derek Bruening. DynamoRIO. [Online], 2009. URL `https://github.com/DynamoRIO/dynamorio`. Accessed: 1 October 2017.

[44] Derek Bruening. Dr. Memory: the memory debugger. [Online], 2010. URL `https://github.com/DynamoRIO/drmemory`. Accessed: 1 October 2017.

[45] Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 213–223, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8. URL `http://dl.acm.org/citation.cfm?id=2190025.2190067`.

[46] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X. URL `http://dl.acm.org/citation.cfm?id=776261.776290`.

[47] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Cambridge, MA, USA, 2004. AAI0807735.

[48] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD, 2018. USENIX Association. ISBN 978-1-931971-46-1. URL `https://www.usenix.org/conference/usenixsecurity18/presentation/bulck`.

[49] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B. Lee, Haibo Chen, and XiaoFeng Wang. Leveraging hardware transactional memory for cache side-channel defenses. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 601–608, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5576-6. doi: 10.1145/3196494.3196501. URL `http://doi.acm.org/10.1145/3196494.3196501`.

[50] S.K. Chin and S.B. Older. *Access Control, Security, and Trust: A Logical Approach*. Chapman & Hall/CRC Cryptography and Network Security Series. CRC Press, 2010. ISBN 9781439894637. URL `https://books.google.com/books?id=gxjSBQAAQBAJ`.

[51] Kai-Min Chung and Rafael Pass. A simple ORAM. Cryptology ePrint Archive, Report 2013/243, 2013. URL `https://eprint.iacr.org/2013/243`.

[52] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 62–81, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-45608-8.

[53] Catalin Cimpanu. New Intel CPU cache architecture boosts protection against side-channel attacks. [Online], July 2017. URL `https://www.bleepingcomputer.com/news/security/new-intel-cpu-cache-architecture-boosts-protection-against-side-channel-attacks/`. Accessed: 13 August 2018.

[54] Ellis Cohen. Information transmission in computational systems. *SIGOPS Oper. Syst. Rev.*, 11(5):133–139, November 1977. ISSN 0163-5980. doi: 10.1145/1067625.806556. URL `http://doi.acm.org/10.1145/1067625.806556`.

[55] Ellis Cohen. Information transmission in computational systems. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77, pages 133–139, New York, NY, USA, 1977. ACM. doi: 10.1145/800214.806556. URL `http://doi.acm.org/10.1145/800214.806556`.

[56] Scott D. Constable, Yuzhe Tang, Shuang Wang, Xiaoqian Jiang, and Steve Chapin. Privacy-preserving gwas analysis on federated genomic datasets. *BMC Medical Informatics and Decision Making*, 15(5):S2, Dec 2015. ISSN 1472-6947. doi: 10.1186/1472-6947-15-S5-S2. URL `https://doi.org/10.1186/1472-6947-15-S5-S2`.

[57] B. Jack Copeland. The Church-Turing thesis. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2017 edition, 2017. URL `https://plato.stanford.edu/archives/win2017/entries/church-turing/`.

[58] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual—combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c*, 2013. No. 325462-048.

[59] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. `https://eprint.iacr.org/2016/086`.

[60] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, January 1967. ISSN 0018-9448. doi: 10.1109/TIT.1967.1053964.

[61] Martin Davis. *Computability & Unsolvability*. Dover Books on Computer Science Series. Dover Publications, 1982. ISBN 9780486614717.

[62] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998. doi: 10.1017/CBO9780511663079.

[63] Christian DeLozier, Richard Eisenberg, Santosh Nagarakatte, Peter-Michael Osera, Milo M.K. Martin, and Steve Zdancewic. Ironclad C++: A library-augmented type-safe subset of C++. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 287–304, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509550. URL `http://doi.acm.org/10.1145/2509136.2509550`.

[64] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977. ISSN 0001-0782. doi: 10.1145/359636.359712. URL `http://doi.acm.org/10.1145/359636.359712`.

[65] Jonas Devlieghere. Guaranteed copy elision. [Online], November 2016. URL `https://jonasdevlieghere.com/guaranteed-copy-elision/`. Accessed: 13 August 2018.

[66] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 2006. ISSN 0018-9448. doi: 10.1109/TIT.1976.1055638. URL `http://dx.doi.org/10.1109/TIT.1976.1055638`.

[67] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. Shielding software from privileged side-channel attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1441–1458, Baltimore, MD, 2018. USENIX Association. ISBN 978-1-931971-46-1. URL `https://www.usenix.org/conference/usenixsecurity18/presentation/dong`.

[68] Vijay D'Silva, Mathias Payer, and Dawn Song. The correctness-security gap in compiler optimization. In *Proceedings of the 2015 IEEE Security and Privacy Workshops*, SPW '15, pages 73–87, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-9933-0. doi: 10.1109/SPW.2015.33. URL `http://dx.doi.org/10.1109/SPW.2015.33`.

[69] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Third Conference on Theory of Cryptography*, TCC'06, pages 265–284, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-32731-2, 978-3-540-32731-8. doi: 10.1007/11681878_14. URL `http://dx.doi.org/10.1007/11681878_14`.

[70] Morris J. Dworkin, Elaine B. Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E Roback, and James F. Dray Jr. Advanced encryption standard (AES). NIST FIPS 197, National Institute of Standards and Technology, November 2001.

[71] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, STC '12, pages 3–8, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1662-0. doi: 10.1145/2382536.2382540. URL `http://doi.acm.org/10.1145/2382536.2382540`.

[72] Anders Fogh. Cache side channel attacks: CPU design as a security problem. Presented in *Hack in the Box*, May 2016. URL `https://conference.hitb.org/hitbsecconf2016ams/sessions/cache-side-channel-attacks-cpu-design-as-a-security-problem/`.

[73] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. SGX-Lapd: Thwarting controlled side channel attacks via enclave verifiable page faults. In Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 357–380, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66332-6.

[74] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. Sgx-lapd: Thwarting controlled side channel attacks via enclave verifiable page faults. In *RAID*, 2017.

[75] Tal Garfinkel, Ben Pfaff, Jim Chow, and Mendel Rosenblum. Data lifetime is a systems problem. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*,

EW 11, New York, NY, USA, 2004. ACM. doi: 10.1145/1133572.1133599. URL `http://doi.acm.org/10.1145/1133572.1133599`.

[76] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, April 1982. doi: 10.1109/SP.1982.10014.

[77] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *1984 IEEE Symposium on Security and Privacy*, pages 75–75, April 1984. doi: 10.1109/SP.1984.10019.

[78] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 182–194, New York, NY, USA, 1987. ACM. ISBN 0-89791-221-7. doi: 10.1145/28395.28416. URL `http://doi.acm.org/10.1145/28395.28416`.

[79] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, May 1996. ISSN 0004-5411. doi: 10.1145/233551.233553. URL `http://doi.acm.org/10.1145/233551.233553`.

[80] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, pages 95–100, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1004-8. doi: 10.1145/2046660.2046680. URL `http://doi.acm.org/10.1145/2046660.2046680`.

[81] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321, Cham, 2016. Springer International Publishing. ISBN 978-3-319-40667-1.

[82] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299, Cham, 2016. Springer International Publishing. ISBN 978-3-319-40667-1.

[83] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 490–505, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4402-1. doi: 10.1109/SP.2011.22. URL `https://doi.org/10.1109/SP.2011.22`.

[84] Chris Hall, Ian Goldberg, and Bruce Schneier. Reaction attacks against several public-key cryptosystems. In Vijay Varadharajan and Yi Mu, editors, *Information and Communication Security*, pages 2–12, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-47942-0.

[85] Howard Hinnant. Allocator boilerplate. [Online], August 2016. URL `https://howardhinnant.github.io/allocator_boilerplate.html`. Accessed: 12 August 2018.

[86] W. M. Hu. Lattice scheduling and covert channels. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 52–61, May 1992. doi: 10.1109/RISP.1992.213271.

[87] ISO/IEC. ISO/IEC 9899:1999 programming languages – C. Standard, International Organization for Standardization (ISO), Geneva, Switzerland, December 1999. URL `https://www.iso.org/standard/29237.html`.

[88] ISO/IEC. Programming Language C++ (C++11 final draft). Working draft N3242, International Standards Organization (ISO), Geneva, Switzerland, 2011. URL `https://isocpp.org/std/the-standard`.

[89] ISO/IEC. ISO/IEC 9899:2011 programming languages – C. Working draft N1570, International Organization for Standardization (ISO), Geneva, Switzerland, December 2011.

[90] ISO/IEC. Programming Language C++ (C++17 final draft). Working draft N4659, International Organization for Standardization (ISO), Geneva, Switzerland, March 2017. URL `https://isocpp.org/std/the-standard`.

[91] Peter Kairouz, Sewoong Oh, and Pramod Viswanath. Secure Multi-party Differential Privacy. In C Cortes, N D Lawrence, D D Lee, M Sugiyama, and R Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2008–2016. Curran Associates, Inc., 2015. URL `http://papers.nips.cc/paper/6004-secure-multi-party-differential-privacy.pdf`.

[92] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. White paper, Advanced Micro Devices, Inc., April 2016.

[93] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 506–525, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-45608-8.

[94] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In Jean-Jacques Quisquater, Yves Deswarte, Catherine Meadows, and Dieter Gollmann, editors, *Computer Security — ESORICS 98*, pages 97–110, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. ISBN 978-3-540-49784-4.

[95] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48405-9.

[96] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre

attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018. URL `http://arxiv.org/abs/1801.01203`.

[97] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-68697-2.

[98] Johannes Koskinen. Metaprogramming in C++. [Online Lecture Notes], March 2004. URL `http://staff.ustc.edu.cn/~xyfeng/teaching/FOPL/lectureNotes/MetaprogrammingCpp.pdf`. Accessed: 19 August 2018.

[99] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10): 613–615, October 1973. ISSN 0001-0782. doi: 10.1145/362375.362389. URL `http://doi.acm.org/10.1145/362375.362389`.

[100] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349320. URL `http://doi.acm.org/10.1145/349299.349320`.

[101] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, April 1985. ISSN 0018-9340. doi: 10.1109/TC.1985.5009385.

[102] Peng Li and Steve Zdancewic. Encoding information flow in haskell. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, CSFW '06, pages 16–, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2615-2. doi: 10.1109/CSFW.2006.13. URL `https://doi.org/10.1109/CSFW.2006.13`.

[103] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, abs/1801.01207, 2018. URL `http://arxiv.org/abs/1801.01207`.

[104] C. Liu, M. Hicks, and E. Shi. Memory trace oblivious program execution. In *2013 IEEE 26th Computer Security Foundations Symposium*, pages 51–65, June 2013. doi: 10.1109/CSF.2013.11.

[105] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 87–101, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694385. URL `http://doi.acm.org/10.1145/2694344.2694385`.

[106] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 359–376, Washington, DC, USA,

2015. IEEE Computer Society. ISBN 978-1-4673-6949-7. doi: 10.1109/SP.2015.29. URL https://doi.org/10.1109/SP.2015.29.

[107] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, May 2015. doi: 10.1109/SP.2015.43.

[108] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, March 2016. doi: 10.1109/HPCA.2016.7446082.

[109] Sahar Mazloom and S. Dov Gordon. Differentially private access patterns in secure computation. Cryptology ePrint Archive, Report 2017/1016, 2017. https://eprint.iacr.org/2017/1016.

[110] D. McCullough. Specifications for multi-level security and a hook-up. In *1987 IEEE Symposium on Security and Privacy*, pages 161–161, April 1987. doi: 10.1109/SP.1987.10009.

[111] R. J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44:114–116, January 1978.

[112] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2118-1. doi: 10.1145/2487726.2488368. URL http://doi.acm.org/10.1145/2487726.2488368.

[113] J. McLean. Security models and information flow. In *Proceedings. 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 180–187, May 1990. doi: 10.1109/RISP.1990.63849.

[114] J. McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, Jan 1996. ISSN 0098-5589. doi: 10.1109/32.481534.

[115] John McLean. Proving noninterference and functional correctness using traces. *J. Comput. Secur.*, 1(1):37–57, January 1992. ISSN 0926-227X. URL http://dl.acm.org/citation.cfm?id=2699855.2699858.

[116] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 69–90. Springer, 2017. doi: 10.1007/978-3-319-66787-4_4. URL https://doi.org/10.1007/978-3-319-66787-4_4.

[117] Jonathan Müller. AllocatorAwareContainer: Introduction and pitfalls of propagate_on_container_XXX defaults. [Online], October 2015. URL `https://foonathan.net/blog/2015/10/05/allocatorawarecontainer-propagation-pitfalls.html`. Accessed: 12 August 2018.

[118] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No.98CB36186)*, pages 186–197, May 1998. doi: 10.1109/SECPRI.1998.674834.

[119] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM. ISBN 1-58113-095-3. doi: 10.1145/292540.292561. URL `http://doi.acm.org/10.1145/292540.292561`.

[120] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. pages 129–142, 1997. doi: 10.1145/268998.266669. URL `http://doi.acm.org/10.1145/268998.266669`.

[121] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on aes. In *Proceedings of the 13th International Conference on Selected Areas in Cryptography*, SAC'06, pages 147–162, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-74461-0. URL `http://dl.acm.org/citation.cfm?id=1756516.1756531`.

[122] Michael Norrish. A formal semantics for C++. Technical report, November 2007. Version 293.

[123] Brett Nuckles. Laptop privacy filters: What to look for and why you need one. [Online]. URL `https://www.businessnewsdaily.com/10859-laptop-privacy-filters-buying-advice.html`. Accessed: 21 August 2018.

[124] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 619–636, Austin, TX, 2016. USENIX Association. ISBN 978-1-931971-32-4. URL `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/ohrimenko`.

[125] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 227–240, Boston, MA, 2018. USENIX Association. ISBN 978-1-931971-44-7. URL `https://www.usenix.org/conference/atc18/presentation/oleksenko`.

[126] Peter-Michael Osera, Richard Eisenberg, Christian DeLozier, Santosh Nagarakatte, S Zdancewic, and Milo M K Martin. Core ironclad. Technical Report MS-CIS-13-06, January 2013.

[127] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-32648-9.

[128] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive, Report 2002/169, 2002. `https://eprint.iacr.org/2002/169`.

[129] Kevin Parrish. Microsoft's latest Windows 10 patch will address spectre variant 2 CPU flaw. [Online], March 2018. URL `https://www.digitaltrends.com/computing/microsoft-windows-patch-skylake-spectre-variant-2/`. Accessed: 4 August 2018.

[130] Martin Pettai and Peeter Laud. Combining differential privacy and secure multiparty computation. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 421–430, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3682-6. doi: 10.1145/2818000.2818027. URL `http://doi.acm.org/10.1145/2818000.2818027`.

[131] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, January 2003. ISSN 0164-0925. doi: 10.1145/596980.596983. URL `http://doi.acm.org/10.1145/596980.596983`.

[132] Aaron Pressman. Why your web browser may be most vulnerable to spectre and what to do about it. [Online], January 2018. URL `http://fortune.com/2018/01/05/spectre-safari-chrome-firefox-internet-explorer/`. Accessed: 4 August 2018.

[133] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, Washington, D.C., 2015. USENIX Association. ISBN 978-1-931971-232. URL `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane`.

[134] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978. ISSN 0001-0782. doi: 10.1145/359340.359342. URL `http://doi.acm.org/10.1145/359340.359342`.

[135] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003. ISSN 0733-8716. doi: 10.1109/JSAC.2002.806121.

[136] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace: Oblivious memory primitives from Intel SGX. Cryptology ePrint Archive, Report 2017/549, 2017. URL `https://eprint.iacr.org/2017/549`.

[137] Stephen Schmidt. Introducing s2n, a New Open Source TLS Implementation. [Online], 2015. URL `https://aws.amazon.com/blogs/security/introducing-s2n-a-new-open-source-tls-implementation/`. Accessed: 21 January 2018.

[138] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54, May 2015. doi: 10.1109/SP.2015.10.

[139] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read arbitrary memory over network, July 2018. URL `https://arxiv.org/pdf/1807.10535`.

[140] Robert C. Seacord. *The CERT®C Coding Standard, Second Edition: 98 Rules for Developing Safe, Reliable, and Secure Systems.* Addison-Wesley Professional, 2nd edition, 2014. ISBN 0321984048, 9780321984043.

[141] Elaine Shi, T. H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((log\,N)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, pages 197–214, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-25385-0.

[142] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: eradicating controlled-channel attacks against enclave programs. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017.* The Internet Society, 2017. URL `https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/t-sgx-eradicating-controlled-channel-attacks-against-enclave-programs/`.

[143] Vincent Simonet and Inria Rocquencourt. Flow caml in a nutshell. In *Proceedings of the first APPSEM-II workshop*, pages 152–165, 2003.

[144] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1169–1184, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813608. URL `http://doi.acm.org/10.1145/2810103.2813608`.

[145] Geoffrey Smith. Principles of secure information flow analysis. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, pages 291–307, Boston, MA, 2007. Springer US. ISBN 978-0-387-44599-1.

[146] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 355–364, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3. doi: 10.1145/268946.268975. URL `http://doi.acm.org/10.1145/268946.268975`.

[147] Peter Snyder. Yao's garbled circuits: Recent directions and implementations. [Online]. URL `https://www.cs.uic.edu/pub/Bits/PeterSnyder/Peter_Snyder_-_Garbled_Circuits_WCP_2_column.pdf`. Accessed: 5 September 2018.

[148] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard. Systematic classification of side-channel attacks: A case study for mobile devices. *IEEE Communications Surveys Tutorials*, 20(1):465–488, Firstquarter 2018. ISSN 1553-877X. doi: 10.1109/COMST.2017.2779824.

[149] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. *CoRR*, abs/1106.3652, 2011. URL `http://arxiv.org/abs/1106.3652`.

[150] 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 *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 299–310, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516660. URL `http://doi.acm.org/10.1145/2508859.2516660`.

[151] Becky Stern. Laptop compubody sock. [Online]. URL `https://www.instructables.com/id/Laptop-Compubody-Sock/`. Accessed: 21 August 2018.

[152] Bjarne Stroustrup. Bjarne Stroustrup's FAQ. [Online]. URL `http://www.stroustrup.com/bs_faq.html`. Accessed: 9 April 2018.

[153] Bjarne Stroustrup. Parameterized Types for C++. In *Proc. USENIX C++ Conference*, pages 1–18, Denver, CO, October 1988.

[154] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013. ISBN 0321563840, 9780321563842.

[155] Bjarne Stroustrup. Concepts: The Future of Generic Programming or How to design good concepts and use them well. Technical Report P0557r1, Morgan Stanely and Columbia University, 2017. URL `http://www.stroustrup.com/good_concepts.pdf`.

[156] Yulei Sui and Jingling Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 265–266, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4241-4. doi: 10.1145/2892208.2892235. URL `http://doi.acm.org/10.1145/2892208.2892235`.

[157] Lee Thomason, Yves Berquin, and Andrew Ellerton. TinyXML-2. [Online]. URL `https://github.com/leethomason/tinyxml2`. Accessed: 21 January 2018.

[158] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Secretly monopolizing the CPU without superuser privileges. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 17:1–17:18, Berkeley, CA, USA, 2007. USENIX Association. ISBN 111-333-5555-77-9. URL `http://dl.acm.org/citation.cfm?id=1362903.1362920`.

[159] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of des implemented on computers with cache. In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems* -

CHES 2003, pages 62–76, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45238-6.

[160] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proceedings 10th Computer Security Foundations Workshop*, pages 156–168, June 1997. doi: 10.1109/CSFW.1997.596807.

[161] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proceedings. 11th IEEE Computer Security Foundations Workshop (Cat. No.98TB100238)*, pages 34–43, June 1998. doi: 10.1109/CSFW.1998.683153.

[162] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, pages 607–621, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-68517-3.

[163] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996. ISSN 0926-227X. URL `http://dl.acm.org/citation.cfm?id=353629.353648`.

[164] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 215–226, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660314. URL `http://doi.acm.org/10.1145/2660267.2660314`.

[165] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 473–482, Dec 2006. doi: 10.1109/ACSAC.2006.20.

[166] Tom Warren. Intel processors are being redesigned to protect against Spectre. [Online], March 2018. URL `www.theverge.com/2018/3/15/17123610/intel-new-processors-protection-spectre-vulnerability`. Accessed: 4 August 2018.

[167] David A. Wheeler. *Secure Programming HOWTO*. v3.72 edition, September 2015. URL `https://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.pdf`.

[168] Wikipedia contributors. Chi-squared test — Wikipedia, the free encyclopedia. [Online], 2018. URL `https://en.wikipedia.org/w/index.php?title=Chi-squared_test&oldid=857915207`. Accessed: 5 September 2018.

[169] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: A parallel oblivious file system. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 977–988, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382299. URL `http://doi.acm.org/10.1145/2382196.2382299`.

[170] J. T. Wittbold and D. M. Johnson. Information flow in nondeterministic systems. In *Proceedings. 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 144–161, May 1990. doi: 10.1109/RISP.1990.63846.

[171] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 640–656, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4673-6949-7. doi: 10.1109/SP.2015.45. URL `https://doi.org/10.1109/SP.2015.45`.

[172] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society. doi: 10.1109/SFCS.1982.88. URL `https://doi.org/10.1109/SFCS.1982.88`.

[173] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, 1986. ISBN 0-8186-0740-8. doi: 10.1109/SFCS.1986.25. URL `http://ieeexplore.ieee.org/document/4568207/`.

[174] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, 2014. USENIX Association. ISBN 978-1-931971-15-7. URL `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom`.

[175] Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. URL `https://eprint.iacr.org/2015/1153`.

[176] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, Boston, MA, 2017. USENIX Association. ISBN 978-1-931971-37-9. URL `https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng`.

[177] Guy Zyskind, Oz Nathan, and Alex Pentland. Enigma: Decentralized computation platform with guaranteed privacy. *CoRR*, abs/1506.03471, 2015. URL `http://arxiv.org/abs/1506.03471`.

Index

Vita

Scott Constable was born in Rochester, New York. He received his Bachelor of Arts degree in Mathematics from Ithaca College in Ithaca, New York in May of 2012. He received his Master of Science degree in Computer Science from Syracuse University in Syracuse, New York in May of 2014. He received his Doctor of Philosophy (Ph.D.) degree in Computer and Information Science and Engineering from Syracuse University in December of 2018.