DEFINING, ENFORCING, AND PROVING EXTENSIBLE LANGUAGE-BASED SECURITY

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Silei Ren

August 2026

© 2026 Silei Ren

ALL RIGHTS RESERVED

DEFINING, ENFORCING, AND PROVING EXTENSIBLE LANGUAGE-BASED SECURITY

Silei Ren, Ph.D.

Cornell University 2026

Language-based information flow control (IFC) enforces security properties in decentralized systems by representing security policies as types. Despite its rich theoretical foundation, IFC is not widely used in practice, in part because of insufficient support for *extensibility*: the ability to enforce security with partial knowledge of trust and in the presence of modules built by untrusted toolchains.

The first work achieves extensibility through a novel IFC label algebra that ensures sound reasoning even as new principals and trust relations are introduced in the future. The expressive label algebra supports *asymmetric delegation*, where principals can partially trust each other for confidentiality, integrity, or both. Our framework also supports downgrading of information and ensures its safety through nonmalleable information flow (NMIF). To demonstrate the practicality of our framework, we design and implement a static NMIF checking algorithm and an efficient label inference procedure that supports bounded label polymorphism, allowing users to write code generic over labels.

The second work proposes a novel formal definition of extensibility as *type-confusion security*, a simulation-based security condition relating well-typed and ill-typed attackers. We formally prove that type-confusion security for IFC can be achieved through a combination of static and dynamic type checking. This work serves as the theoretical foundation for the SCIF compiler, a compiler for a smart-contract language that enforces IFC in open-world settings.

BIOGRAPHICAL SKETCH

Silei Ren (任思磊) was born in Beijing, China in 1996. He spent 6 years in the middle and high school affiliated with Renmin University of China from 2009 to 2015. He then attended Brown University, where he obtained a Bachelor of Science degree in mathematics and computer science and received a senior prize in computer science. After that, he joined Cornell University in 2020 to pursue a PhD in computer science.

For my parents Guiping Li (李桂萍) and Shengli Ren (任胜利).

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Andrew Myers. Andrew has been patient, encouraging, and generous with his trust throughout my PhD, especially in the early years, when I was still developing my research direction and skills. His qualities as a researcher—his breadth of knowledge, his skill in communicating ideas, and his insistence on research that will stand the test of time—have set a standard to which I aspire. I sincerely appreciate how he has shaped me as a researcher.

I would also like to thank my committee members Dexter Kozen, Nate Foster, and Ari Juels for their valuable feedback and support for my thesis research.

My collaborators Josh Acay, Suraaj Kanniwadi, and Gary Chen deserve special thanks for their contributions to my thesis research. Working with them has been not only productive, but also a great deal of fun. Needless to say, this thesis would not have been possible without their help and support.

I would also like to thank my friends and colleagues from the APL group, Rolph Recto, Yulun Yao, Ethan Cecchetti, Siqiu Yao, Haobin Ni, Drew Zagieboylo, Josh Gancher, Mae Milano, Owen Arden, Anitha Gollamudi, Stephanie Ma, Noah Schiff, Vivian Ding, and Charles Sherk. The Friday group meetings have always been productive, fun and invigorating.

More broadly, I would like to thank my friends from the CIS department, especially the PLDG community, the System Lab, and the CIS PL Lab: Kei Imada, Ali Farahbakhsh, Florian Suri-Payer, Austin Li, Anshuman Mohan, Sebastian Holler, Max Fan, Karuna Grewal, Oliver Daids, Vaibhav Mehta, Spencer Van Koevering, Mark Barbone, Aditya Senthilnathan, Jialu Bao, Goktug Saatcioglu, Hongbo Zhang, Elaine Yao, Alicia Yang, Ziyun Wei, Luming Tang, Qizhe Cai, Junxiong Wang, Java Ye. You made me feel that I was not walking alone, and I

have been happy to wake up every day, come to the lab, and see you all.

I would also like to thank my best friends outside Cornell, Ben Zhou, Bill Yu, Shaohui Liu, Joyce Wang, Zihe Wang, Yu Tang, Amanda Shen, Yunyi Zhang, and Shiliang Gao. Completing a PhD in a different country is not easy, and I am blessed to have friends who support me like family. Special thanks to my partner Yuxin Zhao and our cats Chou and Dan. Words cannot express how much more supported I have felt since you came into my life.

Finally, I would like to thank my parents Guiping Li and Shengli Ren for their unconditional support throughout my life. They have taught me to be curious, persistent, brave, humble, and confident. I would not have been where I am today without them. The work in this dissertation was partly funded by the National Science Foundation.

CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Contents . . . vii
List of Figures . . . ix

1 Introduction **1**
1.1 Technical Acknowledgments and Dissertation Outline . . . 5

2 Delegation in Information Flow Control **6**
2.1 A Case for Delegation . . . 9
2.1.1 Semi-Honest Attackers in Cryptography . . . 9
2.1.2 Modeling Security with Delegation . . . 10
2.1.3 Nonmalleable Information Flow . . . 10
2.2 Math Background . . . 12
2.2.1 Lattices . . . 12
2.2.2 Prime Filters . . . 13
2.2.3 Algebraic Semantics . . . 15
2.2.4 Lindenbaum–Tarski Algebra . . . 17
2.3 Semantic Framework . . . 19
2.3.1 The Lattice of Principals . . . 19
2.3.2 Delegation and Attackers . . . 21
2.3.3 Order Characterization and Succinctness . . . 27
2.3.4 Labels . . . 28
2.4 Security Hyperproperties and Delegation . . . 29
2.4.1 Noninterference and the Lattice of Information Flow . . . 31
2.4.2 Downgrading and Nonmalleable Information Flow . . . 32
2.5 Algorithms . . . 36
2.5.1 Acts-for Algorithm . . . 37
2.5.2 NMIF Algorithm . . . 40
2.6 Label Inference and Bounded Polymorphism . . . 46
2.6.1 Prior Approaches to Polymorphism in IFC Compilers . . . 46
2.6.2 Bounded Label Polymorphism . . . 48
2.6.3 Constraint Solver . . . 51
2.6.4 Implementation . . . 55
2.7 Related Work . . . 58
2.8 Conclusion and Future Directions . . . 59

3 Type-Confusion Security **60**
3.1 Motivation . . . 63
3.1.1 Confused Deputy Attacks . . . 63
3.1.2 Reentrancy Attacks . . . 67

3.1.3 Type Confusion and IFC Background . . . 69
3.1.4 The Simple Cases of Type Confusion . . . 73
3.2 Formalizing Type Confusion . . . 74
3.2.1 The Decentralized World Model . . . 74
3.2.2 Type-Confusion Security . . . 75
3.3 A Core Calculus for Type Confusion . . . 77
3.3.1 Syntax . . . 78
3.3.2 Attackers . . . 79
3.3.3 Dynamic Semantics . . . 80
3.3.4 Static Semantics . . . 82
3.3.5 Type Safety of λ_{tc} . . . 85
3.3.6 The Real World . . . 90
3.4 Type-Confusion Security . . . 94
3.4.1 Examples Revisited . . . 97
3.4.2 Base Type Confusion . . . 99
3.5 Proof of Type-Confusion Security . . . 100
3.5.1 Pointer Equality for Attacker Code . . . 101
3.5.2 Top-Level Simulator Construction . . . 102
3.5.3 Translation of Function Types . . . 103
3.5.4 Function Name Translation . . . 104
3.5.5 Store Name Translation . . . 104
3.5.6 Expression Translation . . . 104
3.5.7 Gadget: version selector . . . 105
3.5.8 Gadget: attacker function down-caster . . . 107
3.5.9 Gadget: Remote Reader and Writer . . . 109
3.5.10 Heap Setup . . . 110
3.5.11 Well-typed Translation . . . 110
3.5.12 Proof of Type Confusion . . . 111
3.6 Applications . . . 118
3.6.1 Confused Deputy Attacks . . . 118
3.6.2 Reentrancy Attack . . . 120
3.6.3 Noninterference . . . 121
3.7 Related Work and Discussion . . . 122
3.8 Conclusions . . . 126

4 Conclusion **127**

LIST OF FIGURES

2.1 Yao's Millionaires' problem in Viaduct [6]. The programmer must manually assign labels to hosts. 8
2.2 Yao's Millionaires' problem implemented with delegation. $\mathrm{Alice} \sqcup \mathrm{Bob}$ is shorthand for $\langle \mathrm{Alice} \wedge \mathrm{Bob}, \mathrm{Alice} \vee \mathrm{Bob} \rangle$. 8
2.3 Mutual delegation between Alice and Bob in the semi-honest MPC example collapses the four-point Alice–Bob principal lattice to a single authority level in the quotient lattice. 23
2.4 The lattice of labels $(\mathbb{L}, \leq)$ and the lattice of information flow $(\mathbb{L}, \sqsubseteq)$ share the same underlying set, but use a different ordering. The dotted line depicts labels with equal confidentiality and integrity: strictly above are compromised labels, and on or below are uncompromised labels. 29
2.5 Generated label variables and constraints for the polymorphic `average` example. 49
2.6 Syntax of label constraints. 51
2.7 Syntax of principal constraints. 51
2.8 Translating label constraints to principal constraints. 52
2.9 The running `average` example in the older host-parameterized Viaduct syntax. The old compiler materializes separate specialized copies of `average`, and adding `Chuck` forces every host label to mention an extra integrity component. 56

3.1 The classic Confused Deputy Attack (CDA) example, annotated with information flow labels. For simplicity, only function types and function call sites are annotated with pc (control flow) labels. 64
3.2 Visual illustration of the execution trace of a CDA attack under different enforcement mechanisms. The eager enforcement mechanism performs a dynamic type check upon receiving the function pointer, whereas the lazy enforcement mechanism checks the pointer when it is used (called). Without a centralized authorization engine, the eager enforcement mechanism requires an honest attacker, making it impractical in settings like blockchain. 66
3.3 Reentrancy attack against the compiler example. In addition to an external pc, functions are annotated with a bounding label that bounds the maximum effect during function execution. 67
3.4 The subtyping lattice of well-formed function types. The static type of untrusted pointer variables from the attacker is the rightmost type $(U \xrightarrow{U \triangleleft U} U)_U$ (blue). Type confusion occurs when the attacker uses pointers as a supertype (red). Type confusing the output type is not harmful because of covariance. Note that the label lattice does not need to be a 2-point lattice, and the figure uses a 2-point lattice for clarity of presentation. 71

3.5 Syntax of λ_{tc}. 77
3.6 Dynamic semantics of λ_{tc}. 81
3.7 Static semantics of λ_{tc}: well-formed programs, attacker configurations, and stores. 82
3.8 Static semantics of λ_{tc}: expression typing. 83
3.9 Static semantics of λ_{tc}: subtyping. 84
3.10 Runtime checks in λ^r_{tc}. These rules replace APP-REMOTE and RETURN from λ_{tc}. 91
3.11 Static semantics of λ^r_{tc}. 92
3.12 Trace Erasure and Equivalence. 94
3.13 Simulator Construction 96
3.14 A stronger trace equivalence. 112

CHAPTER 1

INTRODUCTION

Over the past decades, computing systems have evolved from closed, monolithic programs executing on a single machine to decentralized architectures distributed across multiple servers, composed of third-party libraries, and spanning multiple programming languages. Defining and enforcing security in these open systems is inherently challenging, as it requires compositional reasoning over components that operate under heterogeneous trust boundaries. Moreover, each system exhibits a unique threat model, distinct security requirements, and varied static and dynamic assumptions. Consequently, while security enforcement mechanisms may share high-level principles, their practical implementations differ significantly across domains. For example, client-side ecosystems—such as Chrome extensions, Android applications, and VS Code extensions—are open systems where programs with heterogeneous trust interact through tightly regulated interfaces [11, 37, 64]. In these domains, security enforcement relies primarily on building sophisticated execution engines and language runtimes that actively sandbox and monitor API access [11, 37]. Conversely, cloud-native platforms like AWS Lambda and GitHub Actions provide highly open environments where user code—compiled with arbitrary static toolchains and executing on diverse dynamic runtimes—interacts with both other tenant code and the underlying platform infrastructure [7, 12]. Here, the focus of security enforcement shifts away from the language runtime and toward strict hardware-level isolation and both static and dynamic information flow control (IFC) across component boundaries [7, 59, 63]. Finally, blockchain smart contracts do not rely on a concrete centralized authority, allowing arbitrary code from mutually distrusting parties to interact at a low level, often directly as bytecode binaries [76, 102].

In this paradigm, security research primarily focuses on the robustness of underlying consensus protocols and Layer 2 designs [42, 76], while contract-level security relies on binary static analysis and manual auditing [34, 91, 99].

Despite these differences, security enforcement in decentralized systems repeatedly returns to two fundamental questions:

- Authorization policy: who is allowed to observe or modify which data?
- Information flow: what dependencies are allowed between data, modules, and principals?

In practice, however, these policies are often not made explicit at the language level. Instead, they are implicit in the design of the system and in the mechanisms used to enforce them: capabilities, sandboxing, cryptography, and hardware isolation [7, 16, 32, 41]. This mismatch makes security harder to reason about compositionally. When source-level policies are absent, information flow analyses must reconstruct them from lower-level artifacts such as intermediate representations, control-flow graphs, or binaries [3, 78, 91, 99]. Similarly, runtime monitoring becomes necessary when information that would support static reasoning is unavailable, implicit, or scattered across system boundaries. Thus, many systems already perform information flow reasoning, but they do so indirectly and repeatedly, without a common abstraction for stating the intended policy.

This suggests that security policies should be first-class citizens in programming languages for decentralized systems. The analogy is with Rust: Rust makes memory ownership a native programming abstraction, and uses it to enforce memory safety, a general property desirable in all low-level programs [57, 67]. Our goal is analogous: a lightweight, usable type system that enforces information flow properties in decentralized systems. Such a language should represent

security policies as types and use them to statically enforce information flow properties about confidentiality and integrity for both data and control flow. For programmers, representing security policies as types provides a clean separation between specification and enforcement: developers declaratively state security policies, while the compiler sanity-checks the policies and automatically chooses appropriate enforcement mechanisms. This separation allows decentralized systems to leverage the dependency analysis provided by the programming language to enforce system-specific security requirements.

The theory of language-based security and IFC is well developed, and many type systems have been proven to enforce security properties such as noninterference, nonmalleable information flow, and quantitative information flow [18, 23, 47, 58, 87]. Prototype compilers based on these type systems have also incorporated usability features such as syntactic sugar, type inference, polymorphism, and first-class functions [6, 9, 63, 75]. However, IFC type systems have not been widely adopted in practice, in part because they lack *extensibility*. In this thesis, we mainly consider two aspects of extensibility: first, we study sound and complete IFC enforcement in open systems, where principals do not have full knowledge of other principals, policies, or trust relations; second, we study sound and complete IFC security enforcement for modules in open systems that include legacy or adversarial components. In both cases, we begin by defining the semantics, then design enforcement mechanisms, and finally prove them correct with respect to those semantics.

In the first work, we aim to design a compiler whose information flow analysis remains sound as the space of principals, labels, and trust assumptions evolves. The first part of this thesis develops an algebraic semantic framework for IFC labels: instead of baking a particular label model into the language, we

identify the algebraic structure that labels must provide and state security properties in terms of that structure. The algebraic nature of the semantics ensures that information flow analysis does not depend on a fixed set of principals or trust assumptions, so the analysis remains sound as new principals and trust assumptions are added. As a parallel contribution, we use the framework to model *asymmetric delegation*, where principals may delegate confidentiality and integrity independently, and we show how such delegations interact with downgrading and nonmalleable information flow. We then derive static checking algorithms from the semantics and instantiate the design in Viaduct [6], adding concise syntax for trust assumptions and label inference with bounded label polymorphism. The resulting implementation is highly modular: it separates the syntax and semantics of labels, and the core information flow analysis works exclusively with the interpreted semantic domain of labels.

In the second work, we ask how much of this source-level reasoning can survive when typed code is deployed in an open world with legacy or adversarial components. The second part of this thesis studies this problem for IFC type systems with remote calls: trusted code is statically typed, but real-world attackers may ignore the type system or lie about security types. We formalize the required compatibility guarantee as *type-confusion security*, a real–ideal simulation property saying that an ill-typed real attacker has no more power than a well-typed ideal attacker. We give core calculi for decentralized programs with stateful principals, remote calls, and first-class function pointers. The ideal-world semantics assumes a well-typed program and enforces noninterference, security against confused deputy attacks, and security against reentrancy attacks. We prove type-confusion security by constructing a translation from each real-world ill-typed attacker to a well-typed ideal-world simulator. The proof

guides the implementation of closures in the SCIF compiler for a smart contract programming language that enforces information flow and control-flow integrity by construction.

1.1 Technical Acknowledgments and Dissertation Outline

The remainder of the thesis is organized as follows.

Chapter 2 is a lightly revised version of the conference paper [84]: Silei Ren, Coşku Acay, and Andrew C. Myers. An algebraic approach to asymmetric delegation and polymorphic label inference. In *European Symposium on Research in Computer Security (ESORICS)*, September 2025.

Chapter 3 is based on the preprint: Silei Ren, Suraaj Kanniwadi, Hanxi Chen, and Andrew Myers. Type-Confusion Security. January 2026.

Chapter 4 concludes the thesis and discusses future work.

I am deeply grateful to my coauthors Josh, Suraaj, Gary, and Andrew for their contributions to the work in this thesis. Any errors or omissions are my own.

CHAPTER 2

DELEGATION IN INFORMATION FLOW CONTROL

Information Flow Control (IFC) [58, 87] is a well-established approach for enforcing information security. Using *labels*, IFC systems specify fine-grained policies on information flow that can be fully or partly enforced through compile-time analysis. These policies articulate the confidentiality and integrity goals of IFC systems, which are *security properties* [58]: hyperproperties [24] that constrain the set of system behaviors.

The most prominent security property for IFC systems is *noninterference* [47], but it is too restrictive in practice. A major challenge for adopting language-based IFC is providing developers with expressive yet intuitive ways to specify their intended security policies. To capture more nuanced security policies, the expressiveness of IFC systems is enhanced by *downgrading mechanisms* such as *declassification* of confidential information and *endorsement* of untrusted information. Misuse of these mechanisms is further mitigated by enforcing *nonmalleable information flow* [18] (NMIF), a security property controlling downgrading.

Delegation is another common mechanism for specifying security. Delegation allows one principal to grant (delegate) power to another, expressing that the first principal trusts the second. Delegation can compactly represent important aspects of the system's security policy: when there is delegation between two principals, ensuring the delegator's security also necessitates enforcing the delegatee's security. Delegation is commonly supported not only in information flow control systems [8, 9, 71, 75], but also in a wide range of enforcement mechanisms, including access control [13, 31, 89] (where delegation is often referred to as a *principal* or *role hierarchy*), authorization logics [1, 4, 8, 54, 55], and capability systems [61, 66].

The expressive power of delegation can be increased through what we call *asymmetric delegation*: fine-grained delegation of either *confidentiality* or *integrity*. Intuitively, when a principal Alice delegates her confidentiality to another principal Bob, she allows Bob to observe all information visible to her. When Alice delegates her integrity to Bob, she trusts that all information accepted by Bob has not been maliciously modified. With asymmetric delegation, we can model security settings like the semi-honest trust assumption in cryptographic applications and the security setting of blockchains. In the semi-honest setting, principals trust each other to follow the protocol (trust each other with integrity), but do not trust each other with their secrets (i.e., not confidentiality). In the blockchain setting, principals do not trust each other to follow protocols, but all information is public: they effectively trust each other with respect to confidentiality.

While asymmetric delegation increases the expressive power of IFC systems, its precise role—particularly in the presence of downgrading—remains poorly understood. We address this gap by presenting a general and expressive semantic framework for IFC labels that formalizes both asymmetric delegation and its interaction with downgrading. Although prior work [9, 98] develops IFC systems that support certain forms of asymmetric delegation, these systems lack sound and complete NMIF enforcement. Building on our framework, we develop algorithms for verifying the associated semantic security properties.

Experience with language-based security highlights the importance of IFC label inference to reduce the burden on programmers [6, 75]. In addition, allowing programmers to write code that is generic with respect to labels enhances modularity and code reuse. We support such generic programming through an efficient label inference procedure that supports bounded label polymorphism.

To evaluate our approach, we update the label model of the Viaduct com-

```
1  host Alice : {A ∧ B←}
2  host Bob   : {B ∧ A←}
3
4  val a: {A ∧ B←} = Alice.input
5  val b: {B ∧ A←} = Bob.input
6  val w: {A ∧ B} = a > b
7
8  Alice.output(
9    declassify w to {A ∧ B←})
10 Bob.output(
11   declassify w to {B ∧ A←})
```

Figure 2.1: Yao's Millionaires' problem in Viaduct [6]. The programmer must manually assign labels to hosts.

```
1  host Alice, Bob
2  assume Alice = Bob for integrity
3
4  val a: {Alice} = Alice.input
5  val b: {Bob}   = Bob.input
6  val w: {Alice ⊔ Bob} = a > b
7
8  Alice.output(
9    declassify w to {Alice})
10 Bob.output(
11   declassify w to {Bob})
```

Figure 2.2: Yao's Millionaires' problem implemented with delegation. $\mathrm{Alice} \sqcup \mathrm{Bob}$ is shorthand for $\langle \mathrm{Alice} \wedge \mathrm{Bob}, \mathrm{Alice} \vee \mathrm{Bob} \rangle$.

piler [6] and extend its static information flow analysis. Our implementation features a more concise and modular syntax for specifying trust assumptions, as well as a more efficient label inference procedure.

- §2.1 motivates asymmetric delegation using a semi-honest secure multi-party computation (MPC) program.
- §2.2 reviews the mathematical background of order theory and logics necessary for our semantic framework.
- §2.3 presents our semantic framework for labels that captures asymmetric delegation.
- §2.4 studies the effects of asymmetric delegation on security properties using our novel semantic framework.
- §2.5 presents algorithms that statically enforce the security properties.
- §2.6 introduces an inference procedure supporting bounded label polymorphism.

2.1 A Case for Delegation

2.1.1 Semi-Honest Attackers in Cryptography

Asymmetric delegation can capture a wide variety of security settings. Already mentioned is the semi-honest threat model, widely studied in the cryptography literature [103]. In this model, principals correctly follow the protocol, but attempt to improperly learn other principals' secrets. Modeling the semi-honest setting in IFC systems remains a challenge. We first give an example of modeling semi-honest security in Viaduct [6], a state-of-the-art compiler that translates information flow policies to cryptographic protocols. We then illustrate how delegation improves usability and modularity.

Consider Yao's well-known Millionaires' Problem [103], where Alice and Bob wish to compare their wealth without revealing actual numbers. Figure 2.1 shows a Viaduct implementation. Lines 1–2 declare the hosts `Alice` and `Bob` and assign them information flow labels that capture the security assumptions. The hosts are assigned different and incomparable confidentiality labels (`A` for `Alice` and `B` for `Bob`) but the same integrity label (`A ∧ B`) to reflect the trust relation in the semi-honest model. Lines 4–5 gather input from the hosts; input from a host has the same label as that host. Line 6 stores the result of the comparison in `w`, which has a label following standard IFC rules: the result of a computation is more secret and less trusted than all of its inputs. Specifically, `w` has a confidentiality of `A ∧ B` since it is derived using secret data from both hosts, and has an integrity of `A ∧ B` since that is the integrity of both inputs. Finally, lines 8–11 output `w` to `Alice` and `Bob`, respectively. Note that sending `w` to `Alice` leaks information about `Bob`'s secret data (`b`), which violates noninterference. Viaduct requires an explicit **`declassify`** statement to indicate that information leakage is intentional.

2.1.2 Modeling Security with Delegation

Viaduct models security by encoding trust into labels, but this approach has problems. First, programmers must encode security assumptions by carefully crafting host labels, which becomes tricky in large systems with many assumptions. Second, this encoding pollutes the entire program. In fig. 2.1, every label annotation must acknowledge the semi-honest assumption by carrying around additional integrity (i.e., $\texttt{A}^{\leftarrow}$ or $\texttt{B}^{\leftarrow}$). Third, the encoding breaks modularity. For example, to add a new host `Chuck` to the program, we would need to edit every label annotation to carry an extra integrity component of $\texttt{C}^{\leftarrow}$, requiring changes throughout the program even though `Chuck` is not involved in this portion of the computation. Delegation addresses all of these issues.

Figure 2.2 implements Yao's Millionaires' Problem using delegation. Hosts are no longer assigned cryptic information flow labels; instead, line 2 directly states the security assumption: `Alice` and `Bob` trust each other for integrity. Variables have intuitive labels that do not need to repeat the semi-honest security assumption: input from `Alice` has label `Alice`. Finally, adding a new host `Chuck` requires no edits to existing code; we only need to add the following lines:[1]

```
1 host Chuck
2 assume Alice = Chuck for integrity
```

2.1.3 Nonmalleable Information Flow

Downgrading statements (**`declassify`** and **`endorse`**) deliberately violate noninterference, so their unrestricted use poses a threat to security. Prior work [74, 106] identifies cases where the attacker can exploit downgrading to gain undue in-

[1] In fact, we could even support separate compilation as the program need not be type-checked again: a program considered secure with fewer assumptions is secure with more assumptions.

fluence over the execution, and proposes *robust declassification* and *transparent endorsement* to limit such cases.

Robust declassification requires that untrusted data is not declassified, and transparent endorsement requires that secret data is not endorsed. NMIF combines these two restrictions, which are key to enabling the Viaduct compiler to securely instantiate programs with cryptography [6].

Here, "secret" and "trusted" are relative to a given attacker, and NMIF must hold for all attackers. In practice, the program cannot be type-checked separately for every possible attacker, so a conservative condition is enforced: downgraded data must be at least as trusted as it is secret. For our example program, this condition means `w` must have integrity stronger than or equal to its confidentiality. This condition is immediate in Viaduct since `w` has label $\langle$`Alice` $\wedge$ `Bob, Alice` $\wedge$ `Bob`$\rangle$ in fig. 2.1. On the other hand, the same variable `w` in fig. 2.2 has label $\langle$`Alice` $\wedge$ `Bob, Alice` $\vee$ `Bob`$\rangle$, which seemingly has weaker integrity than confidentiality (logically, `Alice` $\vee$ `Bob` does *not* imply `Alice` $\wedge$ `Bob`). However, $\mathrm{Alice} = \mathrm{Bob}$ for integrity, so this label is equivalent to $\langle$`Alice` $\wedge$ `Bob, Alice` $\wedge$ `Bob`$\rangle$ using the following derivation:

$$\mathrm{Alice} \vee \mathrm{Bob} = \mathrm{Alice} \vee \mathrm{Alice} = \mathrm{Alice} = \mathrm{Alice} \wedge \mathrm{Alice} = \mathrm{Alice} \wedge \mathrm{Bob}$$

Delegation necessitates equational reasoning under assumptions, and NMIF creates an interaction between confidentiality and integrity. The combination of these two features is what makes asymmetric delegation tricky: the cleaner syntax comes at the cost of additional technical complexity. The following sections tame this complexity by developing a semantic framework for labels, and algorithms that follow the semantics.

2.2 Math Background

The examples above showed that delegation can make syntactically different labels express the same effective trust assumptions. Before presenting our semantic framework for labels in §2.3, we review the necessary background in order theory and mathematical logic. Readers familiar with order theory and Heyting algebras may safely skip this section and refer back to it as needed.

While all the results here are standard, we choose to present the specialized versions of the more general theorems to make it easier to understand and apply in the context of information flow control.

2.2.1 Lattices

A *preorder* is a set equipped with a reflexive and transitive relation $\leq$. A *partial order* is a preorder satisfying antisymmetry. A *lattice* is a partial order in which every pair of elements has a greatest lower bound ($\wedge$) and a least upper bound ($\vee$).

A *bounded lattice* is a partial order with a greatest element $\top$, a least element $\bot$, and binary operations meet and join. The lattice is *distributive* when meet and join distribute over each other ($(x \wedge y) \vee z = (x \vee z) \wedge (y \vee z)$ and $(x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z)$). If L is any ordered set and $x \in L$, the *upward closure* of x is

$$\uparrow x = \{y \in L \mid x \leq y\}.$$

We will also refer to the *free distributive lattice* over a set of *generators*. This is the most general distributive lattice built from those generators using only $\wedge$, $\vee$, $\top$, and $\bot$, modulo the usual lattice equations. One notable result is that each element of the free distributive lattice can be written in a unique normal form as a join of meets of generators.

A *Heyting algebra* is a bounded distributive lattice equipped with a binary operation $x \to y$, called the *relative pseudocomplement*, satisfying:

$$x \wedge z \leq y \quad \text{iff} \quad z \leq (x \to y).$$

Equivalence relations on a lattice are congruences when they preserve the lattice structure, and congruences induce quotient lattices.

Definition 2.2.1 (Congruence and Quotient Lattice)**.** An equivalence relation $\equiv$ on a lattice L is a *congruence* when:

$$\forall x_1, x_2, y_1, y_2 \in L . ((x_1 \equiv x_2) \wedge (y_1 \equiv y_2)) \implies ((x_1 \wedge y_1 \equiv x_2 \wedge y_2) \wedge (x_1 \vee y_1 \equiv x_2 \vee y_2)).$$

For any congruence $\equiv$, the *quotient lattice* $L/\equiv$ is the lattice whose elements are equivalence classes $[x]_\equiv = \{y \in L \mid y \equiv x\}$, with operations $[x]_\equiv \wedge [y]_\equiv = [x \wedge y]_\equiv$ and $[x]_\equiv \vee [y]_\equiv = [x \vee y]_\equiv$.

2.2.2 Prime Filters

Definition 2.2.2 (Filters, Ideals, and Prime Filters)**.** A subset $F \subseteq L$ is a *filter* when:

$$\forall x, y \in L . \top \in F \wedge ((x \in F \wedge x \leq y) \implies y \in F) \wedge ((x \in F \wedge y \in F) \implies x \wedge y \in F).$$

Dually, a subset $I \subseteq L$ is an *ideal* when:

$$\forall x, y \in L . \bot \in I \wedge ((x \in I \wedge y \leq x) \implies y \in I) \wedge ((x \in I \wedge y \in I) \implies x \vee y \in I).$$

A filter is *prime* when

$$\forall x, y \in L . (x \vee y \in F) \implies (x \in F \vee y \in F).$$

For intuition, we can consider the simple case where the lattice is complete (i.e., all subsets of lattice have a least upper bound and a greatest lower bound).

In such a case, the prime filters are exactly the upward closures of the meets of generators.

An important property of distributive lattices/Heyting algebra is that all filters and ideals can be separated by prime filters. Alternatively, a filter and an ideal that do not already contradict each other can be extended to a prime viewpoint that keeps them separated.

Theorem 2.2.3 (Prime-Filter Theorem)**.** *Let L be a distributive lattice, let $F_0 \subseteq L$ be a filter, and let $I_0 \subseteq L$ be an ideal. If $F_0 \cap I_0 = \emptyset$, then there exists a prime filter F such that*

$$F_0 \subseteq F \quad \textit{and} \quad F \cap I_0 = \emptyset.$$

A reason why prime filters are important is that distributive lattices contain enough information to recover the order. There are many lemmas and theorems that describe the same idea, and we state the following version of the Stone representation theorem that provides the best interpretation for our later use.

Theorem 2.2.4 (Stone Representation, Simplified)**.** *Let L be a distributive lattice, and let $\mathcal{F}$ be the set of prime filters of L. Then each element of L is uniquely determined by the set of prime filters that contain it:*

$$\forall x, y \in L \centerdot x \neq y \implies \{F \in \mathcal{F} \mid x \in F\} \neq \{F \in \mathcal{F} \mid y \in F\}.$$

Essentially this means that the order of a distributive lattice can be characterized by the set of prime filters that contain each element. specifically, if $\widehat{x} = \{F \in \mathcal{F} \mid x \in F\}$, then $x \leq y$ exactly when $\widehat{x} \subseteq \widehat{y}$. The representation therefore lets us reason about algebraic order by reasoning about which prime-filter viewpoints see each element as true. This is the aspect of Stone-style representation used later when modeling attackers as sets of authority levels.

2.2.3 Algebraic Semantics

Prior definitions of authorization logics [1, 2, 8, 43–46, 55] have been based on extensions of intuitionistic logics. We focus on propositional intuitionistic logic, which is the fragment of intuitionistic logic without quantifiers. This will serve as our interpretation of information flow labels as security policies later on in our semantic framework. More specifically, it isolates the algebraic structure that underlies several later authorization calculi. For example, Abadi's Dependency Core Calculus (DCC) [1] and related authorization logics add a principal-indexed protection modality, often written *says*, on top of an acts-for order. In a judgment or type of the form "A says t," our concern is the authority structure of the index A: its order, lattice operations, and the effect of delegation on it. Accordingly, we do *not* define a full authorization logic or a term calculus with a *says* connective, nor do we study the proof theory or operational behavior of that modality itself. Instead, we study the semantic domain over which such modalities are indexed.

In the formal system of propositional intuitionistic logic, formulae (φ) are generated from propositional variables using the connectives $\wedge$, $\vee$, $\Rightarrow$, and the constants $\bot$ and $\top$. It also includes all classical propositional logic axioms except the doubled negation and the law of excluded middle.

The connection to Heyting algebras is immediate from the signature and axioms. In this subsection, we reserve $\Rightarrow$ for the implication connective of formulae, and $\rightarrow$ for the binary operation of a Heyting algebra. The logical connectives $\wedge$, $\vee$, $\Rightarrow$, $\bot$, and $\top$ match exactly the algebraic operations $\wedge$, $\vee$, $\rightarrow$, $\bot$, and $\top$ of a Heyting algebra and the axioms of Heyting algebras match exactly the inference rules of propositional intuitionistic logic. Thus every valuation of propositional variables in a Heyting algebra extends recursively to an interpretation of all formulae.

Definition 2.2.5 (Heyting-Algebra Model)**.** A *model* of propositional intuitionistic logic is a pair (H, v) where H is a Heyting algebra and $v : \mathit{Var} \to H$ is a valuation of propositional variables. The valuation extends uniquely to an interpretation $[\![\varphi]\!]_v \in H$ of formulae by:

$$[\![\bot]\!]_v = \bot \qquad [\![\top]\!]_v = \top$$
$$[\![\varphi \wedge \psi]\!]_v = [\![\varphi]\!]_v \wedge [\![\psi]\!]_v \qquad [\![\varphi \vee \psi]\!]_v = [\![\varphi]\!]_v \vee [\![\psi]\!]_v$$
$$[\![\varphi \Rightarrow \psi]\!]_v = [\![\varphi]\!]_v \to [\![\psi]\!]_v.$$

Definition 2.2.6 (Truth and Validity)**.** A formula φ is *true* in a model (H, v) when $[\![\varphi]\!]_v = \top$. It is *valid* in H when it is true for every valuation $v : \mathit{Var} \to H$.

The standard soundness/completeness connection is that a propositional formula is derivable in intuitionistic logic exactly when it is valid in every Heyting algebra. Logical derivability of implication corresponds to algebraic ordering in every model:

$$\vdash \varphi \Rightarrow \psi \quad \text{iff} \quad \text{for every model } (H, v), [\![\varphi]\!]_v \leq [\![\psi]\!]_v.$$

Here the notation $\varphi \Rightarrow \psi$ denotes a formula of the logic, whereas $\vdash \varphi \Rightarrow \psi$ is a meta-level judgment asserting that this formula is derivable from modus ponens and the axioms of propositional intuitionistic logic. This is the sense in which Heyting algebras are the algebraic semantics of propositional intuitionistic logic.

Having fixed the proof system and its Heyting-algebra semantics, we now add a theory T, that is, a deductively closed set of formulae.

Definition 2.2.7 (Theory)**.** A *theory* T of propositional intuitionistic logic is a set of formulae closed under provability: if $T \vdash \varphi$, then $\varphi \in T$.

2.2.4 Lindenbaum–Tarski Algebra

Informally, the purpose of this subsection is to establish the connection between propositional intuitionistic logic + theory and Heyting algebra + congruence. The rest of this subsection makes the connection precise.

Given a theory T, define a relation $\Rightarrow_T$ on formulae by:

$$\varphi \Rightarrow_T \psi \quad \text{iff} \quad T \vdash \varphi \Rightarrow \psi.$$

By reflexivity and transitivity of derivability, $\Rightarrow_T$ is a preorder. Therefore, each theory T induces an equivalence relation:

$$\varphi \equiv_T \psi \quad \text{iff} \quad (\varphi \Rightarrow_T \psi) \wedge (\psi \Rightarrow_T \varphi).$$

Equivalently, $\varphi \equiv_T \psi$ holds when T proves that φ and ψ imply each other. Since derivability is preserved by substitution into the logical connectives, provable equivalence is also preserved by $\wedge$, $\vee$, $\Rightarrow$, $\bot$, and $\top$. Therefore $\equiv_T$ is a congruence on the algebra of formulae.

Definition 2.2.8 (Lindenbaum–Tarski Algebra)**.** Let $\mathbb{P}$ be the free formula algebra of propositional intuitionistic logic over the set P of propositional variables; that is, $\mathbb{P}$ is the free algebra generated by P in the signature $\wedge$, $\vee$, $\Rightarrow$, $\bot$, and $\top$. The *Lindenbaum–Tarski algebra* of a theory T is the quotient algebra:

$$H_T \;=\; \mathbb{P}/{\equiv_T}.$$

Its elements are equivalence classes of formulae modulo provable equivalence in T, and its operations are induced from the logical connectives. For intuitionistic logic, H_T is again a Heyting algebra, and the pair (H_T, η_T) is the canonical Heyting-algebra model of the theory, where the canonical valuation $\eta_T : \mathit{Var} \to H_T$ is defined by

$$\eta_T(p) = [p]_{\equiv_T}.$$

The quotient order is exactly the entailment relation induced by the theory:

$$[\varphi]_{\equiv_T} \leq [\psi]_{\equiv_T} \quad \text{iff} \quad T \vdash \varphi \Rightarrow \psi \quad \text{iff} \quad \varphi \Rightarrow_T \psi.$$

By structural induction on formulae, the canonical valuation satisfies

$$[\![\varphi]\!] = [\varphi]_{\equiv_T}$$

for every formula φ. Thus the semantic value of a formula in the canonical model (H_T, η_T) is exactly its provable content modulo the theory.

Definition 2.2.9 (Prime-Filter Semantics)**.** Let F be a prime filter of H_T. The two-valued truth judgment induced by F is defined by

$$F \models \varphi \quad \text{iff} \quad [\varphi]_{\equiv_T} \in F.$$

Thus a prime filter is a two-valued viewpoint on the canonical Heyting-algebra model: it declares exactly those formulae in the filter to be true. This truth judgment should not be read as saying that F is a Heyting-algebra homomorphism into the two-point lattice. The missing structure is that intuitionistic implication must remain valid under every refinement of the current viewpoint.

Lemma 2.2.10 (Prime-Filter Forcing)**.** *Let F be a prime filter of H_T. Then:*

$$F \models \varphi \Rightarrow \psi \quad \textit{iff} \quad \forall G \supseteq F \,.\, (G \models \varphi \implies G \models \psi),$$

where G ranges over prime filters of H_T.

Applying theorem 2.2.4 to H_T shows that these prime-filter viewpoints separate formulae modulo theory T. In particular, if two formulae have distinct meanings in the canonical Heyting algebra H_T, then some prime filter makes one true and the other false. This is the canonical source of the prime-filter semantics that we later use to model attackers.

2.3 Semantic Framework

To reason about downgrading semantically, we separate the user-facing syntax of principals from the semantic authority levels induced by delegation. This separation lets us ask two background questions before turning to the technical security definitions: what algebraic structure should authority have, and what semantic adequacy conditions should an induced attacker model satisfy. The order-theoretic terminology used below is collected in §2.2.

2.3.1 The Lattice of Principals

We build our semantic framework upon the *lattice of principals*, used in prior work in authorization logics and information flow systems [6, 8, 9, 72, 75, 97, 98].

A principal $p \in \mathbb{P}$ refers to an entity in decentralized systems that can be either concrete, such as users or server machines, or abstract, such as RBAC roles [89] or quorums [112]. In IFC systems, they are often used as labels to annotate policies on use of information [8]. For example, `a: Alice` in fig. 2.2 requires the variable `a` to only be written by principals that can influence `Alice`'s data, and to remain secret to principals who cannot observe `Alice`'s information.

Principals are ordered by authority. When q delegates trust to p, we say p *acts for* q, written as $p \Rightarrow q$. Conjunction (the "and" logic connective) between principals $p \wedge q$ represents the least combined authority of p and q, and disjunction ("or") $p \vee q$ represents greatest common authority. The maximum authority 0 acts for all other authorities, and the minimum authority 1 trusts all other authorities. More authority is associated with elements lower in the lattice: $0 \Rightarrow p \wedge q \Rightarrow p \Rightarrow p \vee q \Rightarrow 1$. In the order-theoretic notation from §2.2, we read $p \Rightarrow q$ as $p \leq q$. Thus 0 corresponds to the bottom element, and 1 corresponds to

the top element. [2]

Additionally, we use a *delegation context*, with the form $\theta = p_1 \Rightarrow q_1, \cdots, p_n \Rightarrow q_n$, to specify delegations that are not implied by the logical structure of the principal lattice. The declaration `assume Alice = Bob for integrity` from fig. 2.2 is an example of a delegation context, specifying both $\mathrm{Alice} \Rightarrow \mathrm{Bob}$ and $\mathrm{Bob} \Rightarrow \mathrm{Alice}$. Using the delegation context is compatible with much prior work in IFC. This syntax is compatible with much prior work in IFC.

- The syntax of FLAM [8] is a free distributive lattice generated by projections of principal names. Its dynamic *trust configuration* is a delegation context.
- In the Rx label model [98], labels are join-semi-lattices of roles, whose semantics are sets of labels. Sets of labels form a boolean algebra, which is a special kind of distributive lattice. Delegation contexts are modeled as Rx *metapolicies*.
- Just like Rx, the Decentralized Label Model (DLM) [70] also has set-based semantics accompanied by syntactic rewrite rules. The analogue of the delegation context is the *interpretation function*, which allows acts-for between labels with unrelated syntax.
- The label algebra [68] is a unified abstraction for earlier label models including DC labels [97], DLM [72], Asbestos [36], and HiStar [109]. In this framework, labels are pre-lattices whose ordering is parameterized on another pre-lattice of *authorities*, which corresponds to our delegation context.

Semantically, a delegation context need not introduce new principals; instead, it can identify principal expressions that no longer differ as trust assumptions.

[2] It might seem odd to represent maximum authority as 0 and minimum authority with 1, since some prior work (e.g., [8]) makes the opposite choice. An intuitive justification: the "false" proposition entails everything, so no real principal can have authority 0. All principals are trusted with 1.

Those induced equivalence classes are the authority levels that our later security definitions quantify over. This is why we retain the lattice of principals as syntax, but do not take it to be the final semantic object.

The lattice of principals can be interpreted as an authorization logic [1, 45] where each principal is a proposition about authorization policy. Authorization logics are often built upon propositional intuitionistic logics. Accordingly, we assume that $\mathbb{P}$ carries the bounded distributive lattice structure reviewed in §2.2. The stronger Heyting-algebra structure is only needed later, in the label-inference section.

2.3.2 Delegation and Attackers

In the MPC example, a delegation context makes some principals equivalent. In this subsection, we give delegation a precise semantics. The main construction mirrors the logic developed in §2.2: a delegation context is treated as a theory, semantic acts-for is entailment under that theory, and attackers are prime-filter viewpoints on the induced quotient.

In systems with decentralized trust, all principals see other principals as potential attackers. In the extreme case where no principal trusts another, the attacker with respect to each principal controls all other principals. Formally, we characterize an attacker $A \subseteq \mathbb{P}$ by the set of principals it controls. At this point, such a set is only a candidate attacker. The semantic attacker model will keep exactly those candidates that respect the delegation assumptions and the lattice structure of authority.

Trust therefore restricts which candidate attackers we consider. If q trusts p, then an attacker that controls p but not q contradicts that trust assumption, because compromising p already compromises any principal that trusts p. Con-

versely, when a principal is attacker-controlled, so are the principals it acts for. Formally:

Definition 2.3.1 (Consistent Attackers)**.** A is consistent with θ ($A \models \theta$) when:

$$\forall p, q \in \mathbb{P} \,.\, ((p \Rightarrow q) \vee (p \Rightarrow q) \in \theta) \implies ((p \in A) \implies (q \in A))$$

The set of consistent attackers is an *attacker model*: $\mathbb{A}_{|\theta} = \{A \in \mathbb{A} \mid A \models \theta\}$.

The semantic trust levels of principals can be compared based on the set of consistent attackers that control the principals. Formally:

Definition 2.3.2 (Acts-for Semantics)**.** p acts for q (written $\theta \models p \leq q$) when:

$$\forall A \in \mathbb{A}_{|\theta} \,.\, p \in A \implies q \in A$$

We use "$\leq$" to denote the semantics of the acts-for relation "$\Rightarrow$". When viewing principals as authorization propositions, "acts-for" stands for "implies", and a delegation context is a theory (list of propositions). Each consistent attacker is a consistent interpretation (truth assignment) of the principals and the delegation context, where 1 is assigned to the principals the attacker controls.

In fact, a delegation context determines a *congruence relation*[3] over the lattice of principals, where $p \equiv_\theta q$ is defined as $(\theta \models p \leq q) \wedge (\theta \models q \leq p)$. As a result, $\equiv_\theta$ induces a quotient lattice $\mathbb{P}/\equiv_\theta$ where mutually delegating principals are in the same equivalence class. Intuitively, the quotient collapses all syntactic principal expressions that cannot be distinguished by any attacker consistent with θ. This quotient lattice is the algebraic model induced by θ. In the terminology of §2.2, it is the corresponding Lindenbaum algebra of the theory θ.

Theorem 2.3.3 (Algebraic Model)**.** *The algebraic model of the principal lattice $\mathbb{P}$ under delegation context θ is the quotient lattice $\mathbb{P}/\equiv_\theta$.*[4]

[3] A congruence is an equivalence relation that preserves the lattice structure.

[4] Full proofs of all theorems from this paper are available in the technical report [85].

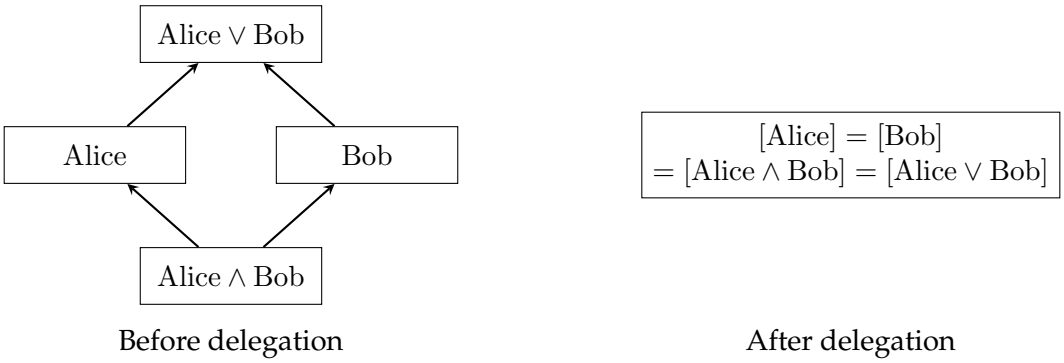

Figure 2.3: Mutual delegation between Alice and Bob in the semi-honest MPC example collapses the four-point Alice–Bob principal lattice to a single authority level in the quotient lattice.

Proof. Follows from the definition of Lindenbaum–Tarski Algebra of propositional intuitionistic logic. The Lindenbaum algebra of theory θ, $\mathbb{P}/\equiv_\theta$, is the initial algebra of the category of Heyting algebras that are consistent with θ. □

The quotient has a simple reading in the semi-honest MPC example from fig. 2.2, visualized in fig. 2.3. There, the integrity delegation context states that Alice and Bob trust each other, so $\mathrm{Alice} \equiv_\theta \mathrm{Bob}$ for integrity. Semantically, this means that the two principals determine the same authority level: any attacker viewpoint consistent with the delegation assumptions must either control both of them or neither of them. In fact, once Alice and Bob are identified, their meet and join are identified with them as well, so the entire four-point lattice collapses to a single authority level. Thus the quotient lattice records only the distinct trust assumptions that remain after delegation, and the attacker model ranges over prime-filter viewpoints on these induced authority levels rather than on the raw syntax of principal expressions.

By the background facts on distributive lattices and prime filters, the semantics of consistent attackers identifies them with the prime filters of the lattice of principals. This gives the attacker model enough viewpoints to separate distinct

authority levels, while excluding viewpoints that violate conjunction, disjunction, or delegation.

Theorem 2.3.4 (Attacker Model). $\mathbb{A}_{|\theta}$ *is the set of* prime filters *of* $\mathbb{P}/\equiv_\theta$.

Proof. We show that consistent attackers are in canonical bijection with the prime filters of the quotient lattice $\mathbb{P}/\equiv_\theta$.

Let $[p]_\theta$ abbreviate the equivalence class of p modulo $\equiv_\theta$. For any $A \in \mathbb{A}_{|\theta}$, define

$$\bar{A} = \{[p]_\theta \mid p \in A\}.$$

This is well-defined: if $[p]_\theta = [q]_\theta$ and $p \in A$, then $p \equiv_\theta q$, so in particular $\theta \models p \leq q$. Since $A \in \mathbb{A}_{|\theta}$, the definition of semantic acts-for gives $p \in A \implies q \in A$. Hence $[p]_\theta \in \bar{A}$ if and only if $[q]_\theta \in \bar{A}$.

We claim that $\bar{A}$ is a prime filter of $\mathbb{P}/\equiv_\theta$. Because A is an attacker, it is a prime filter of $\mathbb{P}$. Therefore:

- $1 \in A$, so $[1]_\theta \in \bar{A}$.
- If $[p]_\theta \in \bar{A}$ and $[p]_\theta \leq [q]_\theta$ in the quotient lattice, then by the quotient order, $\theta \models p \leq q$. Since $A \in \mathbb{A}_{|\theta}$ and $p \in A$, we obtain $q \in A$. Hence $[q]_\theta \in \bar{A}$.
- If $[p]_\theta, [q]_\theta \in \bar{A}$, then $p, q \in A$. Since A is a filter on $\mathbb{P}$, $p \wedge q \in A$. Therefore

  $$[p]_\theta \wedge [q]_\theta = [p \wedge q]_\theta \in \bar{A}.$$

- If

  $$[p]_\theta \vee [q]_\theta = [p \vee q]_\theta \in \bar{A},$$

  then $p \vee q \in A$. Since A is prime on $\mathbb{P}$, either $p \in A$ or $q \in A$. Hence either $[p]_\theta \in \bar{A}$ or $[q]_\theta \in \bar{A}$.

So every consistent attacker determines a prime filter of the quotient lattice.

Conversely, let F be a prime filter of $\mathbb{P}/\equiv_\theta$, and define

$$\underline{F} = \{p \in \mathbb{P} \mid [p]_\theta \in F\}.$$

We show $\underline{F} \in \mathbb{A}_{|\theta}$. First, $\underline{F}$ is a prime filter of $\mathbb{P}$:

- Since F is a filter, $[1]_\theta \in F$, so $1 \in \underline{F}$.
- If $p \Rightarrow q$ and $p \in \underline{F}$, then every attacker in $\mathbb{A}_{|\theta}$ is upward closed for the syntactic order, so $\theta \models p \leq q$. Equivalently, $[p]_\theta \leq [q]_\theta$ in the quotient lattice. Since $[p]_\theta \in F$ and F is upward closed, $[q]_\theta \in F$, hence $q \in \underline{F}$.
- If $p, q \in \underline{F}$, then $[p]_\theta, [q]_\theta \in F$. Since F is a filter,

  $$[p \wedge q]_\theta = [p]_\theta \wedge [q]_\theta \in F,$$

  so $p \wedge q \in \underline{F}$.
- If $p \vee q \in \underline{F}$, then $[p \vee q]_\theta \in F$. Since

  $$[p \vee q]_\theta = [p]_\theta \vee [q]_\theta$$

  and F is prime, either $[p]_\theta \in F$ or $[q]_\theta \in F$. Hence either $p \in \underline{F}$ or $q \in \underline{F}$.

Thus $\underline{F}$ is a prime filter of $\mathbb{P}$, that is, $\underline{F} \in \mathbb{A}$.

It remains to show consistency with θ. Suppose $((p \Rightarrow q) \vee (p \Rightarrow q) \in \theta)$ and $p \in \underline{F}$. By the definition of consistent attackers, every attacker in $\mathbb{A}_{|\theta}$ maps p to q; hence $\theta \models p \leq q$. Equivalently, $[p]_\theta \leq [q]_\theta$ in the quotient lattice. Since $[p]_\theta \in F$ and F is upward closed, $[q]_\theta \in F$, so $q \in \underline{F}$. Therefore $\underline{F} \models \theta$, and thus $\underline{F} \in \mathbb{A}_{|\theta}$.

Finally, the two constructions are inverse. If $A \in \mathbb{A}_{|\theta}$, then $\bar{\underline{A}} = A$ because consistency with θ makes A saturated under $\equiv_\theta$. If F is a prime filter of the quotient lattice, then by definition $\underline{\bar{F}} = F$. Hence $\mathbb{A}_{|\theta}$ is exactly the set of prime filters of $\mathbb{P}/\equiv_\theta$. □

Prime filters have intuitive interpretations, which abstracts and generalizes attacker models from prior work [18, 56]. $A \subseteq \mathbb{P}$ is a prime filter when:

- $1 \in A$: All attackers control the weakest authority;
- $0 \notin A$: No attacker controls the strongest authority;
- If $p \Rightarrow q$ and $p \in A$, then $q \in A$: Attackers are consistent;
- If $p \in A$ and $q \in A$, then $p \wedge q \in A$: An attacker controls the least combined authority of principals it controls;
- If $p \vee q \in A$, then either $p \in A$ or $q \in A$: If two principals are not controlled by an attacker, neither is their greatest common authority.

Defining an attacker as a prime filter may seem to impose many requirements, but this formulation actually abstracts and generalizes prior work:

- The cryptography literature models attackers by giving control of parties (atomic principals/authorities) Alice, Bob, Chuck, etc. to an adversary. We model such attackers by instantiating $\mathbb{P}$ as the free distributive lattice over the set of parties. The prime filters of $\mathbb{P}$ are then sets of the form $\uparrow \mathrm{Alice}$, $\uparrow(\mathrm{Alice} \wedge \mathrm{Bob})$, $\uparrow(\mathrm{Alice} \wedge \mathrm{Bob} \wedge \mathrm{Chuck})$, etc.,which are isomorphic to subsets of the parties. Using delegation, we can even capture the more nuanced notion of general adversary structures [56].
- Ethan Cecchetti et al. [18] when defining NMIFC make exactly the same assumptions (see their appendix B) albeit without making the connection to order theory.
- In finite lattices, all prime filters are upward-closed sets of authorities of the form $p \wedge p_2 \wedge ... \wedge p_n$ where all components p_i are not joins of other authority levels. Such attackers can be interpreted as a set of colluding parties.

- Some information flow systems concerned only with noninterference might seem to assume less: they define attackers as the upward closure of an arbitrary authority level, i.e., $\mathbb{A} = \{\uparrow p \mid p \in \mathbb{P}\}$. Such attacker models contain all prime filters and remain consistent and succinct according to theorem 2.3.3.

2.3.3 Order Characterization and Succinctness

As an additional sanity check on the semantics of delegation and attackers, we present the following two lemmas that have intuitive interpretations and hold for our semantic model.

For each principal $p \in \mathbb{P}$, write

$$\mathbb{A}_p = \{A \in \mathbb{A}_{|\theta} \mid p \in A\}$$

for the consistent attackers that control p.

First, the more attackers controlling an authority level, the less trusted it is. We formalize this intuition as an *order characterization*.

Lemma 2.3.5 (Order Characterization)**.** *For any delegation context θ:*

$$\forall p, q \in \mathbb{P} \centerdot (\theta \models p \leq q) \iff (\mathbb{A}_p \subseteq \mathbb{A}_q)$$

Proof. Unfolding theorem 2.3.2, $\theta \models p \leq q$ means exactly that every $A \in \mathbb{A}_{|\theta}$ containing p also contains q. By the definition of $\mathbb{A}_p$ and $\mathbb{A}_q$, this is equivalent to $\mathbb{A}_p \subseteq \mathbb{A}_q$. □

When two authority levels are controlled by the same set of attackers, they are treated equally in security analysis. So each authority level should be uniquely characterized by the set of attackers that control it—an idea we call *succinctness*.

Lemma 2.3.6 (Succinctness)**.** *For any delegation context θ:*

$$\forall p, q \in \mathbb{P} \centerdot \mathbb{A}_p = \mathbb{A}_q \implies p \equiv_\theta q$$

Proof. Assume $\mathbb{A}_p = \mathbb{A}_q$. To show $p \equiv_\theta q$, it suffices to show both $\theta \models p \leq q$ and $\theta \models q \leq p$. By theorem 2.3.5, it is enough to prove both $\mathbb{A}_p \subseteq \mathbb{A}_q$ and $\mathbb{A}_q \subseteq \mathbb{A}_p$, which are immediate from the assumption $\mathbb{A}_p = \mathbb{A}_q$. □

Evidently, order characterization makes semantic acts-for $\leq$ a preorder; succinctness says that equal attacker sets coincide exactly with semantic equivalence classes.

Order characterization and succinctness serve as simple sanity checks that the semantics of labels, delegations, and the attacker model do not contradict each other.

2.3.4 Labels

Information flow systems mainly consider two aspects of security: confidentiality (read authority) and integrity (write authority). To express differing confidentiality and integrity policies, we use *the lattice of labels*, whose elements are pairs of principals $\mathbb{L} = \mathbb{P} \times \mathbb{P}$. For a label $\ell = \langle p, q \rangle$, p represents confidentiality and q represents integrity. The two components are interpreted independently: p says who may learn the information, while q says whose influence is trusted in the information. Like principals, each label reflects the authority required for a principal to access information. For example, information labeled $\langle 0, 1 \rangle$ can be read by no principal except the strongest 0, but it can be influenced by any principal.

A label ℓ acts for ℓ' when ℓ acts for ℓ' in both confidentiality and integrity. Similarly, conjunction/disjunction of labels is defined by the conjunction/dis-

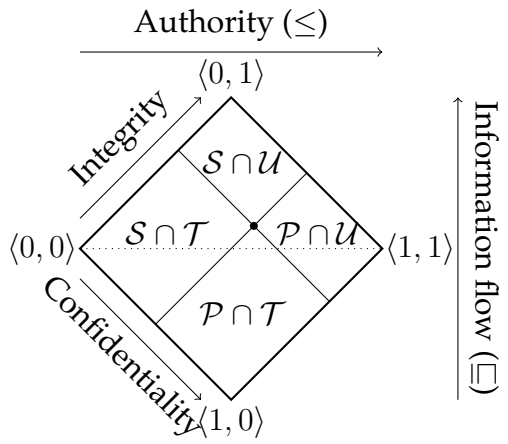

Figure 2.4: The lattice of labels $(\mathbb{L}, \leq)$ and the lattice of information flow $(\mathbb{L}, \sqsubseteq)$ share the same underlying set, but use a different ordering. The dotted line depicts labels with equal confidentiality and integrity: strictly above are compromised labels, and on or below are uncompromised labels.

junction of their confidentiality and integrity. An asymmetric delegation context is a pair of different delegation contexts $\Theta = \langle \theta_{\rightarrow}, \theta_{\leftarrow} \rangle$.

In general, *asymmetric attackers* $A \in \mathbb{A} = \mathbb{A} \times \mathbb{A}$ may control different principals for confidentiality and integrity. We write $A = \langle C \in \mathbb{A}, I \in \mathbb{A} \rangle$, where C represents the principals A controls for confidentiality, and I those for integrity. Consequently, the attacker model $\mathbb{A}$ is a pair of prime filters.

As visualized in fig. 2.4, each attacker defines secret ($\mathcal{S}$), public ($\mathcal{P}$), trusted ($\mathcal{T}$), and untrusted ($\mathcal{U}$) sets over the lattice of labels.

$$\mathcal{P}_{\langle C,I \rangle} = \{ \langle p, q \rangle \mid p \in C \}\ ,\ \ \mathcal{U}_{\langle C,I \rangle} = \{ \langle p, q \rangle \mid q \in I \},$$
$$\mathcal{S}_{\langle C,I \rangle} = \{ \langle p, q \rangle \mid p \notin C \}\ ,\ \ \mathcal{T}_{\langle C,I \rangle} = \{ \langle p, q \rangle \mid q \notin I \}$$

2.4 Security Hyperproperties and Delegation

Our semantic framework formalizes a key insight relating the attacker model and delegation: more delegation means fewer attackers. In turn, fewer attackers

should make it easier for programs to be considered secure. To express delegation, we extend prior definitions of IFC security *hyperproperties* [24] with our semantic framework. Rather than a standard proof of security for an IFC-typed core calculus, we abstract away from computation to concentrate on the core judgment in IFC systems: when it is safe to relabel information, under a delegation context.

The simple system has states $\sigma \in \sigma$, intentionally left unspecified. An execution of the system emits a trace t, which is a sequence of states. Define the *behavior* $B \in \mathbb{B}$ of a program to be the *set* of possible execution traces it can emit. A hyperproperty $\mathbb{HP} \subseteq \mathbb{B}$ is a set of behaviors. A program satisfies a hyperproperty when its behavior is a member of the hyperproperty.

In decentralized IFC systems with heterogeneous trust assumptions, each principal has a different view of the system. To capture the idea of the view of a program from the perspective different principals, we parameterize hyperproperties $\mathbb{HP}_A$ by the choice of attacker: a program secure against one attacker may be insecure against another. Let the hyperproperty $\mathbb{HP}_{\mathbb{A}}$ be the hyperproperty that characterizes programs that are secure against all possible attackers from $\mathbb{A}$. A behavior B of a program falls into $\mathbb{HP}_{\mathbb{A}}$ precisely when $B \in \mathbb{HP}_A$ for all $A \in \mathbb{A}$:

$$\mathbb{HP}_{\mathbb{A}} = \bigcap_{A \in \mathbb{A}} \mathbb{HP}_A \qquad\qquad \Theta(\mathbb{HP}_{\mathbb{A}}) = \bigcap_{A \in \mathbb{A}_{|\Theta}} \mathbb{HP}_A$$

When a program assumes a static delegation context Θ, the attacker model is further restricted to the ones consistent with Θ. Therefore, a delegation context can be understood as a hyperproperty transformer. It follows that hyperproperties accept at least as many programs after a delegation context is added.

Theorem 2.4.1. $\mathbb{HP}_{\mathbb{A}} \subseteq \Theta(\mathbb{HP}_{\mathbb{A}})$.

Proof. By definition, $\mathbb{HP}_{\mathbb{A}} = \bigcap_{A \in \mathbb{A}} \mathbb{HP}_A$ and $\Theta(\mathbb{HP}_{\mathbb{A}}) = \bigcap_{A \in \mathbb{A}_{|\Theta}} \mathbb{HP}_A$. Since $\mathbb{A}_{|\Theta} \subseteq \mathbb{A}$, the intersection defining $\Theta(\mathbb{HP}_{\mathbb{A}})$ ranges over fewer attackers. Therefore every

behavior secure against all attackers in $\mathbb{A}$ is secure against all attackers consistent with Θ, so $\mathbb{HP}_{\mathbb{A}} \subseteq \Theta(\mathbb{HP}_{\mathbb{A}})$. □

This matches our intuition about static delegation: the more trust assumptions, the fewer reasonable attackers, the more programs considered secure.

2.4.1 Noninterference and the Lattice of Information Flow

For confidentiality, noninterference [31, 87] demands that information should not flow from high to low (secret to public). Noninterference of integrity requires that information should not flow from low to high (untrusted to trusted). To illustrate how delegation affects noninterference, we adapt Clarkson and Schneider's [24] definition of *observational determinism* [47].

Definition 2.4.2 (Observational Determinism)**.** Let L be any set of *low* labels. Observational determinism $\mathbb{OD}$ is the hyperproperty:

$$\mathbb{OD}_L = \{B \mid \forall t_1, t_2 \in B \centerdot t_1^0 =_L t_2^0 \implies t_1 \approx_L t_2\}$$

State t^0 denotes the initial state of the trace t. We leave the definition of low equivalence between states unspecified, as in prior work on knowledge-based security [10, 60, 62, 96].

Noninterference for confidentiality $\mathbb{NI}_A^{\rightarrow} = \mathbb{OD}_{\mathcal{P}_A}$ and integrity $\mathbb{NI}_A^{\leftarrow} = \mathbb{OD}_{\mathcal{T}_A}$ are mere instantiations of observational determinism over public and trusted labels for some attacker A. For an attacker model $\mathbb{A}_{|\Theta}$:

$$\Theta(\mathbb{NI}_{\mathbb{A}}) = \bigcap_{A \in \mathbb{A}_{|\Theta}} (\mathbb{OD}_{\mathcal{P}_A} \cap \mathbb{OD}_{\mathcal{T}_A})$$

Prior work [48, 58, 71] enforces low equivalence by ensuring that low-labeled information is not influenced by high-labeled information. Concretely, a dynamic

relabel is safe when it does not relabel high-labeled information to a low label. We formalize safe relabeling as the *flows-to* relation.

Definition 2.4.3 (Flows-to)**.** Label ℓ securely flows to ℓ' ($\Theta \models \ell \sqsubseteq \ell'$) when:

$$\forall A \in \mathbb{A}_{|\Theta} \centerdot (\ell' \in \mathcal{P}_A \implies \ell \in \mathcal{P}_A) \wedge (\ell' \in \mathcal{T}_A \implies \ell \in \mathcal{T}_A)$$

Equivalently, $\langle \theta_{\rightarrow}, \theta_{\leftarrow} \rangle \models \langle p, q \rangle \sqsubseteq \langle p', q' \rangle$ when $\theta_{\rightarrow} \models p' \leq p$ and $\theta_{\leftarrow} \models q \leq q'$.

As visualized in fig. 2.4, flows-to and the acts-for define two lattices on the same underlying set of labels. The Lattice of Information operators are given by:

$$\langle p_1, q_1 \rangle \sqcup \langle p_2, q_2 \rangle = \langle p_1 \wedge p_2, q_1 \vee q_2 \rangle \qquad \langle p_1, q_1 \rangle \sqcap \langle p_2, q_2 \rangle = \langle p_1 \vee p_2, q_1 \wedge q_2 \rangle$$

2.4.2 Downgrading and Nonmalleable Information Flow

Some programs, such as the MPC example from fig. 2.2, intentionally break noninterference. Prior work on dynamic security policies either achieve downgrading by *relabeling* or by *dynamic delegation*. Visually, relabeling moves information downward in fig. 2.4 and dynamic delegation moves the attacker partition leftward.

Nonmalleable Information Flow (NMIF)

To prevent misuse of downgrades by relabeling, Cecchetti et al. [18] propose NMIF, a security hyperproperty that combines robust declassification [106] with transparent endorsement.

Robust declassification requires that secret information flow to public only when the information is trusted. This restriction ensures that attackers do not influence disclosure of information to them. A declassification is only robust when it declassifies secret–trusted information ($\mathcal{S}_A \cup \mathcal{T}_A$).

Transparent endorsement is a dual condition that allows untrusted information to influence trusted information only when the information is public to the attacker. It only allows endorsement of public–untrusted information ($\mathcal{P}_A \cup \mathcal{U}_A$).

Therefore, secret–untrusted information should not be downgraded. Indeed, the key recipe to enforcing NMIF is to enforce noninterference for public or trusted information [18]:

$$\mathbb{NI}^{\blacktriangledown}_{\mathbb{A}} = \mathbb{OD}_{\mathcal{T}_A \cup \mathcal{P}_A}$$

A label is *compromised* when it is secret–untrusted for some attacker [18, 105]. To enforce NMIF, it suffices to reject downgrading information with compromised labels, so that there is no flow out from compromised labels.

Unfortunately, NMIF against $\mathbb{A}_{|\theta}$ rejects all downgrades. Namely, all labels (except for $\langle 1, 0 \rangle$) are compromised against the attacker $A_\top = \langle \{1\}, \mathbb{P} \rangle$ (it controls every principal for integrity and controls no principal for confidentiality). Therefore, further restrictions on the attacker model are needed.

NMIF Attackers

Prior work [105, 106] assumes attackers control more confidentiality than integrity. We call them *valid attackers*:

Definition 2.4.4 (Valid Attackers)**.** $\mathbf{V} = \{\langle C, I \rangle \in \mathbb{A} \mid I \subseteq C\}$

Restricting attackers to valid ones makes our framework mirror attacker models studied in the cryptography literature [103], where a principal is honest (not controlled by the attacker), semi-honest (controlled by the attacker for confidentiality), or malicious (controlled by the attacker for both confidentiality and integrity). The valid attacker restriction excludes unrealistic "malicious but incurious" attackers.

In fact, much existing work satisfies the valid-attacker assumption by construction. For example, robust declassification [106] originally defines integrity and confidentiality by equivalence relations over system states. The state transitions an active attacker may perform are, by construction, observable by the attacker.

Definition 2.4.5 (Uncompromised Labels)**.** Label ℓ is uncompromised under Θ, written $\Theta \models \blacktriangledown \ell$, when ℓ is either public or trusted for all *valid attackers*:

$$\forall A \in \mathbf{V}_{|\Theta} \,.\, \ell \in \mathcal{P}_A \cup \mathcal{T}_A$$

It follows that labels of principals are uncompromised: $\Theta \models \blacktriangledown \langle p, p \rangle$.

Uncompromised labels have an alternative characterization: they are the labels with at least as much integrity as confidentiality. Of course, in the presence of asymmetric delegation, we cannot directly compare a label's confidentiality and integrity components since each component has a different set of delegations. We circumvent this problem by introducing a witnessing principal r who has no more integrity than q and no less confidentiality than p.

Definition 2.4.6 (Closures)**.** Define upward and downward closed sets as follows:

$$\uparrow p = \{q \in \mathbb{P} \mid p \Rightarrow q\} \qquad\qquad \downarrow p = \{q \in \mathbb{P} \mid q \Rightarrow p\}$$

Definition 2.4.7 (Equivalence Classes)**.** The equivalence class of a principal with respect to a delegation context is:

$$[p]_\theta = \{q \in \mathbb{P} \mid (\theta \models p \leq q) \wedge (\theta \models q \leq p)\}$$

We extend this notation to denote the equivalence closure of a set:

$$[P]_\theta = \bigcup_{p \in P} [p]_\theta$$

Remark 2.4.8. We denote the *set* of equivalence classes using standard notation:

$$P/\theta = P/\equiv_\theta = \{[p]_\theta \mid p \in P\}$$

for $P \subseteq \mathbb{P}$ (we might have $P = \mathbb{P}$). This is different from $[P]_\theta \subseteq \mathbb{P}$.

Remark 2.4.9. Note that $[p]_\theta \in \mathbb{P}/\theta$ and $\uparrow[p]_\theta \subseteq \mathbb{P}/\theta$. In particular, $\uparrow[p]_\theta$ is a set of sets of principals, not a set of principals.

Theorem 2.4.10 (Prime Ideal/Filter)**.** *Let L be a distributive lattice, $I \subseteq L$ an ideal, and $F \subseteq L$ a filter such that $I \cap F = \emptyset$. Then there exists a prime ideal P and a prime filter Q such that*

$$I \subseteq P \qquad F \subseteq Q \qquad P \cap Q = \emptyset$$

The following theorem shows an alternative characterization of uncompromised labels that is more amenable to algorithmic implementation we present in §2.5. Intuitively, the principal r witnesses that the label $\langle p, q\rangle$ is uncompromised.

Theorem 2.4.11. $\langle \theta_\rightarrow, \theta_\leftarrow\rangle \models \blacktriangledown\, \langle p, q\rangle \iff \exists r \in \mathbb{P}\,.\, (\theta_\leftarrow \models q \leq r) \wedge (\theta_\rightarrow \models r \leq p)$.

Proof. We prove each direction separately.

- Case $\Longrightarrow$. Assume, for contradiction, that $q \notin [[\downarrow p]_{\theta_\rightarrow}]_{\theta_\leftarrow}$. The sets $\uparrow[q]_{\theta_\leftarrow}$ and $[\downarrow p]_{\theta_\rightarrow}/\theta_\leftarrow$ are disjoint and form a filter/ideal pair of $\mathbb{P}/\theta_\leftarrow$. Thus, theorem 2.4.10 gives a prime filter I_0 of $\mathbb{P}/\theta_\leftarrow$ with $\uparrow[q]_\theta \subseteq I_0$ and $I_0 \cap ([\downarrow p]_{\theta_\rightarrow}/\theta_\leftarrow) = \emptyset$.

  Let $I = \bigcup I_0$. Note that I and $[[\downarrow p]_{\theta_\rightarrow}]_{\theta_\leftarrow}$ are disjoint and $[\downarrow p]_{\theta_\rightarrow} \subseteq [[\downarrow p]_{\theta_\rightarrow}]_{\theta_\leftarrow}$, so I and $[\downarrow p]_{\theta_\rightarrow}$ are disjoint as well. This in turn implies $I/\theta_\rightarrow$ and $\downarrow p/\theta_\rightarrow$ are disjoint. Moreover, these sets form a filter/ideal pair in $\mathbb{P}/\theta_\rightarrow$, so theorem 2.4.10 gives a prime filter C_0 of $\mathbb{P}/\theta_\rightarrow$ with $I/\theta_\rightarrow \subseteq C_0$ and $C_0 \cap (\downarrow p/\theta_\rightarrow) = \emptyset$.

Now, define $C = \bigcup C_0$ and note that

$$I = \bigcup (I/\theta_{\rightarrow}) \subseteq \bigcup C_0 = C$$

Since I_0 is an ideal of $\mathbb{P}/\theta_{\leftarrow}$, we have $I \models \theta_{\leftarrow}$. Since C_0 is an ideal of $\mathbb{P}/\theta_{\rightarrow}$, we have $C \models \theta_{\rightarrow}$. Combining these three results, we have $\langle C, I\rangle \in \mathbf{V}_{|\langle\theta_{\rightarrow},\theta_{\leftarrow}\rangle}$.

Because $\langle\theta_{\rightarrow}, \theta_{\leftarrow}\rangle \models \blacktriangledown \langle p, q\rangle$ and $q \in \uparrow[q]_{\theta_{\leftarrow}} \subseteq I_0 \implies q \in I$, we have $p \in C$. However, $p \in [\downarrow p]_{\theta_{\rightarrow}}$, which contradicts the fact that C_0 and $\downarrow p/\theta_{\rightarrow}$ are disjoint. Thus, we must have $q \in [[\downarrow p]_{\theta_{\rightarrow}}]_{\theta_{\leftarrow}}$.

Finally, $q \in [[\downarrow p]_{\theta_{\rightarrow}}]_{\theta_{\leftarrow}}$ means there exists $r \in [\downarrow p]_{\theta_{\rightarrow}}$ such that $q \in [r]_{\theta_{\leftarrow}}$. By definition, we have $\theta_{\leftarrow} \models q \leq r$. Moreover, $r \in [\downarrow p]_{\theta_{\rightarrow}}$ means there exists $p' \in \downarrow p$ such that $r \in [p']_{\theta_{\rightarrow}}$. By definition, $\theta_{\rightarrow} \models r \leq p'$, and theorem 2.3.4 with $p' \in \downarrow p$ gives $\theta_{\rightarrow} \models p' \leq p$. By transitivity, we have $\theta_{\rightarrow} \models r \leq p$.

- Case $\Longleftarrow$. Assume there exists r such that $\theta_{\leftarrow} \models q \leq r$ and $\theta_{\rightarrow} \models r \leq p$. We claim $\langle p, q\rangle$ is public whenever it is untrusted. Formally, let $\langle C, I\rangle \in \mathbf{V}$ such that $\langle C, I\rangle \models \langle\theta_{\rightarrow}, \theta_{\leftarrow}\rangle$, and assume $q \in I$. We need to show $p \in C$.

  Since $I \models \theta_{\leftarrow}$, $\theta_{\leftarrow} \models q \leq r$, and $q \in I$, we have $r \in I$. Moreover, $I \subseteq C$ (theorem 2.4.4), so $r \in C$. Finally, since $C \models \theta_{\rightarrow}$, $\theta_{\rightarrow} \models r \leq p$, and $r \in C$, we have $p \in C$ as desired.

□

In the absence of asymmetric delegation, theorem 2.4.11 reduces to a simple acts-for check ($\theta \models q \leq p$), which prior systems rely on [18, 106].

2.5 Algorithms

In IFC systems that incorporate delegation, the acts-for relation $\theta \models p_1 \leq p_2$ is frequently checked by label-inference procedures [75] and even at run time, where

it can impose significant overhead [22]. Unfortunately, its semantic definition quantifies over the potentially infinite set of all attackers, which makes direct use of the definition infeasible in practice. Similarly, the definition of uncompromised labels $\Theta \models \blacktriangledown\ell$ does not yield an algorithm.

In this section, we assume oracle access to syntactic lattice operations ($\Rightarrow$, 0, 1, $\wedge$, $\vee$) of $\mathbb{P}$, and propose sound and complete algorithms that check for the aforementioned relations.

2.5.1 Acts-for Algorithm

Algorithm 2.5.1 gives an algorithm for deciding $\theta \models p \leq q$. We write $\theta \vdash p \leq q$ to denote this algorithmic system.

Algorithm 2.5.1 (Acts-for $\theta \vdash p \leq q$)**.**

$$\frac{p \Rightarrow q}{\cdot \vdash p \leq q}\ \mathbb{P}\text{-AXIOM} \qquad \frac{\theta \vdash p \wedge q' \leq q \qquad \theta \vdash p \leq q \vee p'}{\theta, p' \Rightarrow q' \vdash p \leq q}\ \mathbb{P}\text{-DELEGATION}$$

The algorithm recursively applies the lattice axioms and rule $\mathbb{P}$-DELEGATION until the delegation context is empty for all sub-cases. The acts-for relation holds when rule $\mathbb{P}$-AXIOM applies to all sub-cases.

Equivalently, the rules define the following recursive decision procedure. Let $\mathsf{ActsFor}(\theta, p, q)$ be the boolean-valued function:

$$\begin{aligned} \mathsf{ActsFor}(\cdot, p, q) &= (p \Rightarrow q), \\ \mathsf{ActsFor}(\theta, p, q) &= \mathsf{ActsFor}(\theta', p \wedge q', q) \wedge \mathsf{ActsFor}(\theta', p, q \vee p') \end{aligned}$$

when $\theta = \theta', p' \Rightarrow q'$.

That is, each recursive call removes one delegation from the context and replaces the original goal by the two subgoals prescribed by rule $\mathbb{P}$-DELEGATION; the procedure returns true exactly when all leaves reduce to the base check $p \Rightarrow q$.

Below, we develop the correctness of the acts-for algorithm.

Lemma 2.5.2. *If $\theta_1 \models p \leq q$ holds, then $(\theta_1, \theta_2) \models p \leq q$ for any θ_2.*

Proof. Immediate since $\mathbb{A}_{|\theta_1,\theta_2} \subseteq \mathbb{A}_{|\theta_1}$. □

Corollary 2.5.3. *If $p \Rightarrow q$ holds, then $\theta \models p \leq q$ for any θ.*

Proof. We have $\cdot \models p \leq q$ by theorem 2.3.4. The result then follows by theorem 2.5.2. □

Lemma 2.5.4. *If $\cdot \models p \leq q$ holds, then $p \Rightarrow q$.*

Proof. Assume, for contradiction, that $p \not\Rightarrow q$. The set $\uparrow p = \{p' \in \mathbb{P} \mid p \Rightarrow p'\}$ is a filter of $\mathbb{P}$, and $\downarrow q = \{q' \in \mathbb{P} \mid q' \Rightarrow q\}$ is an ideal of $\mathbb{P}$. Moreover, $\uparrow p$ and $\downarrow q$ are disjoint since $p \not\Rightarrow q$. By theorem 2.4.10, there exists a prime filter $P \supseteq \uparrow p$ that is also disjoint from $\downarrow q$. This means $p \in P$ and $q \notin P$. However, $P \in \mathbb{A}$ by theorem 2.3.4 (attackers are prime filters) and $P \models \cdot$ trivially, which contradicts the assumption $\cdot \models p \leq q$. □

Lemma 2.5.5. *If $(\theta, p_1 \Rightarrow q_1) \models p_2 \leq q_2$ holds, then $\theta \models p_2 \wedge q_1 \leq q_2$.*

Proof. Let A be an attacker such that $A \models \theta$ and $p_2 \wedge q_1 \in A$. We need to show $q_2 \in A$. Since $p_2 \wedge q_1 \in A$ and A is upward-closed (theorem 2.3.4), we have $p_2, q_1 \in A$. From $A \models \theta$ and $q_1 \in A$, we get $A \models \theta, p_1 \Rightarrow q_1$. Using this, and the fact that $p_2 \in A$, we invoke our primary assumption to get $q_2 \in A$. □

Lemma 2.5.6. *If $(\theta, p_1 \Rightarrow q_1) \models p_2 \leq q_2$ holds, then $\theta \models p_2 \leq q_2 \vee p_1$.*

Proof. Let A be an attacker such that $A \models \theta$ and $p_2 \in A$. We need to show $q_2 \vee p_1 \in A$. If $p_1 \in A$, then $q_2 \vee p_1 \in A$ since A is upward-closed (theorem 2.3.4), so assume $p_1 \notin A$. Then, $A \models \theta, p_1 \Rightarrow q_1$ and we assumed $p_2 \in A$, so we can invoke our primary assumption to get $q_2 \in A$. Since A is upward-closed, this gives $q_2 \vee p_1 \in A$. □

Theorem 2.5.7 (Correctness of Acts-for)**.** *Algorithm 2.5.1 terminates, and it is sound and complete with respect to theorem 2.3.2:* $\theta \vdash p \leq q \iff \theta \models p \leq q$.

Proof. We first prove termination. The algorithm terminates because each time rule $\mathbb{P}$-DELEGATION is applied, the delegation context gets smaller. Since all delegation contexts are finite, rule $\mathbb{P}$-DELEGATION is applied at most finitely many times until it reaches the base case rule $\mathbb{P}$-AXIOM.

We prove soundness by induction on the derivation of $\theta \vdash p \leq q$.

- Case $\mathbb{P}$-AXIOM.. We have $p \Rightarrow q$ by inversion on the derivation. The results follows from theorem 2.5.3.
- Case $\mathbb{P}$-DELEGATION.. We have $\theta = (\theta', p' \Rightarrow q')$, $\theta' \vdash p \wedge q' \leq q$, and $\theta' \vdash p \leq q \vee p'$. Induction gives $\theta' \models p \wedge q' \leq q$ and $\theta' \models p \leq q \vee p'$. Let A be an attacker such that $A \models \theta', p' \Rightarrow q'$ and $p \in A$. We need to show $q \in A$.

  If $p' \in A$, then $q' \in A$ since $A \models p' \Rightarrow q'$. Since $p, q' \in A$, theorem 2.3.4 gives $p \wedge q' \in A$. The first induction hypothesis then gives $q \in A$.

  Otherwise, $p' \notin A$. The second induction hypothesis gives $q \vee p' \in A$. Theorem 2.3.4 then implies $q \in A$ since we cannot have $p' \notin A$ and $q \notin A$ but $q \vee p' \in A$.

We prove completeness by induction on θ.

- Case $\theta = \cdot$. Assume $\cdot \models p \leq q$. Then $p \Rightarrow q$ by theorem 2.5.4. Applying rule $\mathbb{P}$-AXIOM, we get $\cdot \vdash p \leq q$ as desired.
- Case $\theta = \theta', p' \Rightarrow q'$. Assume $\theta', p' \Rightarrow q' \models p \leq q$. By theorems 2.5.5 and 2.5.6, we get $\theta' \models p \wedge q' \leq q$ and $\theta' \models p \leq q \vee p'$. By induction, we get $\theta' \vdash p \wedge q' \leq q$ and $\theta' \vdash p \leq q \vee p'$. We can now apply rule $\mathbb{P}$-DELEGATION to get $\theta', p' \Rightarrow q' \vdash p \leq q$ as desired. □

2.5.2 NMIF Algorithm

Neither the semantic definition of uncompromised labels (theorem 2.4.5) nor their alternative characterization (theorem 2.4.11) lends itself to an algorithmic implementation. The semantic definition quantifies over all valid attackers (a potentially infinite set), and the alternative characterization conjures up an intermediate principal. Our solution is to use the alternative characterization but with a "best principal." For that, we use $\min_\theta(p)$, the highest-authority principal in the equivalence class of p.[5]

Definition 2.5.8 (Minimal Principal)**.** $\min_\theta(p) \in \mathbb{P}$ is the (necessarily) unique principal such that $\theta \models p \leq q$ if and only if $\min_\theta(p) \Rightarrow q$ for any $q \in \mathbb{P}$.

As shown in Algorithm 2.5.11, our algorithm relies on operations that *factor* lattice elements. In addition to the operators ($\leq$, 0, 1, $\wedge$, $\vee$), we also need to assume an oracle that determines whether a given principal is *join-prime*.

Definition 2.5.9 (Join-Prime)**.** $p \neq 1$ is join-prime if $p \leq q_1 \vee q_2$ implies either $p \leq q_1$ or $p \leq q_2$.

Intuitively, join-prime principals are not the join of other principals. In finite lattices, they are principals of the form $p = q_1 \wedge \cdots \wedge q_n$.

In addition, the factorization assumption need to hold for $\mathbb{P}$ for the min algorithm to exist.

Definition 2.5.10 (Factorization)**.** A lattice has a *join factorization* if every element p has a finite prime factorization; that is, p can be written as $p = q_1 \vee \cdots \vee q_n$, where $q_1, \ldots, q_n$ are join-prime.

[5] Recall that higher authority is lower in the authority lattice, thus the use of $\min$ as opposed to $\max$.

The factorization assumption says that all principals are the join of finitely many join-prime principals. The assumption holds for all finite principal lattices.

A counterexample provides intuition on why a factorization oracle is necessary for the existence of algorithms that find minimal elements. Consider the delegation context $\theta = p \leq p \wedge q$. Then the minimal element of $p \vee q$'s equivalence class is $\min_\theta(p \vee q) = q$. Without the factorization assumption, we may only use the two lattice operators meet $\wedge$ and join $\vee$. There is no way to express the result q in terms of the input elements p, $p \vee q$, $p \wedge q$ using only $\wedge$ and $\vee$.

In fact, we cannot write down all the principals without assuming factorization. Indeed, existing implementations of IFC models assume join factorization by using a syntax for principals based on a finite number of principal names [6, 8, 75, 97].

Algorithm 2.5.11 gives derivation rules for computing $\min_\theta(p)$. The rules can be applied in any order without backtracking since $\min_\theta(p)$ is unique when it exists.

Algorithm 2.5.11 (Min $\min_\theta(p) = q$)**.**

MIN-BASE

$$\frac{\text{join-prime}(p) \qquad \forall (p' \Rightarrow q') \in \theta \centerdot p \not\Rightarrow p'}{\min_\theta(p) = p}$$

MIN-PICK

$$\frac{\text{join-prime}(p) \qquad p \Rightarrow p'}{\min_{\theta, p' \Rightarrow q'}(p) = \min_\theta(p \wedge q')}$$

MIN-FACTOR

$$\frac{\neg\,\text{join-prime}(p) \qquad p = p_1 \vee \cdots \vee p_n \qquad \forall i \in [n] \centerdot \text{join-prime}(p_i)}{\min_\theta(p) = \bigvee_{i \in [n]} \min_\theta(p_i)}$$

Equivalently, the rules define the following recursive procedure. Let $\mathsf{Min}(\theta, p)$

be the principal-valued function:

$$\mathsf{Min}(\theta, p) = \begin{cases} \bigvee_{i \in [n]} \mathsf{Min}(\theta, p_i) & \text{if } p \text{ is not join-prime and } p = p_1 \vee \cdots \vee p_n, \\ \mathsf{Min}(\theta', p \wedge q') & \text{if } p \text{ is join-prime, } \theta = \theta', p' \Rightarrow q', \text{ and } p \Rightarrow p', \\ p & \text{if } p \text{ is join-prime and no } (p' \Rightarrow q') \in \theta \text{ satisfies } p \Rightarrow p'. \end{cases}$$

The procedure first factors all principals into join-prime components and recurses on each component. For a join-prime principal, it consumes any applicable delegation by replacing p with $p \wedge q'$ and removing that delegation from the context; when no delegation applies, the current principal is already the minimal.

Moreover, all derivations are finite since either the size of θ decreases, or we switch from a reducible element to a finite set of irreducible elements. The run time of the algorithm is linear in the size of the delegation context and the lengths of join-prime factorizations.

We must demand more structure on $\mathbb{P}$ to compute $\min_\theta(p)$. Specifically, we rely on an oracle that returns *join-prime* factorizations of arbitrary principals. Intuitively, a join-prime principal cannot be written as the join of other principals. In finite lattices, these are the principals of the form $p = q_1 \wedge \cdots \wedge q_n$. Factorization oracles arise trivially in existing implementations of IFC models, as these are based on lattices with a finite set of principal names [6, 8, 75, 97].

Lemma 2.5.12. *If $\theta \models p_2 \wedge q_1 \leq q_2$ and $\theta \models p_2 \leq p_1$, then $(\theta, p_1 \Rightarrow q_1) \models p_2 \leq q_2$.*

Proof. Follows trivially from theorem 2.3.4 since attackers are closed under $\wedge$. □

Theorem 2.5.13 (Correctness of Min)**.** *Algorithm 2.5.11 terminates, and it is sound and complete with respect to theorem 2.5.8.*

Proof. We first prove termination. Define the following relation $\succeq$ between

expressions in the form of $\min_\theta(p)$:

$$\min_\theta(p) \succeq \min_{\theta'}(p) \ \textbf{if} \ \theta' \subseteq \theta$$

$$\min_\theta(p) \succeq \min_\theta(q) \ \textbf{if} \ \text{join-prime}(q)$$

This is a well-founded relation (no infinite descending chain): because of the uniqueness of $\text{join-prime}(p)$, fixing any specific θ, a chain cannot be longer than 2. So each chain starting at $\min_\theta(p)$ is no longer than $2|\theta|$, which is finite. As rule MIN-PICK and rule MIN-FACTOR are both decreasing, the algorithm eventually reaches the base case and then terminates.

Minimal elements are unique and theorem 2.5.11 always terminates, so it suffices to prove soundness. More concretely, whenever theorem 2.5.11 derives $\min_\theta(p) = p'$, we need to show $\theta \models p \leq q \iff p' \Rightarrow q$ for all q.

Let $q \in \mathbb{P}$. We proceed by induction on the derivation.

- Case MIN-BASE. We have $\theta = (p_1 \Rightarrow q_1, \ldots, p_n \Rightarrow q_n)$, $\min_\theta(p) = p$, $\text{join-prime}(p)$, and $\forall i \in [n] \centerdot p \not\Rightarrow p_i$.

  - Case $\Longrightarrow$. Assume $\theta \models p \leq q$. We apply theorem 2.5.6 n times and theorem 2.5.4 once to get $p \Rightarrow q \vee p_1 \vee \cdots \vee p_n$. Since $\text{join-prime}(p)$ and $p \not\Rightarrow p_i$ for all i, it must be the case that $p \Rightarrow q$ as desired.
  - Case $\Longleftarrow$. Immediate by theorem 2.5.3.

- Case MIN-PICK. We have $\theta = (\theta', p' \Rightarrow q')$, $\min_\theta(p) = \min_{\theta'}(p \wedge q')$, $\text{join-prime}(p)$, and $p \Rightarrow p'$.

  - Case $\Longrightarrow$. Assume $\theta \models p \leq q$. We need to show $\min_{\theta'}(p \wedge q') \Rightarrow q$. Theorem 2.5.5 gives $\theta' \models p \wedge q' \leq q$. We can then apply the induction hypothesis to get the desired result.

- Case $\Longleftarrow$. Assume $\min_{\theta'}(p \wedge q') \Rightarrow q$. We need to show $\theta \models p \leq q$. The induction hypothesis gives $\theta' \models p \wedge q' \leq q$. Theorem 2.5.3 gives $\theta' \models p \leq p'$. Combining these with theorem 2.5.12 gives the desired result.

- Case MIN-FACTOR. We have $p = p_1 \vee \cdots \vee p_n$, $\min_\theta(p) = \bigvee_{i \in [n]} \min_\theta(p_i)$, and $\forall i \in [n] \,.\, \text{join-prime}(p_i)$.

  - Case $\Longrightarrow$. Assume $\theta \models p \leq q$. We need to show $\bigvee_{i \in [n]} \min_\theta(p_i) \Rightarrow q$. By the definition of $\vee$, we have $\theta \models p_i \leq q$ for all $i \in [n]$. The induction hypotheses then give $\min_\theta(p_i) \Rightarrow q$ for all $i \in [n]$. Finally, we use the definition of $\vee$ to get the desired result.

  - Case $\Longleftarrow$. Assume $\bigvee_{i \in [n]} \min_\theta(p_i) \Rightarrow q$. We need to show $\theta \models p \leq q$. By the definition of $\vee$, we have $\min_\theta(p_i) \Rightarrow q$ for all $i \in [n]$. The induction hypotheses then give $\theta \models p_i \leq q$ for all $i \in [n]$. Finally, we use the definition of $\vee$ to get $\theta \models p_1 \vee \cdots \vee p_n \leq q$.

□

Our NMIFC algorithm simply combines Algorithms 2.5.1 and 2.5.11.

Algorithm 2.5.14 (Uncompromised Label Check $\Theta \vdash \blacktriangledown\ell$)**.**

$$\frac{\theta_\rightarrow \vdash \min_{\theta_\leftarrow}(q) \leq p}{\langle \theta_\rightarrow, \theta_\leftarrow \rangle \vdash \blacktriangledown \langle p, q \rangle}$$

Equivalently, the rule defines the following decision procedure:

$$\mathsf{Uncompromised}(\langle \theta_\rightarrow, \theta_\leftarrow \rangle, \langle p, q \rangle) = \mathsf{ActsFor}(\theta_\rightarrow, \mathsf{Min}(\theta_\leftarrow, q), p).$$

That is, the procedure first computes the minimal representative of the integrity component under the integrity delegation context, and then checks, using the

acts-for procedure, whether that principal acts for the confidentiality component under the confidentiality delegation context.

Theorem 2.5.15 (Correctness of Uncompromised Label Check)**.** *Algorithm 2.5.14 terminates, and it is sound and complete with respect to theorem 2.4.5.*

Proof. We show soundness and completeness separately.

- Case Soundness. Assume $\theta_{\rightarrow} \vdash \min_{\theta_{\leftarrow}}(q) \leq p$. Pick $r = \min_{\theta_{\leftarrow}}(q)$. By theorem 2.5.8 (instantiated with $\min_{\theta_{\leftarrow}}(q) \Rightarrow \min_{\theta_{\leftarrow}}(q)$), $\theta_{\leftarrow} \models q \leq \min_{\theta_{\leftarrow}}(q)$. By theorem 2.5.7, $\theta_{\rightarrow} \models \min_{\theta_{\leftarrow}}(q) \leq p$.
- Case Completeness. Assume there exists $r \in \mathbb{P}$ such that $\theta_{\leftarrow} \models q \leq r$ and $\theta_{\rightarrow} \models r \leq p$. By theorem 2.5.8, $\min_{\theta_{\leftarrow}}(q) \Rightarrow r$. By theorem 2.3.4, $\theta_{\rightarrow} \models \min_{\theta_{\leftarrow}}(q) \leq r$. By transitivity, $\theta_{\rightarrow} \models \min_{\theta_{\leftarrow}}(q) \leq p$. Finally, by theorem 2.5.7, $\theta_{\rightarrow} \vdash \min_{\theta_{\leftarrow}}(q) \leq p$.

□

This algorithm checks whether the integrity component of a label is stronger than the weakest principal from the label's confidentiality equivalence class.

We demonstrate our algorithm on the MPC example from fig. 2.2. The asymmetric delegation context is given by $\Theta = \langle \cdot, \mathrm{Alice} \leq \mathrm{Bob}, \mathrm{Bob} \leq \mathrm{Alice} \rangle$. To verify whether `w` in fig. 2.2 is compromised ($\Theta \models \blacktriangledown \mathrm{Alice} \sqcup \mathrm{Bob}$), we run:

$$\mathrm{Alice} = \mathrm{Bob} \vdash \mathrm{Alice} \vee \mathrm{Bob} \Rightarrow \min_{(\cdot)}(\mathrm{Alice} \wedge \mathrm{Bob})$$

The judgment above holds by theorem 2.5.1. Indeed, the strongest principal from $[\mathrm{Alice} \vee \mathrm{Bob}]_{\theta_{\leftarrow}}$ is $\mathrm{Alice} \wedge \mathrm{Bob}$, which is no weaker than $\mathrm{Alice} \wedge \mathrm{Bob}$.

2.6 Label Inference and Bounded Polymorphism

In practical IFC systems, label inference is an important way to avoid redundant user annotations, since it is secure to infer labels of intermediate computations from their inputs and outputs. Prior work on IFC label inference uses Hindley–Milner-style algorithms: first, the compiler generates a system of constraints over existential label variables according to the typing rules; second, it solves for the existential label variables using a constraint solver. Without a delegation context, the constraint solver is purely syntactic.

An obvious benefit of using a delegation context is that it allows label inference to consider delegations. A more subtle payoff is that delegation contexts can be used as contexts to create environments for more effective and modular implementations of label inference algorithms that support *polymorphism*.

We extend the Viaduct compiler [6] with our label algebra, and we provide a novel implementation of label inference and bounded label polymorphism that takes advantage of delegation contexts and the sound and complete algorithms from §2.5. In this section, we first review prior approaches to polymorphism in IFC compilers and highlight their limitations. We then show that our implementation in Viaduct achieves both a more modular and extensible design and a provably complete type system in the sense that it accepts all semantically secure programs.

2.6.1 Prior Approaches to Polymorphism in IFC Compilers

Because a delegation context introduces equational assumptions between labels, it creates a natural setting for symbolic solvers: a procedure can reason once under those assumptions and then reuse the resulting symbolic summary across

many instantiations. From this perspective, the symbolic solver enables a better implementation of *bounded label polymorphism*.

Bounded label polymorphism allows users to write library code reusable at different security levels. As in traditional type systems, allowing code that is generic over labels increases expressiveness significantly. In a conventional polymorphic type system, a function is checked once with type variables standing for unknown types. The function declaration records the bounds or constraints on those variables, and each call site later instantiates the variables with concrete types. This lets the compiler reuse one symbolic type summary instead of rechecking the function body from scratch at every call. Annotation burden on users can be further reduced by assuming information flow from function arguments to return values by default.

The compiler implementation of the IFC-based language Viaduct [6] supported polymorphism by creating specialized monomorphic copies of polymorphic functions at each call site. This approach requires the compiler to walk the call graph and create copies of monomorphized functions while generating constraints. As a result, the compiler generates a single constraint system for the whole program, which can be large and difficult to solve. Some direct downsides of the approach are poor error localization, inefficient constraint solving, and the inability to support runtime polymorphism. In addition, the implementation is inextensible and fragile because it uses a single interprocedural algorithm to handle recursive calls, cache specialized functions, and generate constraints at the same time.

Other existing IFC-based languages, such as Jif [75], Fabric [63], and Flow Caml [95], support bounded label polymorphism by doing type inference per function. In each function, both polymorphic label variables and label variables

are treated as label constants. The compiler generates a list of syntactic flows-to constraints ($\sqsubseteq$), using both the constraints generated from statements and the bounds on polymorphic variables. Then it solves the constraints using a standard dataflow algorithm. However, the constraints generated by this approach are not complete with respect to the semantic definition of flows-to. At a high level, it is an algorithm over the free label algebra instead of the quotient algebra of labels under the delegation context; and it misses the flows-to relation between labels that are not explicitly related by the polymorphic label constraints.

Our new implementation of label inference and bounded label polymorphism in Viaduct uses polymorphic label constraints as delegation contexts for the function body, and the constraint solver directly operates over the algebraic model of labels and delegation.

2.6.2 Bounded Label Polymorphism

In this section, we show how we can do label inference directly over the algebraic model of labels using the algorithms from §2.5.

In fig. 2.5, the polymorphic function `average` is annotated with the label variables and constraints generated by inference, and applied in `main` to arguments with different security labels.

The declaration of `average` introduces polymorphic label variables `X`, `Y`, and `Z`. Its bound $\texttt{X} \sqcup \texttt{Y} \sqsubseteq \texttt{Z}$ is the symbolic condition that callers must satisfy: the result label must be at least as restrictive as both argument labels. Inside the function body, label inference creates an ordinary label variable `lo` for the result of the addition, with generated constraints $\texttt{X} \sqsubseteq \texttt{lo}$, $\texttt{Y} \sqsubseteq \texttt{lo}$, and $\texttt{lo} \sqsubseteq \texttt{Z}$.

The important implementation choice is that polymorphic label variables are treated as local label constants while checking the function body. Thus the

```
1  host Alice, Bob, Chuck
2  assume Alice = Bob for integrity
3  assume Bob = Chuck for integrity
4
5  fun average[X, Y, Z](a: int{X}, b: int{Y}): int{Z}
6    where (X ⊔ Y ⊑ Z)
7  {
8    val r: int{lo} = (a + b) / 2
9    // X ⊑ lo, Y ⊑ lo
10   return r
11   // lo ⊑ Z
12 }
```

```
10 fun main() {
11   val a: int{la} = Alice.input
12   // Alice ⊑ la
13   val b: int{lb} = Bob.input
14   // Bob ⊑ lb
15   val c: int{lc} = Chuck.input
16   // Chuck ⊑ lc
17
18   val r1: int{lr1} = average(a{X1}, b{Y1}){Z1}
19   // la ⊑ X1, lb ⊑ Y1, Z1 ⊑ lr1
20   // X1 ⊔ Y1 ⊑ Z1
21   val r2: int{lr2} = average(b{X2}, c{Y2}){Z2}
22   // lb ⊑ X2, lc ⊑ Y2, Z2 ⊑ lr2
23   // X2 ⊔ Y2 ⊑ Z2
24 }
```

Figure 2.5: Generated label variables and constraints for the polymorphic `average` example.

solution for `lo` can mention `X` and `Y`, producing a symbolic summary for `average` rather than a monomorphic copy. At each call site in `main`, the compiler creates fresh parameter label variables `X1`, `Y1`, `Z1` and `X2`, `Y2`, `Z2`. The generated constraints connect the actual argument labels to these parameter variables, connect the return parameter to the result label, and include a fresh copy of the function bound. For example, the first call generates `la` $\sqsubseteq$ `X1`, `lb` $\sqsubseteq$ `Y1`, `Z1` $\sqsubseteq$ `lr1`, and `X1` $\sqcup$ `Y1` $\sqsubseteq$ `Z1`.

This organization makes the constraint system modular. Each function is solved in its own delegation context, where its polymorphic bounds act like local assumptions. The call site then checks that those assumptions hold for the labels actually passed to the function. Compared with the naive approach of repeatedly specializing and rechecking the callee during type checking, this produces stable symbolic summaries for library functions and smaller constraint systems for callers.

After inference, Viaduct still performs function specialization before protocol selection, because later compiler phases operate on monomorphic functions. The specialization procedure traverses the call graph with a cache keyed by the function name and the inferred instantiation of its polymorphic label variables. If a call site requests an instantiation already in the cache, the existing monomorphic function is reused. Otherwise, the compiler creates one new monomorphic copy and continues the worklist traversal from that copy. This cache gives a minimum-sized set of monomorphic functions for the inferred program and handles recursive calls by the same fixed-point behavior, rather than with a separate recursion case.

$$
\begin{array}{lll}
\text{L. Constants} & \ell & \\
\text{L. Variables} & Y & \\
\text{L. Expressions} & L & ::= \ \ell \mid Y \mid L^{\pi} \mid L_1 \sqcup L_2 \mid L_1 \sqcap L_2 \mid L_1 \vee L_2 \mid L_1 \wedge L_2 \\
\text{L. Constraints} & C & ::= \ L_1 \sqsubseteq L_2 \mid \blacktriangledown L \\
\text{Projections} & \pi & \in \ \{\rightarrow, \leftarrow\}
\end{array}
$$

Figure 2.6: Syntax of label constraints.

$$
\begin{array}{lll}
\text{P. Constants} & p & \\
\text{P. Variables} & Y^{\pi} & \\
\text{P. Expressions} & P^{\pi} & ::= \ p \mid Y^{\pi} \mid P_1^{\pi} \vee P_2^{\pi} \mid P_1^{\pi} \wedge P_2^{\pi} \mid p_1 \rightarrow P_2^{\pi} \mid \min_{\pi'}(P^{\pi'}) \\
\text{P. Constraints} & D & ::= \ P_1^{\pi} \Rightarrow^{\pi} P_2^{\pi}
\end{array}
$$

Figure 2.7: Syntax of principal constraints.

2.6.3 Constraint Solver

We describe a novel constraint solver that computes the minimum-semantic-authority solution to constraint systems with delegation contexts. Minimum-authority solutions are desirable because they allow systems to choose cheaper security enforcement mechanisms [6, 21, 39, 40, 108, 111].

Label and Principal Constraints

Figure 2.6 gives the syntax of the label constraint language. Expressions in the constraint language include label constants, label variables, authority projections, and standard lattice operations. A constraint either asserts that an expression flows to another or asserts that an expression is uncompromised. We translate constraints over labels to constraints over principals to leverage algorithms from §2.5. Figure 2.7 gives the syntax of the principal constraint language. The syntax includes a principal variable Y^{π} for each combination of label variable Y and projection π. That is, $Y^{\rightarrow}$ represents the confidentiality of Y, and $Y^{\leftarrow}$ represents

$\boxed{[\![C]\!] = D_1, \ldots, D_n}$

$$[\![L_1 \sqsubseteq L_2]\!] = [\![L_2]\!]_{\rightarrow} \Rightarrow^{\rightarrow} [\![L_1]\!]_{\rightarrow},\ [\![L_1]\!]_{\leftarrow} \Rightarrow^{\leftarrow} [\![L_2]\!]_{\leftarrow}$$

$$[\![\blacktriangledown L]\!] = [\![L]\!]_{\leftarrow} \Rightarrow^{\leftarrow} \min_{\rightarrow}([\![L]\!]_{\rightarrow})$$

$\boxed{[\![L]\!]_\pi = P^\pi}$

$$[\![L_1 \sqcup L_2]\!]_\pi = [\![(L_1 \wedge L_2)^{\rightarrow} \wedge (L_1 \vee L_2)^{\leftarrow}]\!]_\pi \qquad [\![\langle p, q\rangle]\!]_\pi = \begin{cases} p & \text{if } \pi = \rightarrow \\ q & \text{if } \pi = \leftarrow \end{cases}$$

$$[\![L_1 \sqcap L_2]\!]_\pi = [\![(L_1 \vee L_2)^{\rightarrow} \wedge (L_1 \wedge L_2)^{\leftarrow}]\!]_\pi \qquad [\![Y]\!]_\pi = Y^\pi$$

$$[\![L_1 \vee L_2]\!]_\pi = [\![L_1]\!]_\pi \vee [\![L_2]\!]_\pi \qquad [\![L^{\pi'}]\!]_\pi = \begin{cases} [\![L]\!]_\pi & \text{if } \pi = \pi' \\ 1 & \text{if } \pi \neq \pi' \end{cases}$$

$$[\![L_1 \wedge L_2]\!]_\pi = [\![L_1]\!]_\pi \wedge [\![L_2]\!]_\pi$$

Figure 2.8: Translating label constraints to principal constraints.

Y's integrity.

We index expressions P^π by the component π they represent. This prevents expressions like $Y_1^{\rightarrow} \wedge Y_2^{\leftarrow}$, whose components are mixed. Expressions include principal constants and principal variables, as well as principal-lattice operations $\vee$ and $\wedge$. The operation $\rightarrow$ is called the *relative pseudocomplement* of the meet operation: $p_1 \rightarrow p_2$ is defined as the minimum-authority principal p such that $p_1 \wedge p \Rightarrow p_2$. We use $\rightarrow$ to solve constraints of the form $Y^\pi \wedge p_1 \Rightarrow^\pi P_2^\pi$.The $\min_\pi()$ operation allows mixing integrity and confidentiality components; we use it when solving for labels that must be uncompromised. Principal constraints have the form $P_1^\pi \Rightarrow^\pi P_2^\pi$, which stands for $\theta_\pi \models P_1^\pi \leq P_2^\pi$.

Figure 2.8 gives rules for translating label constraints to principal constraints. The definition of $[\![L]\!]_\pi$ is a straightforward encoding of the label-lattice operations. Using $[\![L]\!]$, we translate a flows-to ($\sqsubseteq$) constraint to two acts-for ($\Rightarrow$) constraints, one for each label component. The constraint $\blacktriangledown L$ follows from Algorithm 2.5.14.

Solving Principal Constraints

Our constraint solver requires the left-hand side of each constraint to be atomic (a constant or a variable), that is, constraints of the form $p_1 \Rightarrow^\pi P_2^\pi$ and $Y_1^\pi \Rightarrow^\pi P_2^\pi$. We exhaustively apply the equational axioms of Heyting algebra (e.g., associativity, absorption, distributivity, etc.) until no left-hand side of any constraint can be further simplified. Because equational axioms are syntactic rewrites, they do not change the constraint system over the underlying algebra. Moreover, this process always terminates, and ensures that the left-hand side of each constraint either is atomic or contains a meet ($\wedge$).[6]

Constraint solving fails if the left-hand side of any constraint contains a meet, since such systems do not have unique solutions. For example, the system $Y_1 \wedge Y_2 \Rightarrow \mathrm{Alice}$ has no minimal solution: we can assign $\{Y_1 \mapsto \mathrm{Alice}, Y_2 \mapsto 1\}$ or $\{Y_1 \mapsto 1, Y_2 \mapsto \mathrm{Alice}\}$, but neither solution is better than the other. Compiler implementations need either to restrict the syntax of polymorphic constraints or to report errors during constraint solving.

Once the simplification process succeeds, we extend the algorithm of Rehof and Mogensen [83] for iteratively solving semilattice constraints. We initialize all principal variables to 1, and use unsatisfied constraints to update variables repeatedly until a fixed point is reached, using the rule:

$$\text{given} \quad Y^\pi \Rightarrow^\pi P^\pi, \quad \text{set} \quad Y^\pi := Y^\pi \wedge \text{current-value}(\Theta, P^\pi),$$

where $\text{current-value}(\Theta, P^\pi)$ is the value of P^π according to the current assignment.

Note that constraints with constants p on the left-hand side are ignored during the fixed point computation. Once a fixed-point solution is reached, we perform

[6]Translation rules in fig. 2.8 never generate constraints with $\rightarrow$ or $\min_\pi(\cdot)$ on the left-hand side, and constraint simplification eliminates all joins ($\vee$) on the left.

the following check for each constraint with a constant left-hand side:

$$\text{given} \quad p \Rightarrow^{\pi} P^{\pi}, \quad \text{check} \quad \theta_{\pi} \models p \leq \text{current-value}(\Theta, P^{\pi}).$$

We state and prove in the technical report [85] that this process terminates with the minimum-authority solution if all such constraints are satisfied; otherwise, there is no valid solution.

Theorem 2.6.1 (Correctness of Constraint Solver)**.** *Constraint procedure always terminates with either minimal-authority solution or correctly fails.*

Proof. We note that the result of the algorithm is a map M from principal variables Y^{π} to principal constants p.

Now define a well-founded relation $\succeq$ over the domain of maps M from Y^{π} to p where the update rule is decreasing:

$$M \succeq M' \text{ if } \forall Y^{\pi} \,.\, M(Y^{\pi}) \Rightarrow M'(Y^{\pi})$$

This is indeed a well-founded relation in practice when M is a finite map and $\mathbb{P}$ is a finite lattice.

Then we formalize the update rule as a function f between maps M given by (here D is the set of constraints):

$$f(M)(Y^{\pi}) = \bigwedge_{Y^{\pi} \Rightarrow^{\pi} P^{\pi} \in D} M(Y^{\pi}) \wedge M(P^{\pi})$$

f is a monotonic function by construction. Since our algorithm repeatedly applies this function to M, this forms a chain of $f^i(M)$, which, by Knaster–Tarski theorem, reaches the greatest fixpoint. Thus, this algorithm terminates with least-authority solution when it exists. □

2.6.4 Implementation

We modified the parser of the Viaduct compiler [6] with the delegation syntax, and extended its static analysis procedure with our label inference algorithm. [7]

The implementation follows the same separation between syntax and semantics used in the formal development. Syntactic label expressions are kept as surface-language syntax trees, while semantic labels implement lattice operations, semantic acts-for checks under delegation contexts, and the minimum-authority operations used by the solver.

Because the constraint solver computes symbolic summaries for polymorphic functions, Viaduct can type-check functions modularly before the later monomorphization pass described in §2.6.2. This has several practical benefits. First, functions can be checked independently, and therefore concurrently, so the compiler can catch static type errors before every call site has been specialized. Second, this supports partial type checking even when later phases still perform monomorphization, which is the relevant form of runtime polymorphism for Viaduct's implementation pipeline. Third, each function generates a smaller constraint system, which improves error localization: failures point to the function summary or call-site instantiation that introduced the unsatisfied constraint. Fourth, library functions have stable symbolic summaries, so the design is more extensible: adding new delegations, hosts, or label forms does not require rewriting the checker or re-solving unrelated function bodies. Finally, the cached specialization procedure creates only the monomorphic copies that are actually needed, and recursive calls are handled by the same fixed-point behavior as other repeated call-graph edges.

For better usability, explicit polymorphic label declarations and constraints

[7] Code available at: https://github.com/apl-cornell/viaduct.

```
1  host Alice : {A ∧ B← ∧ C←}
2  host Bob   : {B ∧ A← ∧ C←}
3  host Chuck : {C ∧ A← ∧ B←}
4
5  fun average1(a: int{x1}, b: int{y1}): int{z1}
6  {
7    val r: int{lo} = (a + b) / 2
8    // x1 ⊑ lo, y1 ⊑ lo
9    return r
10   // lo ⊑ z1
11 }
12
13 fun average2(a: int{x2}, b: int{y2}): int{z2}
14 {
15   val r: int{lo} = (a + b) / 2
16   // x2 ⊑ lo, y2 ⊑ lo
17   return r
18   // lo ⊑ z2
19 }
20
21 fun main() {
22   val a: int{la} = Alice.input
23   // la = A ∧ B← ∧ C←
24   val b: int{lb} = Bob.input
25   // lb = B ∧ A← ∧ C←
26   val c: int{lc} = Chuck.input
27   // lc = C ∧ A← ∧ B←
28
29   val r1: int{lr1} = average1(a, b)
30   // la ⊑ x1, lb ⊑ y1, z1 ⊑ lr1
31   val r2: int{lr2} = average2(b, c)
32   // lb ⊑ x2, lc ⊑ y2, z2 ⊑ lr2
33 }
```

Figure 2.9: The running `average` example in the older host-parameterized Viaduct syntax. The old compiler materializes separate specialized copies of `average`, and adding `Chuck` forces every host label to mention an extra integrity component.

are optional in Viaduct. When they are omitted, the compiler declares a fresh polymorphic variable for each input and output parameter and generates a default constraint that requires all inputs to flow to the output.

Empirically, type inference is fast. On all Viaduct benchmarks, it terminates within 300 milliseconds on a 2023 MacBook Air with an M2 chip.

Example: Average Function

To demonstrate our implementation of label inference and bounded polymorphism, and compare it with the older Viaduct syntax, we show the running `average` example in the older Viaduct syntax in fig. 2.9.

Concretely, the old procedure traverses the call graph and specializes a polymorphic function at each call site before type checking is complete. In fig. 2.9, the call `average1(a, b)` produces one copy with local label variables `x1`, `y1`, and `z1`, while the call `average2(b, c)` produces another copy with `x2`, `y2`, and `z2`. The compiler then solves constraints over this expanded monomorphic program, so each new call site may force additional copying and constraint generation.

Our current implementation follows the opposite order. It first type-checks `average` once in a symbolic environment, using one set of polymorphic parameters and their bounds as a local delegation context, and records the resulting symbolic summary. Each call site is then checked only against that summary, and the later specialization pass materializes monomorphic copies only after inference has already succeeded. Thus the old implementation duplicates function bodies to obtain per-call-site constraints, whereas our implementation first derives one reusable summary and duplicates functions as needed.

2.7 Related Work

Expressive trust delegation has been widely investigated in access control and capability systems [61, 89], where delegation is called *role hierarchy* [89]. Such systems support role adoption, which is a form of dynamic delegation. However, access control systems generally do not connect to information flow security properties. Many prior IFC systems have delegation abstractions compatible with our framework, but they either lack support for asymmetric delegation [6, 18, 105], or do not explore its effect on hyperproperties [8, 62, 73, 74, 106].

Our inference algorithm differs from prior implementations of syntax-directed IFC label inference [6, 75, 110] by operating directly over the underlying algebra. This algebraic approach enhances extensibility: adding principals or delegations does not invalidate existing analysis. Dolan [35] proposes an inference algorithm for an expressive algebraic type system with type constructors and recursive function types. However, because the type system incorporates expressive type constructors, including function and recursive types, the semantic model does not have a simple algebraic structure. As a result, inference does not produce easily interpretable representations of types. In contrast, our label system balances expressiveness with a simple and effective inference algorithm.

FLAM [8] proposes a label model and defines *robust authorization* to address security vulnerabilities arising from dynamic delegation. However, robust authorization is a proof-theoretic definition with no semantic model. Arden and Myers [9] propose the FLAC calculus, based on the FLAM label model, and prove robust declassification for a language that downgrades using dynamic delegation. Similarly, FLAC has no attacker semantics. Cecchetti et al. [18] define NMIF, but their system cannot explicitly express delegations between atomic principals.

2.8 Conclusion and Future Directions

We present an algebraic semantic framework for IFC labels that models asymmetric delegation, along with sound and complete algorithms that enforce security properties and infer security annotations. Our approach provides a solid foundation for building modular, expressive, and extensible IFC systems.

CHAPTER 3

TYPE-CONFUSION SECURITY

Static type systems provide a powerful way to enforce security properties. They allow programmers to reason about code in a high-level language and rule out attacks before the code runs, often with little or no overhead from runtime checking. In type systems for information flow control (IFC), types serve as expressive security specifications for code: they describe confidentiality and integrity requirements, and identify contexts in which code may execute securely. Static information flow type systems are powerful enough to enforce complex security properties such as noninterference [47, 90].

However, purely static reasoning is fragile in open, decentralized systems. For example, smart contracts, federated services, and distributed object systems routinely interact across trust boundaries. In such settings, trusted code may call code supplied by an adversarial principal, receive values from an untrusted principal, or invoke remote procedures whose implementations are controlled elsewhere. The adversary need not obey the source language's typing discipline. It may supply a value whose advertised type does not match its runtime behavior, or a function whose advertised security type does not match the contexts in which it may safely be called. We call this adversary-induced mismatch between the expected and the actual type *type confusion*.

The security implications of type confusion depend on the type system. A simple form of type confusion involves unchecked casts between base types: a runtime value may be used as an integer, boolean, object reference, or function pointer even though it was not produced as a value of that type. Such type confusion is dangerous in low-level languages like C and C++, where it can violate memory safety and control-flow integrity.

A more subtle problem arises when types themselves express security policies. In such cases, type confusion can also lead to well-known but poorly understood security vulnerabilities. For example, we show that *confused deputy attacks (CDA)* [52] can be understood as type confusion for function values whose types express information flow policies. Similarly, *reentrancy attacks* [19] can be understood as type confusion on types that bound a function's control-flow effects. In both cases, the attacker exploits a type-confused function pointer to induce unintended control flow in trusted code, so purely static IFC is insufficient in open-world settings.

Type confusion is not merely hypothetical. Recent high-profile security breaches in blockchain smart contracts have been attributed to CDAs and reentrancy attacks [26–29] that exploit a mismatch in expectations about behavior of callback code written in Solidity. Although Solidity does not permit developers to statically specify information flow policies, information flow enforcement alone does not solve these security issues. Similar issues arise in Fabric [63], a rich, state-of-the-art distributed IFC system in which computation spans mutually distrusting nodes.

Unlike Fabric, our goal is to isolate the core semantic principle behind such dynamic mechanisms and identify the minimal runtime enforcement needed to prevent type confusion. Rather than model mobile code, persistent objects, transactions, stores, and codebases, we study a minimal RPC-style calculus with stateful principals and first-class function pointers. The runtime assumptions are minimal and compatible with modern smart-contract virtual machines such as the EVM: at a call boundary, the callee learns no more than the caller's identity.

Our main technical contribution is a real–ideal definition of *type-confusion security*. In the real world, attackers may be ill-typed (lie about types), but trusted

code is statically typed and protected by runtime checks. In the ideal world, attackers are well-typed and no runtime checks are needed. A program is secure against type confusion if every real-world attacker can be simulated by some well-typed ideal-world attacker with the same observable behavior. Equivalently, lying about types gives the attacker no more power than it would already have in the well-typed ideal world.

This simulation-based definition provides a robust hyperproperty preservation (RHP) guarantee [5]: any hyperproperty [24] proved for the ideal well-typed system continues to hold in the real system with ill-typed attackers. Thus programmers may reason about security in the simpler ideal world, even though deployed code interacts with adversaries that do not respect the type system. Our proof follows the standard secure compilation pattern of context-based backtranslation [33, 77]: we show the existence of a translation from an arbitrary real attacker to a well-typed ideal attacker that simulates the real attacker's observable behavior.

This chapter makes the following contributions:

- We define type-confusion security as a real–ideal simulation property for open systems where adversaries may lie about types.
- We identify two instances of the definition: base-type confusion, handled by deterministic runtime checks, and IFC function-type confusion, adapting and extending the authority-lock mechanisms of SeRIF [19] and also adding novel runtime checks at call boundaries.
- We develop core calculi λ_{tc}, λ^{r}_{tc}, and λ^{b}_{tc} for the IFC-typed calculus, base-typed calculus, and untyped calculus, respectively. The calculi model decentralized programs with stateful principals, remote function calls, and first-class function pointers.

- We prove type-confusion security by a type-directed bisimulation argument, using contextual backtranslation [33, 77] to establish preservation of all ideal-world hyperproperties.
- We instantiate the theorem to prove security against confused deputy attacks, reentrancy attacks, and noninterference in the ideal world.

The rest of the chapter is structured as follows. §3.1 introduces motivating examples and the key intuition behind the runtime checks. §3.2 formalizes type-confusion security and its connection to robust hyperproperty preservation. §3.3 defines the ideal and real calculi. §3.4 proves the main simulation theorem. §3.6 applies the theorem to CDA, reentrancy, and noninterference. §3.7 discusses related work, and §3.8 concludes.

3.1 Motivation

3.1.1 Confused Deputy Attacks

The confused deputy attack (CDA) [52] is a classic security vulnerability in which the attacker tricks a trusted party (called the deputy) into misusing its authority.

The canonical example of a CDA is shown in Figure 3.1. Here, a compiler service is the confused deputy (a modern equivalent is a smart contract token exchange service). In an honest execution, the user calls the compiler with the source code and an output file writer; after compiling the source code, the compiler writes compiled code to the output file and records the fee in the billing file. However, an attacker may trick the compiler by passing the billing file itself as the output file, causing the billing file to be overwritten with compiler output.

CDAs have long been described in the literature [52] and mitigation mechanisms exist in systems where a centralized, trusted runtime is available [82], but

```
User.main() :=
    Compiler.run{U}(src, outwriter)
```

```
Attacker.main() :=
    Compiler.run{U}(src, billwriter)
```

```
Compiler.run{U}(code, outwriter:{U}) :=
    out, cost := compile{T}(code)
    outwriter{U}(out)
    billwriter{T}(cost)
```

Figure 3.1: The classic Confused Deputy Attack (CDA) example, annotated with information flow labels. For simplicity, only function types and function call sites are annotated with pc (control flow) labels.

they continue to cause costly security breaches in modern, decentralized settings such as blockchain smart contracts [26–28].

A *capability system* [32] is a traditional approach to preventing CDAs. In a capability system, a trusted entity (such as an operating system) ensures that resources like data and pointers can only be accessed through unforgeable *capability tokens*. A capability system could require that the compiler possess explicit, distinct capabilities for writing to the bill writer and the output file writer, and that the write operation requires sufficiently powerful capabilities to be passed to it. The call to the output file writer should only be passed the capability for the output file—which would not provide enough authority to write to the bill writer. There are two limitations to this approach. First, nothing forces the programmer to select the correct capability for each call, so mistakes are still easy to make. Further, capabilities must be threaded through the computation, so care is required to ensure that they are not accidentally leaked to the attacker.

Rajani et al. [82] addressed the limitations by *provenance tracking*, which dynamically tracks dependencies between capabilities to prevent explicit and implicit leakage. However, just like prior solutions, this approach relies on a trusted

runtime that is often not available in decentralized systems. Implementing such a runtime requires costly cryptographic primitives, and creates incompatibilities with existing software ecosystems. Ideally, we want to move as much enforcement as possible to static checking for performance while ensuring the security of trusted code even when interacting with untrusted or legacy code that does not respect the static typing discipline.

To address the limitations of capability systems, we propose a language-based solution in which function types include security annotations. In fig. 3.1, all functions are annotated with security labels (in braces), where `T` represents "trusted" and `U` represents "untrusted". This annotation, the *external pc (program counter) label*, governs the contexts in which each function may be called. A function labeled `T` may only be called from trusted contexts, but a function labeled `U` may be called from anywhere. We can think of this label as a form of static access control, but unlike in capability systems, information flow analysis is used to ensure that these security labels are consistent and cannot be chosen arbitrarily.

For example, this consistency implies a subtyping relationship on function types: a `U`-labeled function is a subtype of a `T`-labeled function pointer because it can be used in more contexts. However, the reverse is not true, even though trusted information can be used where untrusted information is expected. The reason is that function subtyping is contravariant in argument types [17].

This static discipline prevents CDA in an ideal system where all parties respect the typing rules: the attacker is ill-typed because it calls the compiler by passing a `T`-labeled function argument (`billwriter`) to a `U`-labeled function parameter (`outwriter`).

However, an attacker who does not respect static typing can mount the

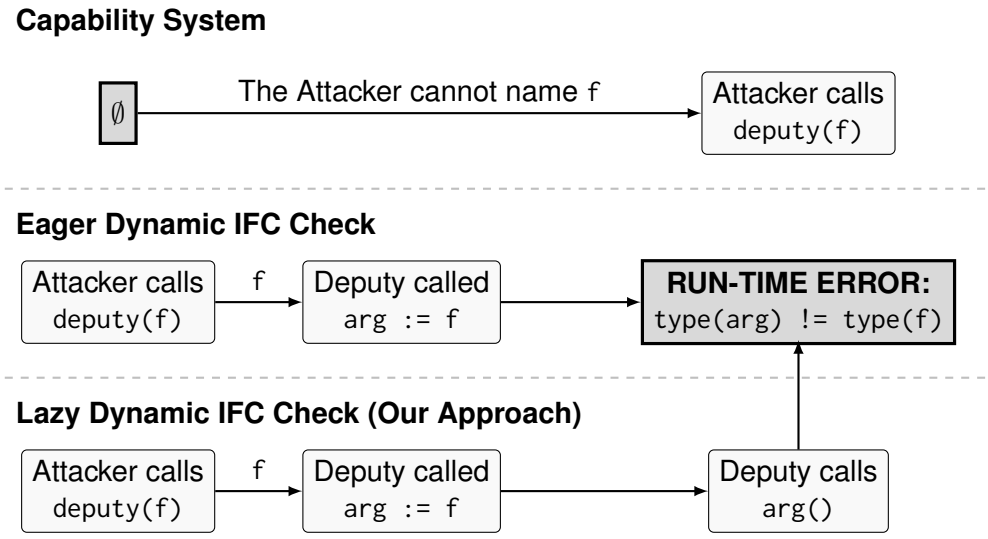

Figure 3.2: Visual illustration of the execution trace of a CDA attack under different enforcement mechanisms. The eager enforcement mechanism performs a dynamic type check upon receiving the function pointer, whereas the lazy enforcement mechanism checks the pointer when it is used (called). Without a centralized authorization engine, the eager enforcement mechanism requires an honest attacker, making it impractical in settings like blockchain.

original attack; in the real world, runtime checking is needed to prevent CDAs. Perhaps surprisingly, CDAs can be prevented with a simple runtime check based on information flow control. Our new defense is that the caller sends its (statically known) current control-flow label to the callee, and the callee verifies that the caller's control-flow label matches its own label. For example, in fig. 3.1, the attacker passes `billwriter` as the argument to the compiler. At line 3 of the compiler code, the compiler notifies the callee `billwriter` that it is being called from a `U`-labeled context. As the callee `billwriter` expects calls only from `T`-labeled contexts, it rejects the call at run time.

Note that the proposed runtime check is "lazy"—it happens when the function is called, not when the function pointer is created or passed around. This is because the type information of function pointers is often erased at runtime, making it impossible to dynamically type-check pointers without expensive

```
User.main() :=
    Compiler.run{U}(src, outwriter)
```

```
Attacker.main() :=
    Compiler.run{U}(src.first(), reenter)

Attacker.reenter{U◁T}(out) :=
    outwriter{U}(out)
    Compiler.run{U}(src.next(), reenter)
```

```
Compiler.run{U◁T}(code, outwriter:{U◁U}) :=
    out, cost := compile{T}(code)
    outwriter{U}(out)
    billwriter{T}(cost)
```

Figure 3.3: Reentrancy attack against the compiler example. In addition to an external pc, functions are annotated with a bounding label that bounds the maximum effect during function execution.

remote calls. fig. 3.2 illustrates the execution trace of the example CDA attack under different enforcement mechanisms.

3.1.2 Reentrancy Attacks

CDAs are control-flow attacks that exploit unexpected control flow. However, with the typing discipline seen so far, other control-flow attacks remain possible. Figure 3.3 shows an example of a *reentrancy attack* [29], where the attacker causes the deputy to be called again before it finishes its first call. Consider the attack shown in fig. 3.3, where the attacker provides a malicious output file writer that calls back to the compiler after writing the output file. This attack turns `Compiler.run` and `Attacker.reenter` into mutually recursive functions that infinitely compile code without billing the user.

As observed by Cecchetti et al. [19], a reentrancy attack happens when functions rely on invariants that do not hold as a result of unexpected control flow.

In the compiler example, the output writer is expected to execute at an untrusted level, but its reentrant call to `Compiler.run` happens at a trusted level. In general, insecure reentrancy happens when the control-flow level moves from a trusted level to an untrusted level and back again to trusted before the first trusted call returns. Such control-flow patterns only happen when calling an *auto-endorsing* function [19]: a function whose external pc label is less trusted than the level at which its code executes. Such functions are essential for expressiveness in decentralized systems. The insight of Cecchetti et al. [19] is that reentrancy attacks, at their core, are caused by unexpected control-flow endorsement via auto-endorsing functions.

We can statically enforce secure reentrancy by annotating functions with a *bounding label* that upper-bounds the level of effects during function execution. A function with a `U` bounding label never endorses its control flow above `U` (the subtype), whereas one with a `T` bounding label may call auto-endorsing functions freely (the supertype). For example, above, all functions are annotated with a bounding label. The function `Compiler.run` is an auto-endorsing function because it endorses control flow from `U` to `T`, with bounding label equal to `T`. Although `Attacker.reenter` is not auto-endorsing, its bounding label is also `T` because it calls `Compiler.run`; unlike the external pc label, which governs access to the function, the bounding label enforces reentrancy statically by constraining the maximum effect of the function.

Once the bounding label is added to function signatures, the body of the reentrancy attacker becomes ill-typed—it calls the compiler with an ill-typed argument labeled {U◁T}. Thus, the reentrancy attack is also a form of type confusion.

To prevent reentrancy in the real world with ill-typed attackers, we add a

second runtime mechanism, adapting the dynamic lock system of Cecchetti et al. [19]: when a function is called, it locks its own level of control flow until it returns. Any auto-endorsing call that endorses control flow up to or beyond the lock is rejected. In the compiler example, `Compiler.run` locks the `T` level when it is called. `Compiler.run` then calls an untrusted function `Attacker.reenter`, which calls back to `Compiler.run` before the first call to it returns. The dynamic lock prevents this second, reentrant call.

Our treatment differs from that of Cecchetti et al. [19] in both presentation and purpose. SeRIF defines a single low-level language in which programmers configure both static bounding and dynamic locks. By contrast, our contribution is to factor the same underlying reentrancy mechanism into two worlds: an ideal language with static enforcement only, and a real-world language with dynamic enforcement only. This factorization isolates exactly the dynamic checks needed to tolerate ill-typed attackers, while also identifying the minimal function signatures needed to enforce secure reentrancy statically in the ideal world. As a result, we can state and prove our type-confusion security theorem as a real–ideal simulation. To keep that proof focused, we further omit *tail reentrancy*. We expect that the same style of result could be extended to a language with tail reentrancy, but doing so would add proof complexity without changing the main contribution of this chapter.

3.1.3 Type Confusion and IFC Background

In both aforementioned attacks, the real-world attacker uses type-confused function pointers to cause unintended control flow in trusted code. In the case of CDA, the adversary tricks a trusted party into authorizing an unsafe call using a type-confused function pointer; for reentrancy, the adversary bypasses the

invariant on control-flow integrity using a type-confused function pointer. Our key observation is that in security-typed systems, dangerous type confusion arises when trusted code calls type-confused function values. Some classic CDA examples might seem not to fit this description, since they involve writing to references guarded by access control policies. However, a reference can be viewed as a pair of functions: a getter and a setter; with this view, the classic examples can also be understood as involving type-confused functions.

Of course, attackers may also lie about *base types*—the input–output types of functions, and the types of data. However, this base-type confusion is not as challenging as security-type confusion. Base-type confusion can be prevented from compromising security by ensuring that values of the wrong type are treated deterministically as values of the expected type. §3.4.2 discusses this simple defense in more detail.

IFC Label Model

We assume a security-typed language in which each type τ includes an information flow label that governs the use of values of that type. In this chapter, we adopt the label model of [84], where the principals $p \in \mathbb{P}$ in the decentralized system generate a lattice of security labels. Given a type τ, we write $\mathsf{lbl}(\tau)$ to denote the label component of the type.

The *flows-to* order $\sqsubseteq$ represents the direction of secure information flow. For simplicity, we focus only on enforcing integrity, so $\ell_1 \sqsubseteq \ell_2$ when ℓ_1 is at least as trusted as ℓ_2. The bottom element $\bot$ represents the most trusted label, and the top element $\top$ represents the most untrusted label. Join $\sqcup$ and meet $\sqcap$ represent the least upper bound and the greatest lower bound of two labels, respectively.

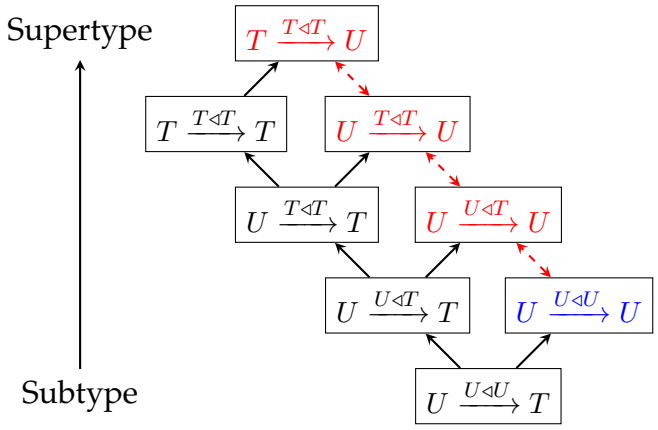

Figure 3.4: The subtyping lattice of well-formed function types. The static type of untrusted pointer variables from the attacker is the rightmost type $(U \xrightarrow{U \triangleleft U} U)_U$ (blue). Type confusion occurs when the attacker uses pointers as a supertype (red). Type confusing the output type is not harmful because of covariance. Note that the label lattice does not need to be a 2-point lattice, and the figure uses a 2-point lattice for clarity of presentation.

IFC Function Signatures

Function declarations are annotated with *function types* with the form $\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o$. Here, τ_i is the input type and τ_o is the output type. pc_{ex} is the access control label, which we refer to as the *external pc (program counter)*; pc_{top} is the bounding label, which we also refer to as the *top pc*. We believe that declassification of control flow is intrinsically unsafe in decentralized systems, so we only consider the integrity aspect of control flow labels in this work. Like other types, *function pointer types* $\tau = (\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o)_\ell$ include an extra label ℓ, which is the label of the function pointer itself.

We can understand the possible security-type confusions by exploring the subtyping lattice of function types in an IFC type system. Figure 3.4 shows the subtyping lattice of function signatures. The output label τ_o is *covariant*—a function that only outputs high integrity values can be used securely where a function that outputs low integrity values is expected. The labels τ_i, pc_{ex}, and pc_{top} are *contravariant*—a function that can be called in an untrusted context and

takes any input may also be called from a trusted context or with a trusted input.

In a decentralized system, an IFC-labeled function type is a security specification. However, certain security specifications are self-contradictory, so certain function types are considered ill-formed. Let p be the principal on which this function is defined. Then a well-formed function type is one for which $pc_{top} \sqsubseteq p \sqsubseteq pc_{ex} \sqsubseteq \mathsf{lbl}(\tau_i)$. First, because each function runs at control-flow integrity of p, p is no more trusted than pc_{top}. Second, the external pc should be no more trusted than p so that p may call its own function. Finally, the labels of function inputs should be no more trusted than the external pc pc_{ex}; if not, this function would take trusted input from an untrusted context.

Type Confusion of IFC Function Types

With this background, we are ready to analyze what kinds of type confusion may affect IFC function types. Standard function-call typing rules in prior IFC type systems [107] taint all labels except pc_{top} within the function pointer by the label of the pointer itself.[1] This rule tracks the attacker's influence over the function output through the function pointer. For example, an untrusted pointer to a function with type $(U \xrightarrow{T \triangleleft T} U)_U$ cannot be called because it requires using an untrusted pointer in a trusted control-flow context.

Because reentrancy security forbids control-flow endorsement within the body of trusted functions, calling a function with type $(U \xrightarrow{U \triangleleft T} U)_U$ from a trusted context is also insecure. As a result, a trusted function may use an untrusted function pointer only when every security label appearing in its type is untrusted (U).

Visually, in fig. 3.4, a type confusion is any type cast not permitted by sub-

[1] Most prior IFC systems prove strict noninterference and do not support control-flow endorsement. Thus, the external pc is equal to the body principal ($pc_{ex} = p$) for function types. By the well-formedness condition, the pointer label taints all labels except pc_{top}.

typing. As type confusion only happens to function pointers provided by the attacker, such casts only happen to untrusted function pointers. The only type of well-formed untrusted pointer used by trusted principals has the form $(U \xrightarrow{U \triangleleft U} U)_U$, so there are only three possible down-casts for each contravariant type constructor: the top pc, the external pc, and the input type. The runtime checks described previously cover all three cases.

3.1.4 The Simple Cases of Type Confusion

Nested Type Confusion

A more complex form of type confusion than those in fig. 3.4 is type confusion of function types whose input and output types are *themselves* function types. This *nested type confusion*, where the attacker lies about the input and output types of function pointers, may cause the deputy to indirectly type-confuse other function pointers. Fortunately, well-formedness of function types ensures that input and output types are untrusted function pointers, whose damage is contained within the untrusted domain.

Base Type Confusion

Up to this point, the discussion has focused on confusion of information flow types. While confusion of IFC types is the major challenge, confusion of *base* types can also lead to insecurity in some systems. For example, an adversary might supply an integer where a function pointer is expected, and trusted code might try to call the function, transferring control to an arbitrary memory address. Since such an integer is untrusted, the IFC mechanisms already protect against such confusion, under certain assumptions. Intuitively, when a value v_1 of base type τ_1 is used as if it were a value of base type τ_2, the language runtime must ensure

that the behavior resulting from using the value is equivalent to deterministically converting v_1 to *some* value of type τ_2.

3.2 Formalizing Type Confusion

The examples from §3.1 demonstrate that a dynamic integrity control-flow check and a reentrancy lock at remote calls suffice to enforce security against type confusion. In this section, we outline a system model for describing decentralized systems and work toward high-level formal definitions of CDA and type confusion. Our definition of type-confusion security is a simulation-based security property inspired by the real–ideal paradigm from the universal composability (UC) framework [16].

3.2.1 The Decentralized World Model

In our system model, a *program* P is a list of top-level function declarations by different principals p representing different trust domains. A principal may be controlled by an attacker $\mathcal{A}$ that executes arbitrary code. Principals are stateful, and each principal has a separate memory heap that maps references to values. Their collective state is a *program context* Σ. We assume (and later define) a deterministic trace semantics for program executions written as $(P, \Sigma, \mathcal{A}) \Downarrow t$. The *behavior* of a program–attacker pair is a function from a program context Σ to a trace t where $(P, \Sigma, \mathcal{A}) \Downarrow t$.

A *hyperproperty* is a set of behaviors [24]. A program satisfies a hyperproperty if its behavior is in the set that defines the hyperproperty. Hyperproperties generalize trace properties: they are relational properties over multiple traces of programs that can express complex security and correctness requirements. For

example, both noninterference [47] and nonmalleable information flow (NMIF) [18] are hyperproperties that relate two and four traces, respectively.

We conservatively model function pointers as an abstraction of various real-world systems, such as the public methods of EVM smart contracts. First, function pointers are publicly visible and callable. Second, we assume function pointers are *opaque* to the caller: that is, the caller cannot inspect the function body or type without transferring control to the callee. Third, the caller and the callee in remote calls know each other's identities. Fourth, static information flow control is enforced locally within each function. Finally, all functions publish their IFC type signatures, though function types provided by adversarial principals may be lies.

3.2.2 Type-Confusion Security

Prior static IFC type systems usually either assume well-typed attackers or restrict the interface between trusted code and untrusted code. Type confusion enters the picture when we reason about information flow but consider a more powerful ill-typed attacker. To formalize the idea of *type-confusion security*, we appeal to a simulation argument between well-typed and ill-typed attackers: a system is type-confusion secure when for each ill-typed attacker, there is a well-typed attacker who has the same behavior. This is a kind of real–ideal simulation—the real world has ill-typed attackers but includes runtime checks to ensure that they cannot cause damage; the ideal world has well-typed attackers but does not need runtime checks. Since the real world is just as secure as the ideal world, the programmer can think about security in terms of the ideal world with its well-behaved attackers.

Definition 3.2.1 (Type-Confusion Security)**.** A program P is secure against type

confusion when for every ill-typed attacker $\mathcal{A}^r$ running in the real world with runtime checks, there exists a well-typed attacker $\vdash \mathcal{A}^i$ that produces the same behavior in the ideal-world dynamic semantics with no runtime checks. With real-world execution and ideal-world execution represented by the notations $\Downarrow_r$ and $\Downarrow_i$, security can be formalized concisely:

$$\forall \mathcal{A}^r \centerdot \exists \vdash \mathcal{A}^i \centerdot \forall \Sigma \centerdot (P, \Sigma, \mathcal{A}^r) \Downarrow^r t \wedge (P, \Sigma, \mathcal{A}^i) \Downarrow^i t' \implies t \approx_{\bar{A}} t'$$

Type-confusion security can be understood as a compiler correctness property where the ideal world is the source language and the real world is the target language [5]. In this case, the compiler does not translate syntax. Instead, it performs static type checking and inserts runtime checks. The automatic runtime check inserted by the compiler is modeled by distinct operational semantics for the ideal world ($\Downarrow_i$) and the real world ($\Downarrow_r$).

This definition ensures *robust hyperproperty preservation* (RHP) [5], which means that all hyperproperties that hold in the well-typed setting are preserved even in the presence of ill-typed attackers.

Type confusion security depends on the semantics of both the well-typed and ill-typed languages. In order for the definition to be useful, the ideal world (well-typed world) should be easier to reason about than the real world (ill-typed world); the ideal world should be expressive enough to build interesting programs, and the real world should be realistic enough to model practical attacks.

In the following sections, we instantiate two IFC-typed core calculi. The ideal-world calculus has no runtime checks and only well-typed attackers; the real-world calculus has runtime checks that cause well-typed programs to abort when they are type-confused. We then instantiate type-confusion security for the calculi, and use it to prove preservation of all security hyperproperties between

$$
\begin{array}{ll}
\text{Principal} & p, q \in \mathbb{P} \subseteq \mathbb{L} \\
\text{Label} & \ell, pc \in (\mathbb{L}, \sqsubseteq) \\
\text{Bool} & b \in \{\mathsf{true}, \mathsf{false}\} \\
\text{Variable} & x \in \mathbb{X} \\
\text{Fun. Name} & f \in \mathbb{F} \\
\text{Ref Name} & h \in \mathbb{H} \\
\text{FnDecl} & F \in \mathbb{P} \times \mathbb{F} \to \mathsf{Signatures} \times \mathsf{Functions} \\
\text{HeapType} & R \in \mathbb{P} \times \mathbb{H} \to \mathsf{Types} \\
\text{Program} & P \in \mathsf{FnDecl} \times \mathsf{HeapType} \\
\text{Adv. Lbl.} & A \subseteq \mathbb{P} \\
\text{Adv. Fun.} & M(A) \in A \times \mathbb{F} \to \mathsf{Signatures} \times \mathsf{Functions} \\
\text{Attacker} & \mathcal{A} \in A \times M(A) \\
\text{Heap} & \Sigma \in \mathbb{P} \times \mathbb{H} \to \mathsf{Values}
\end{array}
$$

$$
\begin{array}{llll}
\text{Functions} & F & ::= & p.f : \tau \xrightarrow{pc_{ex} \lhd pc_{top}} \tau := \lambda x.e \\
\text{Ref. Decl.} & R & ::= & p.h : \tau \\
\text{Value} & v & ::= & () \mid b_\ell \mid p.f_\ell \mid x \\
\text{Expr.} & e & ::= & v \mid \mathsf{abort} \mid v \otimes v \mid v\ [pc] v \mid \{e\}_{p.f} \\
 & & \mid & v \downarrow \ell \mid \mathsf{write}_h(v) \mid \mathsf{read}_h \\
 & & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{if}\ v\ \mathsf{then}\ e\ \mathsf{else}\ e \\
\text{Signature} & \eta & ::= & \tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o \\
\text{Type} & \tau & ::= & 1 \mid \mathsf{bool}_\ell \mid (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o)_\ell \mid 0 \\
\text{Event} & w & ::= & \epsilon \mid (p.h := v) \mid \mathsf{call}^{pc}(p) \mid \mathsf{ret}(p) \mid \bot \\
\text{Trace} & t & ::= & \epsilon \mid t; w
\end{array}
$$

Figure 3.5: Syntax of λ_{tc}.

the ideal and the real world.

3.3 A Core Calculus for Type Confusion

In this section, we define a core calculus λ_{tc} as a formal model of a decentralized system that supports first-class function pointers and remote function calls. λ_{tc} uses both static and dynamic information flow control to enforce security.

3.3.1 Syntax

Figure 3.5 shows the syntax of λ_{tc}. The language models a decentralized system with multiple principals $p \in \mathbb{P}$, a subset of which are controlled by an attacker A. Here, a program $P = (F, R)$ is a pair of a function declaration table F and a heap type declaration table R. The function declaration table F maps each function name $p.f$ to its static type signature and function body. At run time, the attacker function table $M(A)$ replaces the bodies of functions defined on attacker-controlled principals. As the function bodies are statically type-checked against the published function signatures from F, the attacker cannot lie about function signatures. The program also contains a heap declaration table R, which maps heap names $p.h$ to their static type signatures. Σ is the runtime heap that maps heap names to values. Equivalently, the function and heap tables can be characterized by lists of function/heap declarations with distinct names.

Function bodies have the form $\lambda x.e$, where x is the input variable and e is the function body expression. An expression may be a value v, which is either a unit, a boolean, a function pointer, or a variable. Expressions in λ_{tc} follow A-Normal Form (ANF) [88]: each intermediate computation is bound to a variable and control flow occurs only at if-expressions, let-bindings, and remote function applications. Computation includes aborting (abort), binary operations between bools ($v \otimes v$), remote function applications ($v\ [pc]v$), returning ($\{e\}_p$), information downgrading ($v \downarrow \ell$), and reading from and writing to the heap (read_h, $\mathsf{write}_h(v)$). Each remote function application is tagged with the pc label used for the remote call, which in a real language would be inferred statically. Information-downgrading expressions provide controlled declassification and endorsement operations [58] that explicitly violate information flow policies for necessary expressiveness. Write expressions are the only expressions that cause

side effects to the heap.

Wrappers $\{e\}_{p.f}$ are not part of the surface syntax and are used to track the execution context of expressions. Intuitively, each time a remote function is called, the remote function pointer gets replaced by the body of the function wrapped in a wrapper parameterized by the function identifier.

Reads and writes access the heap locations h at the principal the expression runs on. In other words, λ_{tc} has no expressions for remote read and write; however, remote read and write could be implemented by reader and writer functions. This allows us to treat references and function pointers uniformly.

The type system uses standard information flow control (IFC) types [87] explained in §3.1.3, where function signatures are annotated with an input type, an output type, an external pc, and a top pc. The helper function $\mathsf{lbl}(\tau)$ takes a type and returns the label component of that type; unit values are always trusted: $\mathsf{lbl}(1) = \bot$.

The program emits trace events w during execution. An event may be a write event $(p.h := v)$ of a value v, a call event $\mathsf{call}^{pc}(p)$ to principal p, a return event $\mathsf{ret}(p)$ from principal p, or an abort event $\bot$.

3.3.2 Attackers

Formally, each attacker is a dependent pair $\mathcal{A} = (A, M(A))$, where $A \subseteq \mathbb{P}$ is the set of principals the attacker controls. We assume the existence of an attacker-controlled principal p_A who trusts all other attackers (p_A has integrity level $\top$). The attacker function table contains a special main function $p_A.\mathsf{main}$ that serves as the entry point of a program.

Let the program be $P = (F, R)$ and the attacker be $\mathcal{A} = (A, M(A))$. We define a helper function $\mathsf{body}(p.f)$ that dynamically returns the body of the function

after attacker replacement.

$$\mathsf{body}(p.f) = \begin{cases} \lambda x.e & p \in A \ \wedge \ M(A)(p.f) = (\eta, \lambda x.e) \\ \lambda x.e & p \notin A \ \wedge \ F(p.f) = (\eta, \lambda x.e) \end{cases}$$

3.3.3 Dynamic Semantics

The dynamic semantics of λ_{tc} is shown in fig. 3.6. The program starts at the attacker's main function and either terminates with a value or aborts. The runtime configuration keeps track of the heap Σ, the current expression e and a trace t of events emitted so far. We explain the slightly nonstandard cases of remote application, return, abort, read, and write in more detail.

In APP-REMOTE, the call expression steps as usual by the body of the function with the parameter replaced by the argument. To keep track of the execution context of a remote call, the body is enclosed in a wrapper annotated with the function identifier.

Intuitively, the wrapper is a lightweight runtime stack frame. It records which function body is currently executing without changing the surface language or introducing an explicit machine stack. As remote calls nest, wrappers nest with them, so the innermost wrapper always corresponds to the active callee. This is exactly the information needed by the dynamic semantics to determine which principal is currently running. The helper function $\mathsf{cur}()$ over evaluation context simply inspects the innermost wrapper to identify where the current execution is happening.

RETURN removes the wrapper, effectively returning to the caller's context. ABORT steps directly to a special value $\bot$, terminating the program immediately. READ reads from the heap Σ and WRITE writes to the heap and emits an event.

Evaluation Context $E ::= [\cdot] \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid \{E\}_p$

$\boxed{(P, \Sigma, \mathcal{A}) \Downarrow t}$ $\qquad$ $\boxed{\mathsf{cur}(E) = p}$

PROGRAM
$$\frac{p_A.\mathsf{main} = e \qquad \langle \{e\}_{p_A.\mathsf{main}}, \Sigma, () \rangle \to^* \langle v_\perp, \Sigma, t \rangle}{(P, \Sigma, \mathcal{A}) \Downarrow t}$$

$$\mathsf{cur}(E) = \mathsf{h}(E, p_A)$$

$$\mathsf{h}(\{E\}_{p.f}, q) = \mathsf{h}(E, p) \qquad \mathsf{h}([\cdot], p) = p$$

$$\mathsf{h}(\mathsf{let}\ x = E\ \mathsf{in}\ e, p) = \mathsf{h}(E, p)$$

$\boxed{\langle e, \Sigma, t \rangle \to \langle e_\perp, \Sigma, t \rangle}$ $\qquad$ $\boxed{\langle e, \Sigma, t \rangle \to_e \langle e, \Sigma, t \rangle}$

EVAL-EXPR
$$\frac{\langle e, \Sigma, t \rangle \to_e \langle e', \Sigma', t' \rangle}{\langle E[e], \Sigma, t \rangle \to \langle E[e'], \Sigma', t' \rangle}$$

ABORT
$$\frac{t' = (t; \perp)}{\langle E[\mathsf{abort}], \Sigma, t \rangle \to \langle \perp, \Sigma, t' \rangle}$$

WRITE
$$\frac{\mathsf{cur}(E) = p \qquad \Sigma' = \Sigma[p.h \mapsto v] \qquad t' = t; (p.h := v)}{\langle E[\mathsf{write}_h(v)], \Sigma, t \rangle \to \langle E[()], \Sigma', t' \rangle}$$

READ
$$\frac{\mathsf{cur}(E) = p \qquad \Sigma(p.h) = v}{\langle E[\mathsf{read}_h], \Sigma, t \rangle \to \langle E[v], \Sigma, t \rangle}$$

OP
$$\frac{}{\langle v_\ell \otimes v_{\ell'}, \Sigma, t \rangle \to_e \langle (v[\![\otimes]\!]v')_{\ell \sqcup \ell'}, \Sigma, t \rangle}$$

RETURN
$$\frac{t' = t; \mathsf{ret}(q)}{\langle \{v\}_{q.g}, \Sigma, t \rangle \to_e \langle v, \Sigma, t' \rangle}$$

IF-T
$$\frac{}{\langle \mathsf{if}\ \mathsf{true}_\ell\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, \Sigma, t \rangle \to_e \langle e_1, \Sigma, t \rangle}$$

IF-F
$$\frac{}{\langle \mathsf{if}\ \mathsf{false}_\ell\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, \Sigma, t \rangle \to_e \langle e_2, \Sigma, t \rangle}$$

DOWNGRADE
$$\frac{}{\langle b_{\ell'} \downarrow \ell, \Sigma, t \rangle \to_e \langle b_\ell, \Sigma, t \rangle}$$

LET
$$\frac{}{\langle \mathsf{let}\ x = v\ \mathsf{in}\ e, \Sigma, t \rangle \to_e \langle e\{v/x\}, \Sigma, t \rangle}$$

APP-REMOTE
$$\frac{\mathsf{body}(q.g) = \lambda x.e \qquad t' = t; \mathsf{call}^{pc}(q)}{\langle q.g_\ell\ [pc]v, \Sigma, t \rangle \to_e \langle \{e\{v/x\}\}_{q.g}, \Sigma, t' \rangle}$$

Figure 3.6: Dynamic semantics of λ_{tc}.

$\boxed{\vdash P}$ — $\boxed{P \vdash \mathcal{A}}$ — $\boxed{P \vdash \Sigma}$

T-SIGNATURE
$$\frac{F(p.f) = (\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o, \lambda x.e) \qquad x : \tau_i; p; (p, pc_{top}) \vdash e : \tau \qquad pc_{top} \sqsubseteq p \sqsubseteq pc_{ex} \sqsubseteq \mathsf{lbl}(\tau_i)}{\vdash (F, R)}$$

T-ADV
$$\frac{\mathcal{A} = (A, M(A)) \qquad \vdash (M(A), R)}{(F, R) \vdash \mathcal{A}}$$

T-STORE
$$\frac{R(p.h) = \tau \qquad \Sigma(p.h) = v \qquad \emptyset; pc; p; pc_{top}^{self} \vdash v : \tau'}{(F, R) \vdash \Sigma}$$

Figure 3.7: Static semantics of λ_{tc}: well-formed programs, attacker configurations, and stores.

3.3.4 Static Semantics

Figures 3.7 to 3.9 show the static semantics of λ_{tc}. As explained in §3.1, types are covariant in their labels; function types are contravariant in input type and pc types, and covariant in output type. The well-formedness condition $pc_{top} \sqsubseteq p \sqsubseteq pc_{ex} \sqsubseteq \mathsf{lbl}(\tau_i)$ of function signatures is enforced in T-SIGNATURE.

The typing judgment has the form $\Gamma; pc; p; pc_{top}^{self} \vdash e : \tau$, where Γ maps variables to IFC types, pc tracks the current control-flow context, p is the principal on which the expression e is being typed, pc_{top}^{self} is the top pc of the current function, e is the expression being typed, and τ is the type of the expression. The judgment is implicitly annotated with a program $P = (F, R)$. Rules for unit, booleans, operations, let-bindings, and subsumption are standard. T-FIELD queries the program for the type of a top-level function. T-DOWNGRADE changes the label of a value's type to another label. T-IF is mostly standard, except that the e_i may have different types; the if-statement is type-checked with respect to the join of the types of both branches, where the join of types ($\tau_1 \sqcup \tau_2$) is defined

$\boxed{\Gamma; pc; p; pc_{top}^{self} \vdash e : \tau}$

T-UNIT
$$\frac{}{\Gamma; pc; p; pc_{top}^{self} \vdash () : 1}$$

T-BOOL
$$\frac{}{\Gamma; pc; p; pc_{top}^{self} \vdash b_\ell : \mathsf{bool}_\ell}$$

T-FIELD
$$\frac{F(q.g) = (\eta, \lambda x.e)}{\Gamma; pc; p; pc_{top}^{self} \vdash q.g_\ell : \eta_\ell}$$

T-ABORT
$$\frac{}{\Gamma; pc; p; pc_{top}^{self} \vdash \mathsf{abort} : \tau}$$

T-VAR
$$\frac{}{\Gamma, x : \tau; pc; p; pc_{top}^{self} \vdash x : \tau}$$

T-OP
$$\frac{\Gamma; pc; p; pc_{top}^{self} \vdash v_i : \mathsf{bool}_{\ell_i} \qquad \ell = \ell_1 \sqcup \ell_2}{\Gamma; pc; p; pc_{top}^{self} \vdash v_1 \otimes v_2 : \mathsf{bool}_\ell}$$

T-DOWNGRADE
$$\frac{\Gamma; pc; p; pc_{top}^{self} \vdash v : \mathsf{bool}_{\ell'}}{\Gamma; pc; p; pc_{top}^{self} \vdash v \downarrow \ell : \mathsf{bool}_\ell}$$

T-LET
$$\frac{\Gamma; pc; p; pc_{top}^{self} \vdash e' : \tau' \qquad \Gamma, x : \tau'; pc; p; pc_{top}^{self} \vdash e : \tau}{\Gamma; pc; p; pc_{top}^{self} \vdash \mathsf{let}\ x = e'\ \mathsf{in}\ e : \tau}$$

T-IF
$$\frac{\Gamma; pc; p; pc_{top}^{self} \vdash v : \mathsf{bool}_\ell \qquad \Gamma; pc \sqcup \ell; p; pc_{top}^{self} \vdash e_i : \tau_i \qquad i \in \{1, 2\} \qquad \tau = \tau_1 \sqcup \tau_2}{\Gamma; pc; p; pc_{top}^{self} \vdash \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau}$$

T-WRAPPER
$$\frac{F(q.g) = (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o, \lambda x.e) \qquad x : \tau_i; q; q; pc_{top} \vdash e : \tau_o}{\Gamma; pc'; p; pc_{top}^{self} \vdash \{e\}_{q.g} : \tau_o}$$

T-SUB
$$\frac{\tau' \preccurlyeq \tau \qquad \Gamma; pc; p; pc_{top}^{self} \vdash e : \tau'}{\Gamma; pc; p; pc_{top}^{self} \vdash e : \tau}$$

T-READ
$$\frac{R(p.h) = \tau \qquad pc \sqsubseteq \mathsf{lbl}(\tau)}{\Gamma; pc; p; pc_{top}^{self} \vdash \mathsf{read}_h : \tau}$$

T-WRITE
$$\frac{R(p.h) = \tau \qquad \Gamma; pc; p; pc_{top}^{self} \vdash v : \tau \qquad pc \sqsubseteq \mathsf{lbl}(\tau)}{\Gamma; pc; p; pc_{top}^{self} \vdash \mathsf{write}_h(v) : 1}$$

T-APP
$$\frac{\begin{array}{c}\Gamma; pc; p; pc_{top}^{self} \vdash v_1 : (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o)_\ell \qquad \Gamma; pc; p; pc_{top}^{self} \vdash v_2 : \tau \\ \tau \preccurlyeq \tau_i \qquad pc \sqcup \ell \sqsubseteq pc_{ex} \qquad pc_{ex} \sqsubseteq pc_{top} \sqcup p \qquad pc_{top}^{self} \sqsubseteq pc_{top} \qquad \tau_o' = \tau_o \sqcup \ell\end{array}}{\Gamma; pc; p; pc_{top}^{self} \vdash v_1\ [pc_{ex}]v_2 : \tau_o'}$$

Figure 3.8: Static semantics of λ_{tc}: expression typing.

$\boxed{\tau \preccurlyeq \tau}$

$$\text{S-UNIT}\ \frac{}{\tau \preccurlyeq 1} \qquad \text{S-EMPTY}\ \frac{}{0 \preccurlyeq \tau} \qquad \text{S-BOOL}\ \frac{\ell \sqsubseteq \ell'}{b_\ell \preccurlyeq b_{\ell'}}$$

$$\text{S-FUNC}\ \frac{\ell \preccurlyeq \ell' \qquad \tau_o \preccurlyeq \tau_o' \qquad \tau_i' \preccurlyeq \tau_i \qquad pc_{ex}' \preccurlyeq pc_{ex} \qquad pc_{top}' \preccurlyeq pc_{top}}{(\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o)_\ell \preccurlyeq (\tau_i' \xrightarrow{pc_{ex}' \triangleleft pc_{top}'} \tau_o')_{\ell'}}$$

Figure 3.9: Static semantics of λ_{tc}: subtyping.

as the least upper bound of τ_1 and τ_2 with respect to the subtyping relation $\preccurlyeq$. T-READ and T-WRITE both type check with respect to the type to which the principal p and reference name h map within the type heap R. When interacting with the store, the current label of the type τ is no more trusted than the current pc, as untrusted individuals should not access or modify trusted information. Similar to T-SIGNATURE, T-WRAPPER requires the type of the expression in the wrapper to respect the function signature.

T-APP enforces the pc_{ex} and pc_{top} policies from §3.1. First, it ensures that both the calling pc and the label of the function pointer ℓ are no less trusted than the external pc of the function. This ensures that the external pc policy is respected and the attacker's influence over the function pointer is accounted for. The check $pc_{ex} \sqsubseteq pc_{top} \sqcup p$ requires that the external pc either flows to the top pc or flows to the caller p. In the first case, the function being called will never auto-endorse before it returns; in the second case, this function call is not an attenuated call, so the control flow integrity p does not fall below p in the first place. Finally, the call checks that pc_{top} of the function being called is bounded by the current top pc pc_{top}^{self} of the caller. This check ensures that the promise pc_{top}^{self} makes about the maximum effect of the current function is not violated indirectly through remote

calls. We write $\tau_o \sqcup \ell$ as a shorthand for updating the label of τ_o to $\mathsf{lbl}(\tau_o) \sqcup \ell$.

Although the typing rules are presented declaratively, they admit a decidable syntax-directed implementation, assuming that the underlying label order is decidable. This assumption is reasonable in practical, expressive IFC label models [6, 75, 84, 110].

3.3.5 Type Safety of λ_{tc}

For type safety, we require that all outputs from the heap Σ are well-typed (T-STORE).

Lemma 3.3.1 (Type Safety of λ_{tc})**.** *Progress:*

$$\begin{aligned} &\vdash P \wedge P \vdash \Sigma \wedge \Gamma; pc; p; pc_{top}^{self} \vdash e : \tau \wedge e \neq v \\ &\implies \exists e',\ \langle e, \Sigma, t\rangle \rightarrow \langle e', \Sigma', t'\rangle \end{aligned}$$

Preservation:

$$\begin{aligned} &\vdash P \wedge P \vdash \Sigma \wedge \Gamma; pc; p; pc_{top}^{self} \vdash e : \tau \wedge \langle e, \Sigma, t\rangle \rightarrow \langle e', \Sigma', t'\rangle \\ &\implies P \vdash \Sigma' \wedge \exists \tau',\ \Gamma; pc; p; pc_{top}^{self} \vdash e' : \tau' \end{aligned}$$

The rest of this subsection proves the type safety of λ_{tc}.

Lemma 3.3.2 (Substitution)**.**

$$\Gamma, x : \tau'; pc \vdash e : \tau \ \wedge\ \emptyset; pc \vdash v : \tau' \implies \Gamma; pc \vdash e\{v/x\} : \tau$$

Proof. By induction over typing derivation. □

Lemma 3.3.3 (PC Anti-Monotonicity)**.**

$$pc' \sqsubseteq pc \wedge \Gamma; pc; p; pc_{top}^{self} \vdash e : \tau \implies \Gamma; pc'; p; pc_{top}^{self} \vdash e : \tau$$

Proof. By induction over the typing derivation. □

Lemma 3.3.4 (Weakening)**.**

$$\Gamma_1; pc; p; pc_{top}^{self} \vdash e : \tau \implies \Gamma_1, \Gamma_2; pc; p; pc_{top}^{self} \vdash e : \tau$$

Proof. By induction over the typing derivation. □

Lemma 3.3.5 (Let Congruence)**.**

$$\langle e_1, \Sigma, t\rangle \to \langle e_1', \Sigma', t'\rangle \implies \langle \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, \Sigma, t\rangle \to \langle \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2, \Sigma', t'\rangle$$

They also hold for $\{e\}_{p.f}$*.*

Proof. By case analysis over the stepping derivation. □

Lemma 3.3.6 (Progress)**.** *The following holds for* λ_{tc}*:*

$$\vdash P \wedge P \vdash \Sigma \wedge \Gamma; pc; p; pc_{top}^{self} \vdash e : \tau \wedge e \neq v \implies \exists e', \Sigma', t' \,.\, \langle e, \Sigma, t\rangle \to \langle e', \Sigma', t'\rangle$$

They also hold for λ_{tc}^{r} *(replace the respective* $\to$ *and* $\to_e$ *by* $\to_r$ *and* $\to_{er}$*).*

Proof. We do induction over typing derivation of e.

- Case T-ABORT $e = \mathsf{abort}$. ABORT applies.
- Case T-OP $e = v_1 \otimes v_2$. By typing rule, e_1 and e_2 are bools. So OP applies.
- Case T-LET $e = \mathsf{let}\ x = e'\ \mathsf{in}\ e$. In the case when $\langle e', \Sigma, t\rangle \to \langle e'', \Sigma', t'\rangle$, the congruence lemma 3.3.5 applies. Otherwise, LET applies.
- Case T-IF $e = \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. By typing rule, v is a bool. So IF-TRUE or IF-FALSE applies.
- Case T-DOWNGRADE $e = b_{\ell'} \downarrow \ell$. DOWNGRADE applies.
- Case T-READ $e = \mathsf{read}_h$. READ applies.

- Case T-WRITE $e = \mathsf{write}_h(v)$. WRITE applies.
- Case T-APP $e = v_1\ [pc]v_2$. By typing rule, v_1 has function type. So APP-REMOTE applies.
- Case T-WRAPPER $e = \{e'\}_{p.f}$. In the case when $\langle e', \Sigma, t\rangle \to \langle e'', \Sigma', t'\rangle$, the congruence lemma 3.3.5 applies. Otherwise, RETURN and the induction hypothesis applies.
- Case T-SUB $e = e$. The induction hypothesis applies.

To show progress for λ^r_{tc}, we need to show that one of APP-REMOTE-OK, APP-REMOTE-ABORT and APP-REMOTE-CALLEE-ADV applies when APP-REMOTE applies in λ_{tc}, and one of RETURN-POP and RETURN-CALLEE-ADV applies when RETURN applies. This follows from the premises of the rules, which exhaust all cases. □

Lemma 3.3.7 (E-Preservation)**.** *The following holds for λ_{tc}:*

$$\vdash P \wedge P \vdash \Sigma \wedge \Gamma; pc; p; pc^{self}_{top} \vdash e : \tau \wedge \langle e, \Sigma, t\rangle \to_e \langle e', \Sigma', t'\rangle \implies \vdash \Sigma' \wedge \Gamma; pc; p; pc^{self}_{top} \vdash e' : \tau$$

They also hold for λ^r_{tc} (replace the respective $\to$ and $\to_e$ by $\to_r$ and $\to_{er}$).

Proof. We do induction over typing derivation of e. In all cases except T-WRITE, taking a step does not change the store, so $\vdash \Sigma'$ holds by $\vdash \Sigma$. Moreover, T-WRITE and T-READ are immediately discharged because $\mathsf{write}_h(v)$ and read_h are not in the stepping relation $\to_e$.

- Case T-OP $e = v_1 \otimes v_2$. Then it must be that $v_i = b_{l_i}$ by the interpretation function, and $\Gamma; pc \vdash b_{\ell'_i} : \mathsf{bool}_{lbl_i}$, which implies that $\mathsf{bool}_{\ell'_1 \sqcup \ell'_2} \preccurlyeq \mathsf{bool}_{\ell_1 \sqcup \ell_2}$. Preservation holds by T-SUB.
- Case T-LET $e = \mathsf{let}\ x = e'\ \mathsf{in}\ e$. LET applies and $e' = e\{v/x\}$. Preservation holds by theorem 3.3.2.

- Case T-IF $e = \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. By typing derivation, $\tau = \tau_1 \sqcup \tau_2$ where e_i have type τ_i. IF-TRUE or IF-FALSE apply and $e' = e_i$ for some i. Let $\Gamma; pc \vdash b : b_\ell$, then by typing derivation, we know $\Gamma; pc \sqcup \ell \vdash e_i : \tau_i$. By theorem 3.3.3, $\Gamma; pc \vdash e_i : \tau_i$ holds. Rule IF-T or IF-F apply.
- Case T-DOWNGRADE $e = b_{\ell'} \downarrow \ell$. DOWNGRADE applies and the bool value remains well-typed.
- Case T-READ $e = \mathsf{read}_h$. Vacuously true.
- Case T-WRITE $e = \mathsf{write}_h(v)$. Vacuously true.
- Case T-APP $e = f\ [pc]v$. By the typing derivation, $e = p.f\ [pc]v, \Gamma; pc; p; pc_{top}^{self} \vdash f : (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o)_\ell$, and $\Gamma; pc; p; pc_{top}^{self} \vdash v : \tau$. By the stepping derivation, $\mathsf{body}(p.f) = e_{p.f}$, and $e' = \{e_{p.f}\{v/x\}\}_{q.g}$ To prove the goal, we do induction on the first typing judgment of the following more general lemma:

$$\begin{aligned} &\vdash P \wedge P \vdash \Sigma \wedge \Gamma; pc; p; pc_{top}^{self} \vdash p.f : (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o)_\ell \\ &\wedge \Gamma; pc; p; pc_{top}^{self} \vdash v : \tau \wedge \tau \preccurlyeq \tau_i \wedge \mathsf{body}(p.f) = e_{p.f} \\ &\implies \Gamma; pc; p; pc_{top}^{self} \vdash \{e_{p.f}\{v/x\}\}_{q.g} : \tau_o \end{aligned}$$

  The proof only involves two cases

  - Case T-FIELD. Case analysis on $q \in A$. By T-SIGNATURE, in either case both the normal function and attacker's function both type-checked with the same type (i.e. $x : \tau_i; p; (p, pc_{top}) \vdash e_{p.f} : \tau_o$). Preservation follows by T-WRAPPER and theorem 3.3.2, theorem 3.3.4, and the fact that pc is irrelevant in type-checking values.
  - Case T-SUB. By typing derivation, $\Gamma; pc; p; pc_{top}^{self} \vdash p.f : \tau'$ and $\tau' \preccurlyeq (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o)_\ell$, which implies $\tau' = (\tau_i' \xrightarrow{pc'_{ex} \lhd pc'_{top}} \tau_o')_{\ell'}$, and $\tau_o' \preccurlyeq type_o$. The proof follows by induction hypothesis

The proof of this case follows as an instance of the aforementioned lemma, with $\tau_o \preccurlyeq \tau_o$.

- Case T-WRAPPER $e = \{v\}_{p.f}$. RETURN applies and $e = v$. Preservation holds by typing derivation and the fact that pc is irrelevant in type-checking values.

To show preservation of $\rightarrow_{er}$ for λ^r_{tc}, we need to show preservation for APP-REMOTE-CALLEE-ADV, RETURN-POP, and RETURN-CALLEE-ADV, which runs a similar argument as APP-REMOTE and RETURN.

□

Lemma 3.3.8 (Preservation-Eval)**.** *The following holds for λ_{tc}:*

$$\Gamma; pc; p; pc^{self}_{top} \vdash E\,[e] : \tau \wedge (\forall\ \Gamma'\ pc'\ e'\ \tau', \Gamma'; pc'; p; pc^{self}_{top} \vdash e : \tau' \implies \Gamma'; pc'; p; pc^{self}_{top} \vdash e' : \tau')$$
$$\implies \Gamma; pc; p; pc^{self}_{top} \vdash E\,[e'] : \tau$$

Proof. Induction over the evaluation context E.

- Case $[\cdot]$. The proof follows by the second assumption.
- Case let $x = E'$ in e. By the typing derivation, we get $\Gamma; pc; p; pc^{self}_{top} \vdash E'[e] : \tau'$, and $\Gamma, x : \tau'; pc; p; pc^{self}_{top} \vdash e : \tau''$, and $\tau'' \preccurlyeq \tau'$. Preservation-Eval holds by T-SUB, T-LET, and the induction hypothesis.
- Case $\{E'\}_p$. By the typing derivation, we get $F(q.g) = (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o, \lambda x.e')$, $\tau'_o \preccurlyeq \tau_o$, and $x : \tau_i; q; q; pc_{top} \vdash e' : \tau'_o$. Preservation-Eval holds by T-SUB T-WRAPPER, and the induction hypothesis.

□

Lemma 3.3.9 (Preservation)**.** *The following holds for λ_{tc}:*

$$\vdash P \wedge P \vdash \Sigma \wedge \Gamma; pc; p; pc^{self}_{top} \vdash e : \tau \wedge \langle e, \Sigma, t\rangle \rightarrow \langle e', \Sigma', t'\rangle \implies P \vdash \Sigma' \wedge \exists \tau' \,.\, \Gamma; pc; p; pc^{self}_{top} \vdash e' :$$

They also hold for λ^r_{tc} (replace the respective $\rightarrow$ and $\rightarrow_e$ by $\rightarrow_r$ and $\rightarrow_{er}$).

Proof. We do induction over the stepping derivation. In all cases except T-WRITE, taking a step does not change the store, so $\vdash \Sigma'$ holds by $\vdash \Sigma$.

- Case EVAL-EXPR. Preservation follows by theorem 3.3.8 and theorem 3.3.7.
- Case ABORT. Immediately holds as abort can always type check.
- Case READ. By theorem 3.3.8, we want to show that $\Gamma'; pc'; p; pc_{top}^{self} \vdash \mathsf{write}_h(v) : \tau' \implies \Gamma'; pc'; p; pc_{top}^{self} \vdash () : \tau'$, where $\Sigma(p.h) = v$, and $R(p.h) = \tau'$. By T-STORE, read_h is type-checked by the corresponding type in R, which proves the claim.
- Case WRITE. By theorem 3.3.8, we want to show that $\Gamma'; pc'; p; pc_{top}^{self} \vdash \mathsf{write}_h(v) : 1 \implies \Gamma'; pc'; p; pc_{top}^{self} \vdash () : 1$. The claim holds due to T-WRITE and T-UNIT.

To show preservation for λ_{tc}^r, we need to show preservation for all APP-REMOTE-OK and APP-REMOTE-ABORT. They run the same argument as APP-REMOTE above. □

3.3.6 The Real World

We define the calculus λ_{tc}^r of the real world, where the honest principals use dynamic IFC checks.

Attackers from λ_{tc}^r still respect the non-security types—they do not use bools as function pointers and vice versa. Typing rules for attackers in λ_{tc}^r are identical to those of λ_{tc}, except that all IFC subtyping judgments are removed. Figure 3.11 shows the static semantics of these real-world attackers.

Figure 3.10 defines the runtime checks in the operational semantics $\rightarrow_r$ and $\rightarrow_{er}$ of λ_{tc}^r, which are identical to $\rightarrow$ and $\rightarrow_e$ from λ_{tc}, respectively, except that

$\langle e, \Sigma, \mathsf{locks}, t\rangle \to_r \langle e, \Sigma, \mathsf{locks}, t\rangle$ $\qquad$ $\langle e, \Sigma, \mathsf{locks}, t\rangle \to_{er} \langle e, \Sigma, \mathsf{locks}, t\rangle$

APP-REMOTE-OK

$$\frac{q \notin A \qquad \mathsf{expc}(E\,[q.g_\ell\,[pc]v]) = (pc_{ex}^{caller}, pc_{ex}^{callee}) \qquad \mathsf{cur}(E) = p \qquad p \sqcup pc_{ex}^{caller} \sqsubseteq pc_{ex}^{callee} \qquad \forall \ell \in \mathsf{locks},\ p \sqcup pc_{ex}^{caller} \sqsubseteq q \sqcup \ell \qquad \mathsf{body}(q.g) = \lambda x.e \qquad \mathsf{locks}' = \mathsf{locks}; q \qquad t' = t; \mathsf{call}^{pc}(q)}{\langle E\,[q.g_\ell\,[pc]v]\,, \Sigma, \mathsf{locks}, t\rangle \to_r \langle E\,[\{e\{v/x\}\}_q]\,, \Sigma, \mathsf{locks}', t'\rangle}$$

APP-REMOTE-ABORT

$$\frac{q \notin A \qquad \mathsf{expc}(E\,[q.g_\ell\,[pc]v]) = (pc_{ex}^{caller}, pc_{ex}^{callee}) \qquad \mathsf{cur}(E) = p \qquad (p \sqcup pc_{ex}^{caller} \not\sqsubseteq pc_{ex}^{callee} \text{ or } \exists \ell \in \mathsf{locks}\,.\, p \sqcup pc_{ex}^{caller} \not\sqsubseteq q \sqcup \ell)}{\langle E\,[q.g_\ell\,[pc]v]\,, \Sigma, \mathsf{locks}, t\rangle \to_r \langle \mathsf{abort}, \Sigma, \mathsf{locks}, t\rangle}$$

RETURN-POP

$$\frac{q \notin A \qquad \mathsf{locks} = \mathsf{locks}'; q \qquad t' = t; \mathsf{ret}(q)}{\langle \{v\}_{q.g}, \Sigma, \mathsf{locks}, t\rangle \to_{er} \langle v, \Sigma, \mathsf{locks}', t'\rangle}$$

APP-REMOTE-CALLEE-ADV

$$\frac{q \in A \qquad \mathsf{body}(q.g) = \lambda x.e \qquad t' = t; \mathsf{call}^{pc}(q)}{\langle E\left[q.g_\ell\,[pc_{ex}^{caller}]v\right], \Sigma, \mathsf{locks}, t\rangle \to_r \langle E\,[\{e\{v/x\}\}_{q.g}]\,, \Sigma, \mathsf{locks}, t'\rangle}$$

RETURN-CALLEE-ADV

$$\frac{q \in A \qquad t' = t; \mathsf{ret}(q)}{\langle \{v\}_{q.g}, \Sigma, \mathsf{locks}, t\rangle \to_{er} \langle v, \Sigma, \mathsf{locks}, t'\rangle}$$

Figure 3.10: Runtime checks in λ_{tc}^{r}. These rules replace APP-REMOTE and RETURN from λ_{tc}.

$$\text{Base Type} \quad \tau^r \quad ::= \quad 1 \mid 0 \mid \mathsf{bool} \mid \tau^r \to \tau^r$$

$$\mathcal{B}[\![1]\!] = 1 \qquad \mathcal{B}[\![0]\!] = 0 \qquad \mathcal{B}[\![\mathsf{bool}_\ell]\!] = \mathsf{bool} \qquad \mathcal{B}[\![\eta_\ell]\!] = \mathcal{B}[\![\eta]\!]$$

$$\mathcal{B}[\![\eta_\ell \xrightarrow{pc_{ex} \lhd pc_{top}} \eta'_{\ell'}]\!] = \mathcal{B}[\![\eta]\!] \to \mathcal{B}[\![\eta']\!]$$

$\boxed{\vdash P}$

A-SIGNATURE
$$\frac{p \vdash (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o)_p \qquad x : \mathcal{B}[\![\tau_i]\!]; p \vdash^r e : \mathcal{B}[\![\tau_o]\!]}{\vdash^r p.f : \tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o := \lambda x.e}$$

$\boxed{\Gamma^r; p \vdash^r e : \tau^r}$

A-UNIT
$$\frac{}{\Gamma^r; p \vdash^r () : 1}$$

A-BOOL
$$\frac{}{\Gamma^r; p \vdash^r b_\ell : \mathsf{bool}}$$

A-FIELD
$$\frac{q.g : \eta := e \in P}{\Gamma^r; p \vdash^r q.f_\ell : \mathcal{B}[\![\eta]\!]}$$

A-VAR
$$\frac{}{\Gamma^r, x : \tau^r; p \vdash^r x : \tau^r}$$

A-ABORT
$$\frac{}{\Gamma^r; p \vdash^r \mathsf{abort} : 1}$$

A-OP
$$\frac{\Gamma^r; p \vdash^r v_i : \mathsf{bool}}{\Gamma^r; p \vdash^r v_1 \otimes v_2 : \mathsf{bool}}$$

A-LET
$$\frac{\Gamma^r; p \vdash^r e_1 : \tau_1^r \qquad \Gamma^r, x : \tau_1^r; p \vdash^r e_2 : \tau_2^r}{\Gamma^r; p \vdash^r \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2^r}$$

A-IF
$$\frac{\Gamma^r; p \vdash^r v : \mathsf{bool} \qquad \Gamma^r; p \vdash^r e_i : \tau_i^r}{\Gamma^r; p \vdash^r \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau_1^r \sqcup \tau_2^r}$$

A-READ
$$\frac{}{\Gamma^r; p \vdash^r \mathsf{read}_h : \mathcal{B}[\![\tau]\!]}$$

A-WRITE
$$\frac{\Gamma^r; p \vdash^r v : \tau^r \qquad \mathcal{B}[\![\tau]\!] \preccurlyeq \tau^r}{\Gamma^r; p \vdash^r \mathsf{write}_h(v) : 1}$$

A-DOWNGRADE
$$\frac{}{\Gamma^r; p \vdash^r v \downarrow \ell : \mathsf{bool}}$$

A-APP
$$\frac{\Gamma^r; p \vdash^r v_1 : \tau_i^r \to \tau_o^r \qquad \Gamma^r; p \vdash^r v_2 : \tau_i^r}{\Gamma^r; p \vdash^r v_1\ [pc] v_2 : \tau_o^r}$$

A-WRAPPER
$$\frac{\Gamma^r; p \vdash^r e : \tau^r}{\Gamma^r; p \vdash^r \{e\}_{q.g} : \tau^r}$$

Figure 3.11: Static semantics of λ^r_{tc}.

APP-REMOTE-OK, APP-REMOTE-ABORT, and APP-REMOTE-CALLEE-ADV replace APP-REMOTE from λ_{tc}; RETURN-POP and RETURN-CALLEE-ADV replace

RETURN. The honest principals also maintain a dynamic lock list to prevent reentrancy attacks [19]. Each honest principal puts itself into the lock list when it is called and removes itself from the lock list when it returns.

All dynamic checks occur on the callee's side. When the callee is controlled by the attacker, no dynamic check occurs, and the call always succeeds (APP-REMOTE-CALLEE-ADV). The lock list remains intact because the attacker ignores all runtime mechanisms.

Otherwise, the callee is honest and performs two checks after receiving pc_{ex}^{caller} from the caller: 1) whether the caller and the callee agree on the external pc of the function being called; 2) whether this call violates an existing reentrancy lock. This runtime check uses the auxiliary function expc to model the caller's and callee's knowledge of the call. $\mathsf{expc}(E\,[q.g_\ell\,[pc]v]) = (pc_{ex}^{caller}, pc_{ex}^{callee})$ is given by:

$$pc_{ex}^{caller} = \begin{cases} \bot & \mathsf{cur}(E) = p \text{ and } p \in A \\ pc & \mathsf{cur}(E) = p \text{ and } p \notin A \end{cases}$$

$$pc_{ex}^{callee} = pc_{ex} \text{ where } P(q.g) = (\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o, \lambda x.e)$$

We conservatively assume that attackers always try to bypass the dynamic check when calling honest principals. Any attacker who wishes such a call to fail may be simulated by another attacker who simply aborts before the call. So when the caller is controlled by the attacker, it always sends $pc_{ex}^{caller} = \bot$ to the callee so that the call passes the first external pc check. Otherwise, the caller is trusted and sends the correct calling pc pc as pc_{ex}^{caller}. In cases where the callee is honest, the callee uses pc_{ex} from its own function signature as pc_{ex}^{callee}.

The check $p \sqcup pc_{ex}^{caller} \sqsubseteq pc_{ex}^{callee}$ ensures that the caller uses sufficient integrity to call the callee function. Mathematically, this is two checks in one: $p \sqsubseteq pc_{ex}^{callee}$ and $pc_{ex}^{caller} \sqsubseteq pc_{ex}^{callee}$. When the caller is honest and does not lie, this check

$$
w_{|\bar{A}} = \begin{cases} (p.h := ()) & w = (p.h := q.g) \\ \epsilon & w = (p.h := v),\ \mathsf{lbl}(R(p.h)) \in A \\ \epsilon & w = \mathsf{call}^{pc}(q),\ q \in A \\ \epsilon & w = \mathsf{ret}(q),\ q \in A \\ w & \text{otherwise} \end{cases}
$$

$$
t_{|\bar{A}} = \begin{cases} t' & t = t'; w,\ w_{|\bar{A}} = \epsilon \\ t'; w_{|\bar{A}} & t = t'; w,\ w_{|\bar{A}} \neq \epsilon \\ \epsilon & t = \epsilon \end{cases}
$$

$$
t \approx_{\bar{A}} t' \iff t_{|\bar{A}} = t'_{|\bar{A}}
$$

Figure 3.12: Trace Erasure and Equivalence.

ensures that the calling pc is no less trusted than the external pc of the callee: $p \sqsubseteq pc \sqsubseteq pc_{ex}$. A confused caller would fail this check when it uses untrusted pc to call a high-external-pc function. When the caller is controlled by the attacker, $pc_{ex}^{caller} = \bot \sqsubseteq pc_{ex}^{callee}$ always passes. In such a case, we rely on $p \sqsubseteq pc_{ex}^{callee}$, which upper-bounds the lie about pc_{ex}^{caller} by the attacker's own integrity.

The reentrancy lock check is performed entirely on the callee's side. When level ℓ is locked, the called function either does not endorse control flow ($pc_{ex} \sqsubseteq q$), or the control flow is no less trusted than the lock level to begin with ($pc \sqsubseteq \ell$).

3.4 Type-Confusion Security

Informally, we require that every *observable behavior* with an ill-typed attacker is also possible with a well-typed attacker. Observable behaviors include writes to trusted heap references, calls to trusted principals, returns from trusted principals, and aborts. Figure 3.12 defines trace erasure $t_{|\bar{A}}$, which erases all untrusted events from a trace t. Trace equivalence $\approx_{\bar{A}}$ requires that two traces are identical after erasure. Finally, function values are considered indistinguishable as they

are opaque.

Theorem 3.4.1 (Type-Confusion Security)**.** *Type confusion security holds between λ_{tc} and λ^r_{tc} for all well-typed programs.*

$$\vdash P \implies \forall \mathcal{A}^r \,.\, \exists \mathcal{A}^i \,.\, \forall \Sigma. \vdash^r \mathcal{A}^r \wedge \vdash \mathcal{A}^i \wedge \vdash \Sigma \wedge$$
$$(P, \Sigma, \mathcal{A}^r) \Downarrow^r t \wedge (P, \Sigma, \mathcal{A}^i) \Downarrow^i t' \implies t \approx_{\bar{A}} t'$$

Proof. We sketch the core insights; a more detailed proof appears in §3.5.

The proof has three parts: constructing a simulator attacker $\mathcal{A}^i$ from the real attacker $\mathcal{A}^r$, showing that the simulator is well-typed, and proving that the two attackers have the same observable behavior.

Construction. Figure 3.13 defines the translation $\mathcal{T}[\![\cdot]\!]$, which maps $\mathcal{A}^r = (A, M^r(A))$ to $\mathcal{A}^i = (A, M^i(A))$. The key idea is to replace each attacker-visible function by translated variants that hard-wire the top execution label to one of the two possible cases, $\ell = \bot$ and $\ell = \top$. Fresh names for these variants are generated by $\mathcal{N}[\![\cdot]\!]$, their signatures are assigned by $\mathcal{U}[\![\cdot]\!]$, and the auxiliary map $\mathsf{C}^\ell(\cdot)$ switches a value to the corresponding variant. Intuitively, the simulator normalizes attacker behavior by making the control-flow choice explicit in the code it runs. Original program functions are left unchanged by S-ORIGINAL; only the attacker-facing view of those functions is replaced.

Well-typedness. The central invariant is that every freshly generated function belongs to the attacker principal p_A, whose integrity is the least trusted label $\top$. This matters because all translated code therefore executes at the fixed top level $\top$, so it cannot accidentally perform further control-flow endorsement or violate the reentrancy discipline. The two translated variants differ only in how they simulate the surrounding calling context, while both remain well-typed under the attacker types assigned by $\mathcal{U}[\![\cdot]\!]$.

$\boxed{\mathcal{T}[\![M(A), F]\!](p.f) = (\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o, \lambda x.e)}$

S-ORIGINAL

$$\frac{M(A)(p.f) = (\eta, \lambda x.e)}{\mathcal{T}[\![M(A), F]\!](p.f) = (\eta, \lambda x.(\mathcal{N}^\top[\![p.f]\!]_\top)\ [\top](\mathsf{C}^\top(x) \downarrow \mathcal{U}^\top[\![\tau_i]\!]))}$$

S-HIGH/S-AUTOENDORSE-ABORT

$$\frac{F(p.f) = (\eta, \lambda x.e) \qquad (pc_{ex} \notin A \qquad \text{or} \qquad pc_{ex} \in A \qquad p \notin A \qquad \ell = \top)}{\mathcal{T}[\![M(A), F]\!](\mathcal{N}^\ell[\![p.f]\!]) = (\mathcal{U}^\ell[\![\eta]\!], \lambda x.\mathsf{abort})}$$

S-AUTOENDORSE-SUCCESS

$$\frac{F(p.f) = (\eta, \lambda x.e) \qquad pc_{ex} \in A \qquad p \notin A \qquad \ell = \bot}{\mathcal{T}[\![M(A), F]\!](\mathcal{N}^\ell[\![p.f]\!]) = (\mathcal{U}^\ell[\![\eta]\!], \lambda x.\mathsf{C}^\ell(p.f_\top\ ((\mathsf{C}^\top(x)) \downarrow \tau_i)))}$$

S-LOW

$$\frac{M(A)(p.f) = (\eta, \lambda x.e) \qquad p \in A}{\mathcal{T}[\![M(A), F]\!](\mathcal{N}^\ell[\![p.f]\!]) = (\mathcal{U}^\ell[\![\eta]\!], \lambda x.\mathsf{C}^\ell(\mathcal{S}^\ell[\![e]\!]))}$$

$\boxed{\mathcal{U}^\ell[\![\tau]\!] = \tau}$

$$\mathcal{U}^\ell[\![\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o]\!] = \mathcal{U}^\ell[\![\tau_i]\!]_\top \xrightarrow{\top \triangleleft \ell} \mathcal{U}^\ell[\![\tau_o]\!]_\top \qquad \mathcal{U}^\ell[\![\mathsf{bool}_{\ell'}]\!] = \mathsf{bool}_\top \qquad \mathcal{U}^\ell[\![1]\!] = 1$$

$\boxed{\mathsf{C}^\ell(v) = v}$

$$\frac{v = x \text{ or } v = ()}{\mathsf{C}^\ell(v) = v} \qquad \mathsf{C}^\ell(b_{\ell'}) = b_\top \quad \mathsf{C}^\ell(p.f_{\ell'}) = \mathcal{N}^\ell[\![p.f]\!]_\top \quad \mathsf{C}^\ell(e) = \mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{C}^\ell(x)$$

$\boxed{\mathcal{S}^\ell[\![v]\!] = v}$

$$\mathcal{S}^\ell[\![v]\!] = \mathsf{C}^\top(v) \qquad \mathcal{S}^\ell[\![\mathsf{abort}]\!] = \mathsf{abort} \qquad \mathcal{S}^\ell[\![v \downarrow \ell']\!] = \mathcal{S}^\ell[\![v]\!]$$

$$\mathcal{S}^\ell[\![v_1 \otimes v_2]\!] = \mathcal{S}^\ell[\![v_1]\!] \otimes \mathcal{S}^\ell[\![v_2]\!] \qquad \mathcal{S}^\ell[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] = \mathsf{let}\ x = \mathcal{S}^\ell[\![e_1]\!]\ \mathsf{in}\ \mathcal{S}^\ell[\![e_2]\!]$$

$$\mathcal{S}^\ell[\![\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!] = \mathsf{if}\ \mathcal{S}^\ell[\![v]\!]\ \mathsf{then}\ \mathcal{S}^\ell[\![e_1]\!]\ \mathsf{else}\ \mathcal{S}^\ell[\![e_2]\!] \qquad \frac{\mathcal{H}[\![p.h]\!] = p_A.h'}{\mathcal{S}^\ell[\![\mathsf{read}_h]\!] = \mathsf{read}_{h'}}$$

$$\frac{\mathcal{H}[\![p.h]\!] = p_A.h'}{\mathcal{S}^\ell[\![\mathsf{write}_h(v)]\!] = \mathsf{write}_{h'}(\mathcal{S}^\ell[\![v]\!])} \qquad \mathcal{S}^\ell[\![v_1\ v_2]\!] = \mathsf{C}^\ell(v_1)\ [\top]\mathsf{C}^\ell(v_2) \quad \mathcal{S}^\ell[\![\{e\}_{q.g}]\!] = \{\mathcal{S}^\ell[\![e]\!]\}_{q.g}$$

Figure 3.13: Simulator Construction

Behavioral equivalence. The appendix proves a stronger simulation invariant and then a one-step unwinding theorem: every real-world step can be matched by zero or more ideal-world steps of the simulator while preserving the corresponding heap, expression, and public trace. The main case split follows the translation rules in fig. 3.13. S-HIGH and S-AUTOENDORSE-ABORT cover the cases where the real execution would fail a dynamic check, so the simulator substitutes an aborting behavior. S-LOW and S-AUTOENDORSE-SUCCESS cover the non-confusing cases, where the simulator forwards execution to the appropriately translated function body. Finally, S-ORIGINAL reuses code that is already compatible with the ideal world. Chaining the one-step simulation over an entire execution yields $t \approx_{\bar{A}} t'$, which proves type-confusion security.

□

3.4.1 Examples Revisited

For better intuition of the construction of the simulator, we revisit the examples from §3.1. Let the lattice be a two-point lattice $\{U = \top, T = \bot\}$ where $T \sqsubseteq U$. The attacker adv has label U and the compiler has label T. In the syntax of λ_{tc}, the compiler's type is $((\mathsf{bool}_U \xrightarrow{U \triangleleft U} 1)_U) \xrightarrow{U \triangleleft T} 1$ (we ignore the code argument for simplicity).

The CDA Attack

The billwriter's type is $\mathsf{bool}_U \xrightarrow{T \triangleleft T} 1$. The attacker is ill-typed because it uses a high-external-pc function pointer as an argument to a function that takes low-external-pc function pointer only. The ill-typed attacker and its simulator are

given as follows:

$$\mathsf{adv.main} := \mathsf{Compiler.compile}_U\ [U]\mathsf{billwriter}_U$$

$$\mathcal{S}^T[\![\mathsf{adv.main}]\!] := \mathsf{C^T}(\mathsf{Compiler.compile}_U)\ [U]\mathsf{C^T}(\mathsf{billwriter}_U)$$

The simulator translates the attacker's main function by applying $\mathcal{S}^T[\![\cdot]\!]$ to its body where T label means this function is allowed to auto-endorse. Therefore, the translation replaces both the function pointer and the argument by their T-versions. As shown in fig. 3.13, the body of $\mathsf{C^T}(\mathsf{Compiler.compile}_U)$ is:

$$\lambda x.\mathsf{Compiler.compile}_U\ [U]\mathsf{C^U}(x)$$

It simply calls billwriter with the U-version of the argument, which ensures that the argument never auto-endorses. Finally, by fig. 3.13, the body of $\mathsf{C^U}(\mathsf{billwriter})$ is `abort` because it is an attacker-controlled function pointer with trusted external pc. This function aborts when being used, simulating the runtime-check failure in the real world.

The Reentrancy Attack

The type of the function adv.reenter is $\mathsf{bool}_U \xrightarrow{U \lhd T} 1$. The attacker is ill-typed because it uses a high-top-pc function pointer as an argument to a function that takes low-top-pc function pointer only. The ill-typed attacker, its simulator and the T-version of the compiler function pointer are defined as follows:

$$\mathsf{adv.main} := \mathsf{Compiler.compile}_U\ [U]\mathsf{adv.reenter}_U$$

$$\mathcal{S}^T[\![\mathsf{adv.main}]\!] := \mathsf{C^T}(\mathsf{Compiler.compile}_U)\ [U]\mathsf{C^T}(\mathsf{adv.reenter}_U)$$

$$\mathsf{C^T}(\mathsf{Compiler.compile}_U) := \lambda x.\mathsf{Compiler.compile}_U\ [U]\mathsf{C^U}(x)$$

By fig. 3.13, the body of $\mathsf{C^U}(\mathsf{adv.reenter})$ is identical to that of adv.reenter except that all auto-endorsing calls are replaced by aborting functions. The behav-

ior of $\mathsf{C}^{\mathsf{U}}(\mathsf{adv.reenter})$ simulates that of $\mathsf{adv.reenter}$ correctly the cases where the reentrancy lock is active.

3.4.2 Base Type Confusion

In this section, we describe λ^b_{tc} to model base type confusion attacks where attackers do not follow any typing discipline. λ^b_{tc} has the same syntax as λ_{tc} and accepts all terms. To ensure that progress holds for this language, we need to define dynamic semantics for ill-typed operations. For exmple, the language should take a step when the guard of an if statement is a function pointer.

We assume the existence of a low-level abstract semantic domain $\mathbb{C}$, and all values are interpretations of codes from this domain. Let the interpretation be a surjection $[\![c]\!]_\tau$, which maps each code $c \in \mathbb{C}$ into a value of type τ. For example, in C, the interpretation function can be instantiated by setting the domain to be the bytes and $[\![c]\!]_{\mathsf{bool}} = \mathsf{false}$ should hold only when c is 0, and true otherwise.

Instead of defining semantics over values, we define them over their codes. When a $(v : \tau^r)$ is used in elimination forms (like IF-T for bools and APP-REMOTE for functions), the value is reinterpreted into the expected type. Formally, in each execution, we use the function $[\![v]\!]^{-1}$ to define the low-level code of v. To ensure that well-typed programs of λ^b_{tc} behave the same as those of λ^r_{tc}, $[\![[\![v]\!]^{-1}]\!]_{\tau^r} = v$ should hold for well-typed values. Note that the semantics of λ^b_{tc} are nondeterministic for ill-typed operations, because multiple codes may encode the same value.

Type confusion security holds between λ^r_{tc} and λ^b_{tc} when fixing $[\![v]\!]^{-1}\tau^r$, the source of nondeterminism.

Lemma 3.4.2 (Base Type-Confusion Security)**.** *Type confusion security holds between*

λ^r_{tc} and λ^b_{tc} for all well-typed P.

$$\vdash P \implies \forall \mathcal{A}^b \,.\, \exists \mathcal{A}^r \,.\, \forall \Sigma. \vdash^b \mathcal{A}^b \wedge \vdash^r \mathcal{A}^r \wedge \vdash \Sigma \wedge$$
$$(P, \Sigma, \mathcal{A}^b) \Downarrow^b t' \wedge (P, \Sigma, \mathcal{A}^r) \Downarrow^r t \implies t \approx_{\bar{A}} t'$$

Proof sketch: the simulator synthesizes a type derivation tree for the program and simply reinterprets all casted values as the correct type.

More importantly, type-confusion security composes. Combining the two simulation results shows that an attacker that confuses both base types and IFC types can be simulated by a well-typed attacker.

Lemma 3.4.3 (Composed Type-Confusion Security)**.** *Type confusion security holds between λ^b_{tc} and λ_{tc} for all well-typed P.*

$$\vdash P \implies \forall \mathcal{A}^b \,.\, \exists \mathcal{A}^i \,.\, \forall \Sigma. \vdash^b \mathcal{A}^b \wedge \vdash \mathcal{A}^i \wedge \vdash \Sigma \wedge$$
$$(P, \Sigma, \mathcal{A}^b) \Downarrow^b t' \wedge (P, \Sigma, \mathcal{A}^i) \Downarrow^i t \implies t \approx_{\bar{A}} t'$$

3.5 Proof of Type-Confusion Security

This section proves type-confusion security by constructing a well-typed ideal-world simulator for each real-world ill-typed attacker. The proof uses the standard technique of contextual backtranslation, which means that we compile all ill-typed attacker code into well-typed simulator code that simulates the behavior of the original attacker. The two major proof objectives are

- **Well-typed simulator:** the translation of the ill-typed attacker code is always well-typed.
- **Bisimulation:** the simulator produces the same public trace as the original attacker.

Since we assume that attackers can break pointer opacity, the pointer-equality operation is available to attacker code. The pointer-equality operations are then used as building blocks to construct language-level gadgets that implement version selection, down-casting, and remote reads and writes. Each gadget is accompanied by a proven admissible type rule. The gadgets are then used to implement the simulator translation of attacker code.

The simulator construction uses a few tricks to ensure that it generates well-typed code in well-typed contexts. We walk through the key ideas and present a proof that the simulator is well-typed.

Finally, we prove bisimulation between attacker and simulator. The simulation argument has four main components. First, we strengthen trace equivalence to the relation $\approx$, which tracks the extra structure needed by the simulation. Second, we prove a one-step unwinding theorem for the real-world transition relation: every step of λ^r_{tc} can be matched by zero or more steps of the ideal-world simulator while preserving expression, heap, and trace correspondence. Third, we lift the one-step result to arbitrary executions by induction over the reflexive-transitive closure of the real-world step relation. Finally, the remaining lemmas discharge structural obligations of the simulation, including how expression translation interacts with evaluation contexts and how the stronger trace relation implies the public trace relation used in the main theorem.

3.5.1 Pointer Equality for Attacker Code

We assume that the attacker—in both the ideal and real worlds—can test whether two values are the same pointer via the expression $v_1 \equiv v_2$. This capability is *not* available to honest parties; granting it only to the attacker strengthens the attacker model without expanding the trusted language.

The typing rule for pointer equality is shown below. The operands must have the same type, and the result is a boolean with the label of the operand type.

$$\text{A-EQ}\quad \frac{\Gamma^r; p \vdash^r v_1 : \tau^r \qquad \Gamma^r; p \vdash^r v_2 : \tau^r}{\Gamma^r; p \vdash^r v_1 \equiv v_2 : \mathsf{bool}}$$

$$\text{T-EQ}\quad \frac{\Gamma; pc; p; pc^{self}_{top} \vdash v_1 : \tau \qquad \Gamma; pc; p; pc^{self}_{top} \vdash v_2 : \tau \qquad \ell = \mathsf{lbl}(\tau)}{\Gamma; pc; p; pc^{self}_{top} \vdash v_1 \equiv v_2 : \mathsf{bool}_\ell}$$

The operational semantics for pointer equality are shown below.

$$\text{EQ-TRUE}\quad \frac{}{\langle p.f_\ell \equiv p.f_\ell, \Sigma, t\rangle \to \langle \mathsf{true}_\ell, \Sigma, t\rangle} \qquad \text{EQ-FALSE}\quad \frac{p.f \neq q.g}{\langle p.f_\ell \equiv q.g_\ell, \Sigma, t\rangle \to \langle \mathsf{false}_\ell, \Sigma, t\rangle}$$

3.5.2 Top-Level Simulator Construction

The top-level simulator construction $\mathcal{T}[\![P, \mathcal{A}]\!](p.f) = (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o, \lambda x.e)$ translates all function declarations and attacker function declarations into simulator definitions. For each real attacker function definition, the simulator constructs three versions:

- The **original version** is the original function definition, which is only used by a trusted principal.
- The $\top$**-version** simulates the behavior of the original function when the high integrity is locked.
- The $\bot$**-version** simulates the behavior of the original function when the low integrity is locked.

3.5.3 Translation of Function Types

The function signature of an original version of a function (from the simulator) is the same as the type of the original function (from the attacker) (S-ORIGINAL). This ensures that the simulator does not have the ability to replace function signatures.

The function signatures of the $\top$-version and $\bot$-version are normalized versions of the original function definition, denoted by $\mathcal{U}^\ell[\![\tau]\!]$. This normalized form sets all labels except toppc to $\top$, and sets toppc to ℓ, the parameter of the translation. This provides a canonical IFC type for each base type, which is essential for synthesizing types for the simulator. Formally:

Lemma 3.5.1 (Canonical Types)**.** *The following holds for all τ_1 and τ_2:*

$$\mathcal{B}[\![\tau_1]\!] = \mathcal{B}[\![\tau_2]\!] \iff \mathcal{U}^\ell[\![\tau_1]\!] = \mathcal{U}^\ell[\![\tau_2]\!]$$

Proof. We prove by induction on the type structure. The key observation is that $\mathcal{U}^\ell[\![\cdot]\!]$ replaces every label in τ with a fixed canonical value ($\top$ or ℓ depending on position), so its result depends only on the structural shape of τ, not on specific label values—the same information captured by $\mathcal{B}[\![\cdot]\!]$.

- **Unit.** Both $\mathcal{B}[\![1]\!]$ and $\mathcal{U}^\ell[\![1]\!]$ are constant, so both directions hold trivially.
- **Bool.** $\mathcal{B}[\![\mathsf{bool}_{\ell'}]\!]$ is independent of ℓ', and $\mathcal{U}^\ell[\![\mathsf{bool}_{\ell'}]\!] = \mathsf{bool}_\top$ is likewise independent of ℓ'. Both directions hold trivially for any two bool types.
- **Function.** Let $\tau_k = \tau_i^k \xrightarrow{pc_{ex}^k \lhd pc_{top}^k} \tau_o^k$ for $k \in \{1, 2\}$. Then $\mathcal{B}[\![\tau_1]\!] = \mathcal{B}[\![\tau_2]\!]$ iff $\mathcal{B}[\![\tau_i^1]\!] = \mathcal{B}[\![\tau_i^2]\!]$ and $\mathcal{B}[\![\tau_o^1]\!] = \mathcal{B}[\![\tau_o^2]\!]$; and $\mathcal{U}^\ell[\![\tau_1]\!] = \mathcal{U}^\ell[\![\tau_2]\!]$ iff the normalized input and output types are respectively equal. The equivalence follows by the induction hypothesis applied to τ_i^1, τ_i^2 and τ_o^1, τ_o^2.

□

3.5.4 Function Name Translation

Besides the original version of each function, the simulator also constructs two new versions of each function. As a result, the simulator needs fresh function names for these new versions.

These names exist because we assume that the namespace $\mathbb{F}$ is infinite and that programs are finite, and thus mention only finitely many functions. We denote the ℓ-version of each function $p.f$ as $\mathcal{N}^\ell[\![p.f]\!]$. $\mathcal{N}^\ell[\![p.f]\!]$ implicitly depends on the set of function names the P mentions. The result of name translation is always a fresh function name defined on a designated principal p_A, which is the least trusted principal (by construction, this principal is controlled by the attacker).

3.5.5 Store Name Translation

In order to simulate the original ill-typed attacker functions, the ℓ-versions of functions from the simulator need their own heap locations at the principal p_A. The store $\mathcal{H}[\![p.h]\!]$ translation is a store analogous to the function name translation. Instead of using the heap of the original function, the store translation uses the translated heap. Note that this heap is not parameterized by ℓ because both the $\top$-version and $\bot$-version of a function simulate the same original attacker function.

3.5.6 Expression Translation

Expressions are translated by an inductively defined $\mathcal{S}^\ell[\![e]\!]$. Here ℓ represents the top pc of the current execution. When $\ell = \top$, the expression should not call autoendorsing functions; when $\ell = \bot$, the expression is free to autoendorse.

Most expressions are translated homomorphically except for a few cases. Translation of read and write use the translated heap. Function applications are translated into applications of the ℓ-version of the function, thereby enforcing reentrancy security. Values v are translated into $\mathsf{C}^{\top}(v)$, which has canonical types.

3.5.7 Gadget: version selector

Unlike the previous meta-level translations, $\mathsf{C}^{\ell}(v)$ is a language-level gadget that dynamically selects which version of a function to call based on the current top pc ℓ. This makes a difference only when val is a variable x: $\mathsf{C}^{\ell}(x)$ is the identity translation when $\mathsf{C}^{\ell}(\cdot)$ is meta-level translation; When $\mathsf{C}^{\ell}(\cdot)$ is a language-level translation, $\mathsf{C}^{\ell}(x)$ turns x into its ℓ-version at run time.

This gadget can be implemented as a set of functions by the attacker, each parameterized by the ℓ and the type of input v (it does not need to be parameterized by a type if there is ad hoc polymorphism). The implementation of $\mathsf{C}^{\ell}(v)$ is a function call to a gadget whose body is a series of if statements that check whether the input is equal to a specific constant value and return its ℓ-version. At a high level, the implementation of $\mathsf{C}^{\ell}(v)$ is a table lookup that maps each constant value to its ℓ-version.

Of course, the gadget functions need fresh names, which we denote as $\mathcal{N}_v[\![\tau, \ell]\!]$ (defined on principal p_A). Such names exist because of the infinite function-namespace assumption.

Formally, $\mathsf{C}^{\ell}(v)$ is syntactic sugar for $(\mathcal{N}_v[\![\tau, \ell]\!])\ [\top]v$, where $\mathcal{N}_v[\![\tau, \ell]\!]$ is defined as follows.

Definition 3.5.2 (Version Selector)**.** Let $F_S = \mathcal{T}[\![P, \mathcal{A}]\!]$ be the simulator function table. Then:

$$F_S(\mathcal{N}_v[\![\tau, \ell]\!]) = (\tau \xrightarrow{\top \triangleleft \top} \mathcal{U}^{\ell}[\![\tau]\!], \lambda x.e)$$

Let $p_1.f_1, \cdots, p_n.f_n$ be all the function pointers typeable at τ. Equivalently, the selector could enumerate all pointers with the same erased type, using subsumption to type each equality test at a common supertype. When τ is a function type:

$$
\begin{aligned}
e = & \mathsf{if}\ x \equiv p_1.f_1\ \mathsf{then}\ \mathcal{N}^\ell[\![p_1.f_1]\!]\ \mathsf{else} \\
& \mathsf{if}\ x \equiv p_2.f_2\ \mathsf{then}\ \mathcal{N}^\ell[\![p_2.f_2]\!]\ \mathsf{else} \\
& \cdots \\
& \mathsf{if}\ x \equiv p_{n-1}.f_{n-1}\ \mathsf{then}\ \mathcal{N}^\ell[\![p_{n-1}.f_{n-1}]\!]\ \mathsf{else} \\
& \mathsf{if}\ x \equiv p_n.f_n\ \mathsf{then}\ \mathcal{N}^\ell[\![p_n.f_n]\!]\ \mathsf{else} \\
& \mathsf{abort}
\end{aligned}
$$

$e = (\mathsf{if}\ x\ \mathsf{then}\ \mathsf{true}_\top\ \mathsf{else}\ \mathsf{false}_\top)$ when τ is a bool type. $e = ()$ when τ is a unit type.

A simple argument shows that version selectors are well-typed and that the following typing rule is admissible.

$$
\text{T-VSELECTOR}\quad \frac{\Gamma; pc; p; pc_{top}^{self} \vdash v : \tau}{\Gamma; pc; p; pc_{top}^{self} \vdash \mathsf{C}^\ell(v) : \mathcal{U}^\ell[\![\tau]\!]}
$$

Lemma 3.5.3 (Well-typed Translated Pointer)**.** *If $p.f$ is typeable at τ, then $\mathcal{N}^\ell[\![p.f]\!]$ is typeable at $\mathcal{U}^\ell[\![\tau]\!]$.*

Proof. By the definition of $\mathcal{T}[\![P, \mathcal{A}]\!]$ and subsumption. □

Lemma 3.5.4 (Well-typed Version Selector)**.**

$$\forall \tau, \ell \vdash F_S(\mathcal{N}_v[\![\tau, \ell]\!])$$

Proof. The unit case is immediate, and the bool case follows by T-IF. For the function case, each test is well-typed by T-EQ, since the selector enumerates

pointers typeable at τ. Each successful branch is well-typed by the well-typed translated pointer lemma, and the default branch is well-typed by T-ABORT and subsumption. □

3.5.8 Gadget: attacker function down-caster

Notice that in the top level translation we overloaded the downgrade operator $v \downarrow \ell$ to also apply to function pointers. We use this with attacker-controlled labels, with the restriction that it does not downcast top pc. This can also be implemented as language-level gadget by the simulator, who uses a combination of auto-endorsing functions and downgrade expressions.

Let ℓ_A be the label that flows to all other attacker-controlled labels. Formally, $v \downarrow \tau$ where the type of v is τ' is a syntactic function defined as follows.

$$v \downarrow \tau = \begin{cases} 1 & \text{if } \tau' \text{ is unit type} \\ v \downarrow \mathsf{lbl}(\tau) & \text{if } \tau' \text{ is bool type} \\ (\mathcal{N}[\![v \downarrow \tau]\!])\ [\ell_A]v & \text{if } \tau' \text{ is function type} \end{cases}$$

Definition 3.5.5 (Attacker Function Down-Caster)**.** Let $F_S = \mathcal{T}[\![P, \mathcal{A}]\!]$ be the simulator function table. Let the type of f be τ', then:

$$F_S(\mathcal{N}[\![v \downarrow \tau]\!]) = (\tau' \xrightarrow{\top \triangleleft \top} \tau, \lambda f.e)$$

Let $\mathsf{lbl}(\tau) = \ell$. Let $p_1.f_1, \cdots, p_n.f_n$ be all the function pointers typeable at τ'.

When v is a variable:

$$\begin{aligned} e = &\mathsf{if}\ f \equiv p_1.f_1\ \mathsf{then}\ \mathcal{N}[\![p_1.f_1 : \tau]\!]_\ell\ \mathsf{else} \\ &\mathsf{if}\ f \equiv p_2.f_2\ \mathsf{then}\ \mathcal{N}[\![p_2.f_2 : \tau]\!]_\ell\ \mathsf{else} \\ &\cdots \\ &\mathsf{if}\ f \equiv p_{n-1}.f_{n-1}\ \mathsf{then}\ \mathcal{N}[\![p_{n-1}.f_{n-1} : \tau]\!]_\ell\ \mathsf{else} \\ &\mathsf{if}\ f \equiv p_n.f_n\ \mathsf{then}\ \mathcal{N}[\![p_n.f_n : \tau]\!]_\ell\ \mathsf{else} \\ &\mathsf{abort} \end{aligned}$$

When $v = p.f$ is a pointer, $e = \mathcal{N}[\![p.f : \tau]\!]_\ell$.

Here $\mathcal{N}[\![p.f : \tau]\!]$ is the name of a function that behaves the same as $p.f$ but has different type signature. Let $\tau = (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o)_\ell$ and let the type of $p.f$ be $\tau_i' \xrightarrow{pc'_{ex} \lhd pc'_{top}} \tau_o'$ Formally:

$$\begin{aligned} &F_S(\mathcal{N}[\![p.f : \tau]\!]) = \\ &\quad (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o, \lambda x.(p.f\ [pc'_{ex}](x \downarrow \tau_i')) \downarrow \tau_o) \end{aligned}$$

A simple argument shows that downcasters are well-typed and that the following typing rule is admissible.

$$\frac{}{A \vdash 1} \qquad \frac{\ell \in A}{A \vdash \mathsf{bool}_\ell} \qquad \frac{\ell, pc_{ex} \in A \qquad A \vdash \tau_i, \tau_o}{A \vdash (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o)_\ell}$$

T-DOWNCASTER

$$\frac{\Gamma; pc; p; pc_{top}^{self} \vdash v : \tau' \qquad A \vdash \tau, \tau' \qquad \mathsf{pctop}(\tau) = \mathsf{pctop}(\tau')}{\Gamma; pc; p; pc_{top}^{self} \vdash v \downarrow \tau : \tau}$$

Lemma 3.5.6 (Well-typed Downcaster)**.**

$$\forall v, \tau\ \vdash \mathcal{N}[\![v \downarrow \tau]\!] \wedge\ \vdash \mathcal{N}[\![v : \tau]\!]$$

3.5.9 Gadget: Remote Reader and Writer

We will need to implement the remote read and write expressions as language-level gadgets. Remote read and write expressions are simply implementations of the getter and setter functions.

Definition 3.5.7 (Reader and Writer)**.** Let $F_S = \mathcal{T}[\![P, \mathcal{A}]\!]$ be the simulator function table. We define $\mathsf{reader}_{p.h}$ and $\mathsf{writer}_{p.h}(v)$ as language-level gadgets that read from and write to the remote heap location $p.h$. As usual, we need fresh names for remote readers $\mathcal{N}_r[\![p.h]\!]$ and remote writers $\mathcal{N}_w[\![p.h]\!]$. Unlike previous gadgets, the remote reader and writer gadgets are defined on the heap location owner p instead of p_A. Formally:

$$F_S(\mathcal{N}_r[\![p.h]\!]) = (1 \xrightarrow{\top \triangleleft p} R(p.h), \lambda x.\mathsf{reader}_{p.h})$$
$$F_S(\mathcal{N}_w[\![p.h]\!]) = (R(p.h) \xrightarrow{\top \triangleleft p} 1, \lambda x.\mathsf{writer}_{p.h}(x))$$

Note that these readers and writers endorse the pc from $\top$ to p and the attacker is allowed to do this. If an honest party wishes to expose a reader and writer, it should set the external pc to be at least the label of the heap cell so that the reader and writer do not endorse control flow. A simple argument shows that readers and writers are well-typed and that the following typing rule is admissible.

T-READER
$$\frac{R(q.h) = \tau \qquad q \in A}{\Gamma; pc; p; q \vdash \mathsf{reader}_{q.h} : \tau}$$

T-WRITER
$$\frac{R(q.h) = \tau \qquad \Gamma; pc; p; pc_{top}^{self} \vdash v : \tau \qquad q \in A}{\Gamma; pc; p; q \vdash \mathsf{writer}_{q.h}(v) : \tau}$$

Lemma 3.5.8 (Well-typed Reader and Writer)**.**

$$\forall p, h \ \vdash \mathcal{N}_r[\![p.h]\!] \wedge \ \vdash \mathcal{N}_w[\![p.h]\!]$$

3.5.10 Heap Setup

At the start of simulator execution, the simulator needs to copy the heap of the ill-typed attacker to their translated locations in the simulator heap. As shown in S-MAIN, the main function first executes the heap setup code $\mathcal{H}_S[\![R]\!]$ that performs this copying and then executes the translation of the attacker main code $\mathcal{S}^{\bot}[\![e]\!]$. Note that the translation of main code uses $\bot$ as the parameter, which reflects the fact that the reentrancy lock is open and the attacker is free to autoendorse.

The heap translation $\mathcal{H}_S[\![M(A)]\!]$ is a sequence of read and write expressions that copy each value from the original heap to its translated location. Let $p_1.h_1, \cdots, p_n.h_n$ be the heap locations mentioned in the attacker code $M(A)$.

$$
\begin{aligned}
&\mathcal{H}_S[\![M(A)]\!] := \\
&\mathsf{let}\ x_1 = \mathsf{reader}_{p_1.h_1}\ \mathsf{in}\ \mathsf{let}\ x_1' = \mathsf{write}_{\mathcal{H}[\![p_1.h_1]\!]}(\mathsf{C}^{\top}(x_1))\ \mathsf{in} \\
&\mathsf{let}\ x_2 = \cdots\ \mathsf{in} \\
&\mathsf{let}\ x_n = \mathsf{reader}_{p_n.h_n}\ \mathsf{in}\ \mathsf{write}_{\mathcal{H}[\![p_n.h_n]\!]}(\mathsf{C}^{\top}(x_n))
\end{aligned}
$$

3.5.11 Well-typed Translation

Lemma 3.5.9 (Simulators are well-typed)**.**

$$\vdash^r M(A) \wedge \mathcal{T}[\![P, (A, M(A))]\!] = (A, M'(A)) \implies \vdash M'(A)$$

Proof. We do induction over the translation $\mathcal{S}[\![\cdot]\!]$.

- Case B-FIELD-DEF Let $\mathcal{U}^{\ell}[\![\tau_1]\!] = (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top} \sqcup \ell} \tau_o)_{\top}$ and $\mathcal{U}^{\ell}[\![\tau_2]\!] = \tau$.

  We need to show the following: $pc \sqcup \ell \sqsubseteq pc_{ex}$, $pc_{ex} \sqsubseteq pc_{top} \sqcup \ell \sqcup p$, $pc_{top}^{self} \sqsubseteq pc_{top}$ and $\tau \preccurlyeq \tau_i$. Note that $pc_{ex} = \ell = pc = p = pc_{top} = \top$, $\tau = \mathcal{U}^{\top}[\![\tau_2]\!] = \tau_i$ So all four conditions hold.

- Case B-UNIT, B-BOOL, B-FIELD, B-VAR hold by typing derivation of λ_{tc}.
- Case B-ABORT.
- Case B-OP, B-LET, B-ENDORSE. Holds by inductive hypothesis and typing derivation of λ_{tc}.
- Case B-IF $e = \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. Since the guard is downgraded to p_A the branches also type check under the pc p_A. Holds by inductive hypothesis and typing derivation of λ_{tc}.
- Case B-READ. To show this is well-typed, we need to show $pc \sqsubseteq \mathsf{lbl}(\tau)$. This follows from the definition of $\mathsf{C}^{\ell}(p.h)$ which ensures $\mathsf{lbl}(\tau') = \top = pc$.
- B-WRITE. To show $\Pi \vdash \mathsf{write}_{\mathsf{C}^{\ell}(p.h)}(v) : \tau$, we need to show $\tau' \preccurlyeq \tau$ and $pc \sqsubseteq \mathsf{lbl}(\tau)$. This holds by definition of $\mathsf{C}^{\ell}(p.h)$ and translation of values.
- Case B-APP By theorem 3.5.9, let $\mathcal{U}^{\ell}[\![\tau_1]\!] = (\tau_i \xrightarrow{pc_{ex} \lhd pc_{top} \sqcup \ell} \tau_o)_{\top}$ and $\mathcal{U}^{\ell}[\![\tau_2]\!] = \tau$.

  We need to show the following: $pc \sqcup \ell \sqsubseteq pc_{ex}$, $pc_{ex} \sqsubseteq pc_{top} \sqcup \ell \sqcup p$, $pc_{top}^{self} \sqsubseteq pc_{top}$ and $\tau \preccurlyeq \tau_i$. Note that $pc_{ex} = \ell = pc = p = \top$, $pc_{top} = \ell' \sqcup \ell$, $pc_{top}^{self} = \ell$, $\tau = \mathcal{U}^{\ell}[\![\tau_2]\!] = \tau_i$ So all four conditions hold.
- Case B-WRAPPER

Then we show that all attacker functions are well-typed. This requires shoing $\tau \preccurlyeq \tau_o$. Case 1 AT-SIGNATURE-TOP: holds by definition of $\mathcal{U}^{\ell}[\![\cdot]\!]$. Case 2 AT-SIGNATURE-EX: Same. □

3.5.12 Proof of Type Confusion

We prove type confusion using the following unwinding lemmas for $\rightarrow_{er}$ and $\rightarrow_{r}$, respectively. We use a stronger trace equivalence relation $\approx$ defined in fig. 3.14.

$\boxed{e \approx e'}$

$$\frac{\ell \in A \qquad \mathsf{cur}(E) \in A \qquad \mathsf{C}^\top(v) = \mathsf{C}^\top(v')}{E\,[q.g_\ell] \approx E\,[q'.g'_{\ell'}]}$$

$$\frac{\ell \notin A \qquad \mathsf{cur}(E) \in A \qquad q.g_\ell = q'.g'_{\ell'}}{E\,[q.g_\ell] \approx E\,[q'.g'_{\ell'}]} \qquad \frac{\mathsf{cur}(E) \in A \qquad v'_i = \mathsf{C}^\ell(v_i)}{E\,[v_1\ v_2] \approx E\,[v'_1\ v'_2]}$$

$$\frac{\text{other redex cases} \qquad \mathsf{C}^\ell(e) = \mathsf{C}^\ell(e')}{v \approx v'} \qquad \frac{E\,[e_i] \approx E\,[e'_i]}{E\,[\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2] \approx E\,[\mathsf{let}\ x = e'_1\ \mathsf{in}\ e'_2]}$$

$$\frac{q \notin A \qquad \mathsf{cur}(E) \in A \qquad q.g = q'.g' \qquad E\,[e] \approx E\,[e']}{E\,[\{e\}_{q.g}] \approx E\,[\{e'\}_{q'.g'}]}$$

$$\begin{aligned}
&\Sigma \approx \Sigma' \iff \forall p, h \\
&p \notin A \implies \Sigma(p.h) = \Sigma'(p.h) \wedge \\
&p \in A \implies \Sigma(\mathcal{H}[\![p.h]\!]) = \Sigma'(\mathcal{H}[\![p.h]\!])
\end{aligned}$$

$$w_{|C} = \begin{cases} (p.h := \mathsf{C}^\top(v)) & w = (p.h := v) \\ w & \text{otherwise} \end{cases}$$

$$t_{|C} = \begin{cases} t' & t = t'; w,\ w_{|C} = \epsilon \\ t'; w_{|C} & t = t'; w,\ w_{|C} \neq \epsilon \\ \epsilon & t = \epsilon \end{cases}$$

$$t \approx t' \iff t_{|C} = t'_{|C}$$

Figure 3.14: A stronger trace equivalence.

Theorem 3.5.10 (One Step Type-Confusion Security)**.** *Type-confusion security between λ_{tc} and λ^r_{tc} for all well-typed programs holds for $\rightarrow_r$.*

$$
\begin{aligned}
&\vdash P \wedge P \vDash \Sigma_{\mathcal{A}} \wedge P \vDash \Sigma_{\mathcal{A}^i}, \wedge \Sigma_{\mathcal{A}} \approx \Sigma_{\mathcal{A}^i} \wedge \\
&\quad \Gamma; pc; p; \ell \vdash e_{\mathcal{A}} : \tau_{\mathcal{A}} \ \wedge e_C \approx e_D \ \wedge t_C \approx t_D \wedge \\
&\quad \langle e_C, \Sigma_C, \mathsf{locks}, t_C \rangle \rightarrow_r \langle e_C, \Sigma'_C, \mathsf{locks}', t'_C \rangle \implies \\
&\quad \exists e'_D \ \Sigma'_D \ t'_D, \langle e_D, \Sigma_D, t_D \rangle \rightarrow^* \langle e'_D, \Sigma'_D, t'_D \rangle \wedge e'_C \approx e'_D \wedge \\
&\quad \Sigma'_C \approx \Sigma'_D \wedge t'_C \approx t'_D
\end{aligned}
$$

Proof. We do case analysis over the stepping relation of the real world $\rightarrow_{er}$.

- Case EVAL-EXPR. By the typing derivation, $e_C = E\,[e]$, $e'_C = E\,[e']$, and $\langle e, \Sigma_C, t_C \rangle \rightarrow_{er} \langle e', \Sigma'_C, t'_C \rangle$. We want to find e'_D such that $\langle \mathcal{S}, [\![E[e]]\!] \Sigma_D, t_D \rangle \rightarrow e'_D$. We do induction over the stepping relation of the real world $\rightarrow_{er}$. All except WRITE have $\Sigma'_D = \Sigma_D \approx \Sigma_C = \Sigma'_C$.
  - Case OP. By the typing derivation, $e'_C = v [\![\otimes]\!] v'$, which implies that $v = b_l$, $v = b'_{l'}$, and, as a consequence, $e_D = b_l \otimes b_{l'}$. Similarly, by typing derivation, $t_C \approx t_D$ and $\Sigma'_C = \Sigma_C \approx \Sigma_D$ (following a similar argument as above that most of the $\rightarrow_{er}$ rules do not change Σ. By theorem 3.5.12 and theorem 3.5.13, the case holds by EVAL-EXPR and OP.
  - Case IF-T and IF-F. Similar as above.
  - Case DOWNGRADE. Note that $\mathcal{S}^{=} [\![e_C]\!] \mathcal{S}^{l''} [\![b_{\ell'} \ell \downarrow]\!] = b_\top = e_D$, and Σ_C and t_C do not change. The case holds by taking zero step.
  - Case LET. We first note a fact that translation ($\mathcal{S}^\ell [\![e]\!]$) and substitution ($e\{v/x\}$) are commuting:

$$\mathcal{S}^\ell [\![e\{v/x\}]\!] = \mathcal{S}^\ell [\![e]\!] \{\mathcal{S}^\ell [\![v]\!]/x\}$$

The proof of this fact involves induction over the expression e, and the fact that translation is idempotent.

For the case, by typing derivation, $\mathcal{S}^{\ell}[\![\mathsf{let}\ x = v\ \mathsf{in}\ e]\!] = \mathsf{let}\ x = \mathcal{S}^{\ell}[\![v]\!]\ \mathsf{in}\ \mathcal{S}^{\ell}[\![e]\!]$. By the above lemma, $e'_D = E[\mathcal{S}^{\ell}[\![e\{v/x\}]\!]]$, which proves the claim using theorem 3.5.12 and theorem 3.5.13.

– Case RETURN-POP. By typing derivation, $q \notin A$, $t' = t_C; \mathsf{ret}(q)$, $e'_C = v$, and $e_C = \{v\}_{q.g}$. By definition, $\mathcal{S}^{\ell}[\![\{v\}_{q.g}]\!] = \{\mathcal{S}^{\ell}[\![v]\!]\}_{q.g}$. Because $\mathsf{C}^{\top}(v)$ is also value. Let $e'_D = E[\mathcal{S}^{\ell}[\![v]\!]]$. By theorem 3.5.12 and theorem 3.5.13, the case is proven by EVAL-EXPR and RETURN. RETURN-CALLEE-ADV follows the same proof structure.

- Case ABORT. This case holds by taking one step using theorem 3.5.12, theorem 3.5.13.
- Case READ. By typing derivation, $\mathsf{cur}(E) = p$, $\Sigma(p.h) = v$, $e_C = E\,[\mathsf{read}_h]$, and $e'_C = E\,[v]$. By assumption, e_D will equal to some $E'[\mathsf{read}_h]$, where $\mathcal{H}[\![p.h]\!] = p_A.h'$. Let $e'_D = E'[v']$, where $\Sigma'_D(p,h) = v'$. The case holds by the assumption that $\Sigma_C \approx \Sigma_D$. $t'_C = t_C \approx t_D = t'_D$, and $e'_C \approx e'_D$ by theorem 3.5.12 and theorem 3.5.13.
- Case WRITE. By typing derivation, $\mathsf{cur}(E) = p$, $\Sigma' = \Sigma\,[p.h \mapsto v]$, $t' = t; (p.h := v)$, $e_C = E\,[\mathsf{write}_h(v)]$, and $e'_C = E\,[()]$. By assumption, e_D will equal to some $E'[\mathsf{write}_h(v)]$, where $\mathcal{H}[\![p.h]\!] = p_A.h'$. Let $e'_D = E'[()]$, where $\Sigma'_D = \Sigma_D\,[p_A.h' \mapsto v]$. Then $\Sigma'_C \approx \Sigma'_D$. Also, $t'_C \approx t'_D$ by definition, and $e'_C \approx e'_D$ by theorem 3.5.12 and theorem 3.5.13.
- Case APP-REMOTE-CALLEE-ADV

  $e_C = v_1\ v_2$ where $v_1 = q.g_{\ell_1}$.

  $e'_C = \{e\{v_2/x\}\}_{q.g}$;e is the body of $q.g$. Let

$e_D = v_1'\ [\top] v_2'$. This function takes one step to

$e_D' = \{e'\{v_2'/x\}\}_{q'.g'}$;e' is body of $v_1' = q'.g'_{\ell_1'}$.

It remains to show $\{e\{v_2/x\}\}_{q.g} \approx \{e'\{v_2'/x\}\}_{q'.g'}$.

Because $q \in A$ holds by premise of APP-REMOTE-CALLEE-ADV, by definition of $\approx$ and commutivity of $\approx$ and substitution we only need to show $\mathsf{C}^\top(v_i) = \mathsf{C}^\top(v_i')$. This holds in general by definition of $\approx$.

- Case APP-REMOTE-OK.

  $e_C = v_1\ v_2$ where $v_1 = q.g_{\ell_1}$.

  $e_C' = \{e\{v_2/x\}\}_{q.g}$;e is the body of $q.g$. Let

  $e_D = v_1'\ [\top] v_2'$. This function takes one step to

  $e_D' = \{e'\{v_2'/x\}\}_{q'.g'}$;e' is body of $v_1' = q'.g'_{\ell_1'}$. Let the signature of v_1 be $\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o$.

  It remains to show $\{e\{v_2/x\}\}_{q.g} \approx \{e'\{v_2'/x\}\}_{q'.g'}$.

  Because $q \notin A$ holds by premise of APP-REMOTE-OK, by definition of $\approx$ we need to show:

  (1)$q'.g' = \mathsf{C}^\ell(q.g)$, (2) $e \approx e'$ and (3) $v_2 \approx v_2'$.

  Case on if $\mathsf{cur}(E) = p \in A$ or not.

  - $\mathsf{cur}(E) \in A$. Note that by premise of APP-REMOTE-OK, $\ell = \top$, so S-AUTOENDORSE-SUCCESS applies for e'. By definition of $\approx$ and S-AUTOENDORSE-SUCCESS , we have $v_i' = \mathsf{C}^\ell(v_i)$, which proves (1) and (2).

    Since $\mathsf{cur}(E) \in A$, $\ell_2 \in A$ by typing derivation of the body of $q.g$. (3) holds by definition of $\approx$.

  - $\mathsf{cur}(E) \notin A$. We first case on ℓ_1:

* $\ell_1 \notin A$. In this case $v_1 = v_1'$ and S-ORIGINAL applies. (1) and (2) hold. (3) holds by another casing over the label of v_2.
* $\ell_1 \in A$. By definition of $\approx$ and S-HIGH, we have $\mathsf{C}^\top(v_i) = \mathsf{C}^\top(v_i')$. Then by definition of $\approx$, (1), (2), and (3) all hold.

- Case APP-REMOTE-ABORT

  $e_C = v_1\ v_2$ where $v_1 = q.g_{\ell_1}$.

  $e_C' = \mathsf{abort}$. Let

  $e_D = v_1'\ [\top] v_2'$. This function takes one step to

  $e_D' = \{e'\{v_2'/x\}\}_{q'.g'}$;e' is body of $v_1' = q'.g'_{\ell_1'}$. We need to show that $e' = \mathsf{abort}$. Let the signature of v_1 be $\tau_i \xrightarrow{pc_{ex} \lhd pc_{top}} \tau_o$.

  By premise of APP-REMOTE-ABORT, either $pc_{ex} \notin A$ or $\ell = \top$ holds. In the first case e' takes the body of S-HIGH and in the second case the body takes of S-AUTOENDORSE-ABORT. It aborts in both cases.

□

Theorem 3.5.11 (Type-Confusion Security)**.** *Type-confusion security holds between λ_{tc} and λ_{tc}^r for all well-typed programs.*

$$\begin{aligned}
&\vdash P \wedge P \vDash \Sigma_{\mathcal{A}} \wedge P \vDash \Sigma_{\mathcal{A}^i}, \wedge \Sigma_{\mathcal{A}} \approx \Sigma_{\mathcal{A}^i} \wedge \\
&\quad \Gamma; pc; p; \ell \vdash e_{\mathcal{A}} : \tau_{\mathcal{A}}\ \wedge e_C \approx e_D\ \wedge t_C \approx t_D \wedge \\
&\quad \langle e_C, \Sigma_C, \mathsf{locks}, t_C \rangle \rightarrow_r \langle e_C, \Sigma_C', \mathsf{locks}', t_C' \rangle\ \Longrightarrow \\
&\quad \exists e_D'\ \Sigma_D'\ t_D', \langle e_D, \Sigma_D, t_D \rangle \rightarrow^* \langle e_D', \Sigma_D', t_D' \rangle \wedge e_C' \approx e_D' \wedge \\
&\quad \Sigma_C' \approx \Sigma_D' \wedge t_C' \approx t_D'
\end{aligned}$$

Proof. Induction on reflexive, transitive closure of the step. The claim vacuously holds for the reflexive case. For inductive case, we apply theorem 3.5.10 and the induction hypothesis. □

Lemma 3.5.12 (Translation of Evaluation Context)**.** *The following holds for translation of expression and evaluation context:*

$$\forall e\ E, \exists E', \mathcal{S}^{\ell}[\![E\,[e]]\!] = E'[\mathcal{S}^{\ell}[\![e]]\!]$$

Proof. Induction on E. We present here the LET case.

- Case $E = \mathsf{let}\ x = E'\ \mathsf{in}\ e'$. Therefore, $\mathcal{S}^{\ell}[\![E[e]]\!] = \mathsf{let}\ x = \mathcal{S}^{\ell}[\![E'[e]]\!]\ \mathsf{in}\ e'$. By induction hypothesis, there exists E'' such that $\mathcal{S}^{\ell}[\![E''[e]]\!] = E''[\mathcal{S}^{\ell}[\![e]]\!]$. Let $E' = \mathsf{let}\ x = E''\ \mathsf{in}\ e'$ proves the claim.

□

Lemma 3.5.13 (Uniqueness of Translation of Evaluation Context)**.** *The following holds for translation of expression and evaluation context:*

$$\begin{aligned}
&\forall e_1\ e_2\ E\ E_1\ E_2,\\
&\mathcal{S}^{\ell}[\![E[e_1]]\!] = E_1[\mathcal{S}^{\ell}[\![e_1]]\!] \wedge\\
&\mathcal{S}^{\ell}[\![E[e_2]]\!] = E_2[\mathcal{S}^{\ell}[\![e_2]]\!]\\
&\implies E_1 = E_2
\end{aligned}$$

Proof. Induction on E. This holds true because $\mathcal{S}^{\ell}[\![\cdot]\!]$ is a function. □

Lemma 3.5.14 (Simulation Relation)**.**

$$t \approx t' \implies t \approx_{\bar{A}} t'$$

Proof. By induction over the definition of $\approx$. □

Lemma 3.5.15 (Reentrancy Security, Real World)**.** *At any point of the execution, if a trusted function calls an untrusted function, then all functions called after that are also*

untrusted.

$$\langle \{e\}_{p_A.\mathsf{main}}, \Sigma, () \rangle \to^{r*} \langle e', \Sigma, t \rangle \wedge$$
$$\mathsf{stack}(t) = p_1.f_1, \ldots, p_n.f_n \implies$$
$$\exists i < j \in [1,n] \centerdot p_i \notin A \wedge p_j \in A \implies \forall k \in [j,n] \centerdot p_k \in A$$

Proof. By theorem 3.5.16 and the definition of $\to^r$. □

Lemma 3.5.16 (Lock)**.**

$$\langle \{e\}_{p_A.\mathsf{main}}, \Sigma, \mathsf{locks}, () \rangle \to^{r*} \langle e', \Sigma, \mathsf{locks}', t \rangle$$
$$\wedge\ \mathsf{stack}(e') = p_1.f_1, \ldots, p_n.f_n \implies \mathsf{locks}' \neq \emptyset$$

Proof. Note that the lock list only contains labels of trusted functions. The lemma hold by induction over the operational semantics. □

3.6 Applications

Our simulation result ensures that all hyperproperties satisfied by well-typed programs are also satisfied in the real world. The following subsections demonstrate the payoff: real-world (λ^r_{tc}) programs are CDA and reentrancy-secure, simply because well-typed (λ_{tc}) programs are. We also formalize noninterference preservation between the two settings.

3.6.1 Confused Deputy Attacks

As the example in fig. 3.1 shows, confused deputy attacks (CDAs) are violations of external pc policies. That is, a caller's *calling pc* level must not be less trusted than the callee's *external pc*.

Definition 3.6.1 (CDA call)**.** Let $P = (F, R)$ be a program. A call event is a CDA call when a caller's *calling pc* level is less trusted than the callee's required *external pc*.

Formally, let $F(q.g) = (\tau_i \xrightarrow{pc_{ex} \triangleleft pc_{top}} \tau_o, \lambda x.e)$, then:

$$\mathcal{CDA}(\mathsf{call}^{pc}(q.g)) \iff pc \not\sqsubseteq pc_{ex}$$

Definition 3.6.2 (CDA Security)**.** We define what it means for the tuple $(P, \Downarrow, Atks)$ to be CDA-secure, where P is a program, $\Downarrow$ is its semantics, and $Atks$ is the set of attacks under consideration. Informally, we simply require all its traces to be CDA-event free.

Formally, $(P, \Downarrow, Atks)$ is CDA-secure if, for all Σ and for all $\mathcal{A} = (A, M(A)) \in Atks$,

$$(P, \Sigma, \mathcal{A}) \Downarrow t \implies \forall\, w \in t_{|\bar{A}}\ \neg\mathcal{CDA}(w)$$

Well-typed programs are CDA-secure by construction.

Lemma 3.6.3 (Well-typed programs are CDA-secure)**.** *Let $\vdash P$ be a well-typed program and $AtksIdeal$ the set of well-typed ideal-world attacks. Then $(P, \Downarrow, AtksIdeal)$ is CDA-secure.*

Proof. The pc_{ex} restriction in T-App of the static semantics (fig. 3.8) enforces this restriction. □

Theorem 3.6.4 (CDA Security in the Real World)**.** *Let $\vdash P$ be a well-typed program and $AtksReal$ the set of real-world attacks. Then $(P, \Downarrow^r, AtksReal)$ is CDA-secure.*

Proof. Hyperproperty preservation is ensured by theorem 3.4.1, which we specialize to theorem 3.6.2 and theorem 3.6.3. □

3.6.2 Reentrancy Attack

A reentrancy attack occurs when an auto-endorsing function is called while a trusted level is locked. We formalize this property using the *call stack*: the sequence of principals actively called in a trace.

Definition 3.6.5 (Call Stack)**.** The call stack of a trace t is given by $\mathsf{stackof}(t)$. Push and pop operations are defined in the obvious way.

$$\mathsf{stackof}(t) = \begin{cases} \epsilon & t = () \\ \mathsf{push}(\mathsf{stackof}(t'), q) & t = t'; \mathsf{call}^{pc}(q) \\ \mathsf{pop}(\mathsf{stackof}(t')) & t = t'; \mathsf{ret}(q) \\ \mathsf{stackof}(t') & t = t'; other \end{cases}$$

Additionally, we say $t \leq t'$ when t is a prefix of t'.

Definition 3.6.6 (Reentrancy Security)**.** We define what it means for the tuple $(P, \Downarrow, Atks)$ to be reentrancy-secure, where P is a program, $\Downarrow$ is its semantics, and $Atks$ is the set of attacks under consideration.

Informally, we require that no intermediate trusted-function calls be made while a trusted function executes an untrusted call.

Formally, $(P, \Downarrow, Atks)$ is reentrancy-secure under the following condition.

Take Σ, $\mathcal{A} = (A, M(A)) \in Atks$, and $(P, \Sigma, \mathcal{A}) \Downarrow t$. Then consider all $t' \leq t_{|\bar{A}}$ and let $\mathsf{stackof}(t') = p_1, \ldots, p_n$. We require that in all such cases, if $\exists i\ p_i \notin A \wedge p_{i+1} \in A$, then $\forall k > (i+1)\ p_k \in A$.

Similar to CDA security, well-typed programs are reentrancy-secure in both worlds.

Theorem 3.6.7 (Reentrancy Security)**.** *Let $\vdash P$ be a well-typed program, $AtksIdeal$ the set of ideal-world attacks, and $AtksReal$ the set of real-world attacks.*

Then both $(P, \Downarrow, AtksIdeal)$ *and* $(P, \Downarrow^r, AtksReal)$ *are reentrancy-secure.*

Proof. The pc_{top} restriction in T-APP of the static semantics in fig. 3.8 establishes the ideal-world result. The real-world result follows from hyperproperty preservation (theorem 3.4.1). □

3.6.3 Noninterference

Noninterference is a security hyperproperty that ensures trusted information is not influenced by untrusted data and secret information does not influence public data.

Here, we define noninterference for integrity using low-equivalence $\approx_{\bar{A}}$ between Σ pairs, defined in fig. 3.12.

Definition 3.6.8 (Noninterference of Integrity)**.** The program-semantics pair $(P, \Downarrow)$ satisfies noninterference when for attackers who control the same set of principals A, running the program with trusted-equivalent initial configurations results in trusted-equivalent traces.

Formally, consider Σ_1, Σ_2, $\mathcal{A}_1^i = (A, M(A)_1)$, and $\mathcal{A}_2^i = (A, M(A)_2)$. We require

$$\Sigma_1 \approx_{\bar{A}} \Sigma_2 \implies$$
$$(P, \Sigma_1, \mathcal{A}_1^i) \Downarrow t_1 \wedge (P, \Sigma_2, \mathcal{A}_2^i) \Downarrow t_2 \implies$$
$$t_1 \approx_{\bar{A}} t_2$$

Theorem 3.6.9 (Noninterference Simulation)**.** *If noninterference holds for* $(P, \Downarrow)$*, then it also holds for* $(P, \Downarrow^r)$*.*

Proof. This result also follows from hyperproperty preservation ensured by theorem 3.4.1. □

3.7 Related Work and Discussion

Compiler Correctness and Secure Compilation

Our work can be viewed as a secure compilation result, where the source language λ_{tc} is compiled to the target language λ_{tc}^{r}. At a high level, our proof follows the standard context-based backtranslation strategy [33, 77]: for every target-level attacker, we construct a source-level attacker with matching observable behavior.

Prior backtranslation proofs often rely on logical relations, which are well suited to pure calculi such as the lambda calculus and STLC. Our setting is stateful and hyperproperty-based, so we instead use a bisimulation argument. In this respect, our proof is closest in spirit to the work of Patrignani, Künnemann, and Wahby [79], which relates universal composability [16] to robust hyperproperty preservation (RHP) [5]. Their proof is carried out in an ITM/RILC-style setting with cryptographic primitives, reactive traces, and nontrivial concurrency. By contrast, our calculus models single-threaded execution of code with high-level static security specifications.

The backtranslations also differ in character. UCRHP formulates the real/ideal correspondence primarily as trace equivalence between a real protocol/adversary and an ideal functionality/simulator: the simulator is correct insofar as it reproduces the same observable behavior. Our backtranslation is more intensional: it gives a syntactic, type-directed construction of a well-typed ideal attacker from an ill-typed real attacker.

Partial Typing in Open Systems

Riely and Hennessy [86] and Hennessy, Rathke, and Yoshida [53] both aim to provide secure execution in a decentralized system in which not all code is

well-typed. Their computation models are based on the π-calculus and differ from ours in three major ways: 1. instead of RPC calls, principals send code blocks to each other; 2. the type system does not track information flow; 3. their primary theoretical result is subject reduction. Lacking RPC calls, their computational model cannot reason about attacker-controlled runtimes, and cannot model standard CDA attacks.

CDA and Capability Systems

Rajani, Garg, and Rezk [82] formally characterize CDA vulnerabilities and show that capabilities cannot prevent all CDAs. Our work extends their idea in a more expressive setting: 1. instead of first-class references, we have function values, yielding a more expressive language and a stronger attacker model; 2. our real-world semantics does not assume a centralized, trusted runtime. They showed that CDAs are only prevented when *Full Provenance* is enforced, tracking all influences from both control flow and data. This is the same insight as using information flow control for both control flow and data flow in our work. Our major theoretical result is a simulation theorem that generalizes to other kinds of attacks, and CDA security is just an instance of our results.

Reentrancy Security

Our approach to reentrancy is inspired by SeRIF [19]. SeRIF is a core calculus that enforces secure reentrancy using a combination of static and dynamic IFC, and its attacker model is similar to that of our real world λ^{r}_{tc}. The language supports a wide range of language features that allow users to toggle between static and dynamic IFC for performance. Our real-world dynamic semantics is analogous to a SeRIF program that uses dynamic IFC mechanisms only. While

reentrancy security is not the main focus of our work, it is an essential ingredient of type-confusion security: the dynamic lock prevents an ill-typed attacker from gaining control-flow power unavailable to a well-typed ideal-world attacker.

Gradual Typing

While the problem of connecting a statically typed ideal world and an ill-typed real world is reminiscent of gradual typing [93, 101] where types can be left statically unspecified, our work solves an orthogonal problem. Unlike gradually typed languages, λ_{tc}^{r} assumes that all trustworthy top-level functions expose correct typing signatures. While prior work has been done on gradually typed IFC languages [14, 20], they focus on different semantic conditions like gradual guarantee [94] and noninterference [47].

Real–Ideal Simulation

Our definition of type-confusion security is inspired by the real–ideal paradigm widely used in the cryptography literature [16]. Patrignani, Wahby, and Künnemann [80] note that real–ideal simulation can be used as a compiler correctness criterion, which requires that the source level simulate the target-level behavior. When interpreted as compiler correctness, our type-confusion security theorem means that a compiler that compiles the ideal world to the real world is a secure compiler.

Control Flow Integrity (CFI)

Abadi, Budiu, Erlingsson, and Ligatti [3] propose Control Flow Integrity (CFI), a security policy that requires all executions to follow paths in a predetermined control-flow graph (CFG). Niu and Tan [78] explore Modular CFI (MCFI), a mech-

anism that enforces CFI in an open system where independently instrumented libraries are linked dynamically.

In fact, our reentrancy mechanism enforces a form of CFI over trusted principals, but our notion of control-flow integrity is stronger, because it forbids adversarial influence on trusted control flow, except through explicit auto-endorsement. This stronger notion is a key ingredient for proving type-confusion security.

Prior Implementations that Target Type Confusion

Several prior implementations of IFC systems [63, 104] aim to provide sound IFC security in a similar open world setting. Instead of building a specific system that solves mobile code, persistent objects, transactions, and exceptions, we focus on the semantic definition and formal model of type confusion. This allows us to better isolate the challenges of type confusion in remote procedure call-based systems, and provides a formal basis for the design of such systems.

Smart Contract Security

Our core calculus and security theorems provide a formal basis for the consensus-based [65, 100] design patterns that originated from the smart contract community. For example, the Checks–Effects–Interactions (CEI) pattern says callbacks should be executed after all state changes are made, enforcing secure reentrancy. The "pull over push" pattern recommends withdrawing over sending in transferring funds, which reduces risk of CDA attacks by reducing attenuated calls by the deputy.

3.8 Conclusions

Defining security in decentralized, reactive systems has long been a challenge. In this work, we introduced type-confusion security, a formal security definition that tames this complexity through a compositional approach to attacker models. The key insight for enforcing type-confusion security is enforcing security against attacks on control-flow integrity, which static security mechanisms struggle to protect against. We identify two well-known attacks on control-flow integrity, CDA and reentrancy attacks, and identify a simple static enforcement mechanism for each in the ideal world with well-typed attackers. We also propose dynamic enforcement mechanisms in the real world with ill-typed attackers, and use type-confusion security to show that the real world is as secure as the ideal world. Finally, we demonstrate the applicability of our results by applying type-confusion security to various security hyperproperties. The broader lesson is that static IFC alone is not enough at an open-world boundary: small, targeted dynamic mechanisms can recover the security reasoning available in a closed, well-typed system.

CHAPTER 4

CONCLUSION

In this thesis, we presented two studies of semantic definitions, enforcement mechanisms, and formal proofs of extensible language-based security. While the goal of the thesis is to design and build security-typed programming languages, our approach generalizes beyond information flow control. In the rest of this section, we highlight some future directions.

Dynamic Trust Delegation Chapter 2 studies delegation as a congruence on the trust lattice of principals and characterizes how delegation affects security hyperproperties such as noninterference and nonmalleable information flow control. The formal development in that chapter assumes a static delegation context and uses it as a parameter of the security hyperproperties. However, prior systems such as FLAM and FLAC show that delegation can also be a dynamic language mechanism, allowing principals to grant authority during execution [8, 9]. A natural next step is therefore to bring dynamic delegation, and possibly dynamic revocation, into the same algebraic framework. This would require modeling delegation as a language construct, giving typing rules for changing the delegation context during execution, and proving NMIF for the resulting language.

More Expressive Algebraic Subtyping The algebraic approach to labels suggests a broader direction for type systems with semantic annotations. Dolan's algebraic subtyping shows that an algebraic account of types can support principal type inference in an open world of extensible types [35]. In this thesis, we showed how such a treatment of types greatly benefits compiler implementations and enables sound and complete label inference and label polymorphism.

A natural future direction is to apply this style of algebraic subtyping to other type systems whose annotations already have clean abstract semantics. This would allow practical compiler implementations of these type systems to be built on top of an algebraic subtyping framework. Refinement types are a promising example: a refinement can be interpreted as an element of an abstract domain, and subtyping can be interpreted as inclusion between concretizations. Recent work on data-flow refinement type inference makes this connection explicit by parameterizing refinement inference over abstract domains such as intervals, octagons, and polyhedra [81]. Sized types for resource analysis provide a similar example, where types describe structural size bounds and the analysis is built as an abstract domain [92].

Effect and graded type systems provide another family of examples. Traditional effect systems often organize effects as a semilattice, while sequential effect systems use richer algebraic structures such as effect quantales to model ordered composition of effects [49, 50]. Graded modal type theories similarly parameterize typing over semiring-like structures that describe resource usage, erasure, or data flow [69]. In each of these settings, the typing rules could be stated against an abstract algebraic interface, while different analyses instantiate that interface with different semantic domains.

Type Confusion for Base Types Enforcing type-confusion security for IFC types and base types shares a common high-level intuition. In the case of IFC type systems, the central recipe that enables type-confusion security is a well-formedness condition over function types, a runtime check over the contravariant type argument, and the lock mechanism that ensures atomicity of high observable effects. In the case of base-type confusion, the key recipe is to ensure well-defined behavior when arbitrary values are used as function pointers. Although we have

a high-level intuition that type-confusion security can be achieved by runtime checks of contravariant type arguments at elimination forms, a formal account of this intuition is still missing.

Similar to algebraic subtyping, a universal approach to type confusion allows generalization to other type systems. One promising application is to study type confusion for type systems that track ownership, assets, or contract state in decentralized applications. Move's resource types enforce conservation properties for digital assets [15], and Obsidian combines typestate and ownership to rule out common smart-contract errors [25]. In both settings, untyped or legacy code that can misrepresent ownership, asset status, or object state would create a natural analogue of type confusion: trusted code reasons as if a value satisfies a rich static discipline, while the real value may not. Another promising application is protocol and session typing for distributed components. Session types and resource-aware session types specify the order and shape of interactions among parties, including in digital-contract settings [30]. In open systems, modules that employ session types may benefit from type-confusion security when they need to interact with untyped or legacy code that can misrepresent session types.

Type Confusion in More Expressive Languages The core calculus of type confusion is a simple programming language with first-class function pointers. Language features such as mobile code, classes and objects, inheritance, and dependent labels may create new challenges for type-confusion security. For example, mobile code may allow untrusted code to be injected into a trusted runtime, while full dynamic type checking imposes significant runtime overhead. Dependent labels enable dynamic configuration of security policies, which may create new attack vectors for type confusion. While object-oriented features do not necessarily introduce many theoretical challenges, inheritance and dynamic

dispatch would significantly increase the complexity of the proofs. Research in this direction requires not only formalizing and proving type-confusion security, but also designing new abstractions that balance performance, expressiveness, correctness, and usability.

Formal Verification Artifacts for Simulation Type confusion security is a secure-compilation theorem: the ideal language provides the source-level model in which programmers reason about security, while the real language models deployed code with runtime checks interacting with ill-typed adversaries. The proof follows the standard pattern of contextual backtranslation and bisimulation, but applies it to a stateful programming language with first-class function pointers. As suggested earlier in this section, a universal approach to type confusion allows generalization to other type systems. In such cases, mechanizing type-confusion security for these type systems would benefit from a common library of type-directed simulator construction and bisimulation.

Towards Practical Type-Confusion Security The theory of type-confusion security has guided the implementation of closures in the SCIF compiler. The next step is to connect type-confusion security to deployed systems. It would be worth considering how it can apply to a broader class of real systems that already face a gap between source-level type reasoning and untrusted or separately compiled code. Move is one such target in the domain of smart contract languages. Its resource types give a static abstraction for digital assets [15], while deployed smart-contract platforms must still defend against verifier bugs, cross-module interactions, and malicious bytecode. Type confusion security could clarify what runtime checks are needed so that violations of the static resource discipline do not give an ill-typed attacker more power. WebAssembly component systems

provide a second target. Wasm is already used as a portable sandboxed target, and RichWasm studies typed interoperability across languages with different memory and ownership disciplines [38, 51]. A type-confusion theorem for such components could characterize when adapters, monitors, or host-side checks make an untrusted component no more powerful than a well-typed one. Finally, typed RPC and service-interface frameworks provide a more speculative but natural setting. If service interfaces were enriched with effects, capabilities, session states, or information flow labels, type-confusion security would describe the boundary checks needed when a remote service advertises richer types than it actually respects.

BIBLIOGRAPHY

[1] Martín Abadi. Access control in a core calculus of dependency. In *11th ACM SIGPLAN Int'l Conf. on Functional Programming*, pages 263–273, New York, NY, USA, 2006. ACM.

[2] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, January 1999.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, New York, NY, USA, 2005. Association for Computing Machinery.

[4] Martín Abadi. Variations in access control logic. In Ron van der Meyden and Leendert van der Torre, editors, *Deontic Logic in Computer Science*, volume 5076 of *Lecture Notes in Computer Science*, pages 96–109. Springer Berlin Heidelberg, 2008.

[5] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *32nd IEEE Computer Security Foundations Symp. (CSF)*, pages 256–271. IEEE Computer Society, 2019.

[6] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew Myers, and Elaine Shi. Viaduct: An extensible, optimizing compiler for secure distributed programs. In *42nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 740–755. ACM, June 2021.

[7] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[8] Owen Arden, Jed Liu, and Andrew C. Myers. Flow-limited authorization. In *28th IEEE Computer Security Foundations Symp. (CSF)*, pages 569–583, July 2015.

[9] Owen Arden and Andrew C. Myers. A calculus for flow-limited authorization. In *29th IEEE Computer Security Foundations Symp. (CSF)*, pages 135–147, June 2016.

[10] Aslan Askarov and Stephen Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 308–322, 2012.

[11] Adam Barth, Adrienne Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. 01 2010.

[12] Giacomo Benedetti, Luca Verderame, and Alessio Merlo. Automatic security assessment of github actions workflows. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, SCORED'22, page 37–45, New York, NY, USA, 2022. Association for Computing Machinery.

[13] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA,

April 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[14] Abhishek Bichhawat, McKenna McCall, and Limin Jia. Gradual security types and gradual guarantees. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–16, 2021.

[15] Sam Blackshear, David Dill, Shaz Qadeer, Clark Barrett, John Mitchell, Oded Padon, and Yoni Zohar. Resources: A safe language abstraction for money, 04 2020.

[16] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Symposium on Foundations of Computer Science (FOCS)*, pages 136–145. IEEE Computer Society, 2001.

[17] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2–3):138–164, 1988. Also in *Readings in Object-Oriented Database Systems,* S. Zdonik and D. Maier, eds., Morgan Kaufmann, 1990.

[18] Ethan Cecchetti, Andrew C. Myers, and Owen Arden. Nonmalleable information flow control. In *24th ACM Conf. on Computer and Communications Security (CCS)*, pages 1875–1891. ACM, October 2017.

[19] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C. Myers. Compositional security for reentrant applications. In *IEEE Symp. on Security and Privacy*, May 2021.

[20] Tianyu Chen and Jeremy G. Siek. Quest complete: The holy grail of gradual security. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[21] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning.

In *21st ACM Symp. on Operating System Principles (SOSP)*, pages 31–44, October 2007.

[22] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *16th USENIX Security Symp.*, August 2007.

[23] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative information flow, relations and polymorphic types. *Journal of Logic and Computation*, 18(2):181–199, 2005.

[24] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *21st IEEE Computer Security Foundations Symp. (CSF)*, pages 51–65, June 2008.

[25] Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and assets for safer blockchain programming. *ACM Trans. on Programming Languages and Systems*, 42(3), November 2020.

[26] CoinDesk. Cross-chain DeFi site poly network hacked; hundreds of millions potentially lost. https://www.coindesk.com/markets/2021/08/10/cross-chain-defi-site-poly-network-hacked-hundreds-of-millions-potentially-lost, 10 August 2021. Accessed August 2023.

[27] CoinDesk. Defi protocol LI.FI struck by $11M exploit. https://www.coindesk.com/business/2024/07/16/defi-protocol-lifi-struck-by-8m-exploit/, 16 July 2024. Accessed October 2024.

[28] Cointelegraph. Dexible aggregator hacked for $2M via 'selfSwap' function. https://cointelegraph.com/news/

dexibleapp-aggregator-hacked-for-2m-via-selfswap-function, 17 February 2023. Accessed August 2023.

[29] Phil Daian. Analysis of the DAO exploit. https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/, 18 June 2016. Accessed March 2021.

[30] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. Resource-aware session types for digital contracts. In *34th IEEE Computer Security Foundations Symp. (CSF)*. IEEE, 2019.

[31] Dorothy E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, 1976.

[32] J. B. Dennis and E. C. VanHorn. Programming semantics for multiprogrammed computations. *Comm. of the ACM*, 9(3):143–155, March 1966.

[33] Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel. Modular, fully-abstract compilation by approximate back-translation. *Logical Methods in Computer Science*, Volume 13, Issue 4, October 2017.

[34] Monika di Angelo and Gernot Salzer. Consolidation of ground truth sets for weakness detection in smart contracts. In Aleksander Essex, Shin'ichiro Matsuo, Oksana Kulyk, Lewis Gudgeon, Ariah Klages-Mundt, Daniel Perez, Sam Werner, Andrea Bracciali, and Geoff Goodell, editors, *Financial Cryptography and Data Security. FC 2023 International Workshops*, pages 439–455, Cham, 2024. Springer Nature Switzerland.

[35] Stephen Dolan. *Algebraic subtyping: Distinguished Dissertation 2017*. BCS Learning & Development Ltd, Swindon, GBR, 2017.

[36] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *20th ACM Symp. on Operating System Principles (SOSP)*, October 2005.

[37] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, page 627–638, New York, NY, USA, 2011. Association for Computing Machinery.

[38] Michael Fitzgibbons, Zoe Paraskevopoulou, Noble Mushtak, Michelle Thalakottur, Jose Sulaiman Manzur, and Amal Ahmed. Richwasm: Bringing safe, fine-grained, shared-memory interoperability down to webassembly. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[39] Cédric Fournet, Guervan le Guernic, and Tamara Rezk. A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms. In *16th ACM Conf. on Computer and Communications Security (CCS)*, pages 432–441, November 2009.

[40] Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In *35th ACM Symp. on Principles of Programming Languages (POPL)*, pages 323–335, January 2008.

[41] J. Steven Fritzinger and Marianne Mueller. Java security. Technical report, Sun Microsystems, Inc., 1996.

[42] Ankit Gangwal, Haripriya Ravali Gangavalli, and Apoorva Thirupathi. A survey of layer-two blockchain protocols. *J. Netw. Comput. Appl.*, 209(C), January 2023.

[43] Deepak Garg and Martín Abadi. A modal deconstruction of access control logics. In *Proceedings of the Theory and Practice of Software, 11th International Conference on Foundations of Software Science and Computational Structures*, FOSSACS'08/ETAPS'08, page 216–230, Berlin, Heidelberg, 2008. Springer-Verlag.

[44] Deepak Garg, Lujo Bauer, Kevin Bowers, Frank Pfenning, and Michael Reiter. A linear logic of authorization and knowledge. pages 297–312, 09 2006.

[45] Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, 2006.

[46] Valerio Genovese, Deepak Garg, and Daniele Rispoli. Labeled sequent calculi for access control logics: Countermodels, saturation and abduction. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 139–153, 2012.

[47] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

[48] Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *IEEE Symp. on Security and Privacy*, pages 75–86, April 1984.

[49] Colin S. Gordon. A Generic Approach to Flow-Sensitive Polymorphic Effects. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:31, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[50] Colin S. Gordon. Polymorphic iterable sequential effect systems. *ACM Trans. Program. Lang. Syst.*, 43(1), April 2021.

[51] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with webassembly. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 185–200. ACM, June 2017.

[52] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, October 1988.

[53] Matthew Hennessy, Julian Rathke, and Nobuko Yoshida. SAFEDPI: A language for controlling mobile code. *Acta Informatica*, 42(4):227–290, 2005.

[54] Andrew K. Hirsch, Pedro H. Azevedo de Amorim, Ethan Cecchetti, Ross Tate, and Owen Arden. First-order logic for flow-limited authorization. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 123–138, 2020.

[55] Andrew K. Hirsch and Michael R. Clarkson. Belief semantics of authorization logic. CCS '13, New York, NY, USA, 2013. Association for Computing Machinery.

[56] Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *J. Cryptol.*, 13(1):31–60, 2000.

[57] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[58] Eliza Kozyri, Stephen Chong, and Andrew C. Myers. Expressing information flow properties. *Foundations and Trends in Privacy and Security*, 3(1):1–102, 2022.

[59] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *21st ACM Symp. on Operating System Principles (SOSP)*, 2007.

[60] Jaisook Landauer and Timothy Redmond. A lattice of information. In *6th IEEE Computer Security Foundations Workshop (CSFW)*, pages 65–70. IEEE Computer Society Press, June 1993.

[61] Ninghui Li, Benjamin N Grosof, and Joan Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):128–171, 2003.

[62] Peixuan Li and Danfeng Zhang. Towards a general-purpose dynamic information flow policy, 2021.

[63] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *22nd ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, October 2009.

[64] Yue Liu, Chakkrit Tantithamthavorn, and Li Li. Protect Your Secrets: Understanding and Measuring Data Exposure in VSCode Extensions . In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 551–562, Los Alamitos, CA, USA, March 2025. IEEE Computer Society.

[65] Lodovica Marchesi, Livio Pompianu, and Roberto Tonelli. Security checklists for Ethereum smart contract development: patterns and best practices. *Blockchain: Research and Applications*, page 100367, 2025.

[66] Sinisa Matetic, Moritz Schneider, Andrew Miller, Ari Juels, and Srdjan Capkun. Delegatee: Brokered delegation using trusted execution environments. In *USENIX Security Symposium*, pages 1387–1403, 2018.

[67] Nicholas D. Matsakis and Felix S. Klock. The Rust language. In *ACM SIGAda Ada Letters*, October 2014.

[68] Benoît Montagu, Benjamin C. Pierce, and Randy Pollack. A theory of information-flow labels. In *26th IEEE Computer Security Foundations Symp. (CSF)*, pages 3–17, June 2013.

[69] Benjamin Moon, Harley Eades III, and Dominic Orchard. Graded modal dependent type theory. In *Programming Languages and Systems: 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings*, page 462–490, Berlin, Heidelberg, 2021. Springer-Verlag.

[70] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, January 1999.

[71] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *16th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, October 1997.

[72] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[73] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *17th IEEE Computer Security Foundations Workshop (CSFW)*, pages 172–186, June 2004.

[74] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

[75] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow. Software release, http://www.cs.cornell.edu/jif, July 2006.

[76] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[77] Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 103–116, New York, NY, USA, 2016. Association for Computing Machinery.

[78] Ben Niu and Gang Tan. Modular control-flow integrity. *SIGPLAN Not.*, 49(6):577–587, June 2014.

[79] Marco Patrignani, Robert Künnemann, and Riad S. Wahby. Universal composability is robust compilation. arXiv ePrint 1910.08634, 2022.

[80] Marco Patrignani, Riad S. Wahby, and Robert Künneman. Universal composability is secure compilation. arXiv ePrint 1910.08634, 2019.

[81] Zvonimir Pavlinovic, Yusen Su, and Thomas Wies. Data flow refinement type inference. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[82] Vineet Rajani, Deepak Garg, and Tamara Rezk. On access control, capabilities, their equivalence, and confused deputy attacks. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 150–163, June 2016.

[83] Jakob Rehof and Torben Æ. Mogensen. Tractable constraints in finite semilattices. In *3rd International Symposium on Static Analysis*, number 1145 in Lecture Notes in Computer Science, pages 285–300. Springer-Verlag, September 1996.

[84] Silei Ren, Coşku Acay, and Andrew C. Myers. An algebraic approach to asymmetric delegation and polymorphic label inference. In *European Symposium on Research in Computer Security (ESORICS)*, September 2025.

[85] Silei Ren, Coşku Acay, and Andrew C. Myers. An algebraic approach to asymmetric delegation and polymorphic label inference (technical report), April 2025.

[86] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. *J. Automated Reasoning*, 31(3):335–370, 2003.

[87] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[88] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3–4):289–360, November 1993.

[89] Ravi S. Sandhu. Role hierarchies and constraints for lattice-based access controls. In *4th European Symp. on Research in Computer Security (ESORICS)*, September 1996.

[90] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2001. Also available as TR 99-1759, Computer Science Department, Cornell University, Ithaca, New York.

[91] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. eThor: Practical and provably sound static analysis of ethereum smart contracts. In *27th ACM Conf. on Computer and Communications Security (CCS)*, page 621–640, November 2020.

[92] Alejandro Serrano, Pedro López-García, and Manuel V Hermenegildo. Resource usage analysis of logic programs via abstract interpretation using sized types. *Theory and Practice of Logic Programming*, 14(4-5):739–754, 2014.

[93] Jeremy Siek and Walid Taha. Gradual typing for objects. In *21st European Conf. on Object-Oriented Programming*, pages 2–27, July 2007.

[94] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *Summit on Advances in Programming Languages*, 2015.

[95] Vincent Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.

[96] Matvey Soloviev, Musard Balliu, and Roberto Guanciale. Security properties through the lens of modal logic, 2023.

[97] Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In Peeter Laud, editor, *Proceedings of the 16th Nordic conference on Information Security Technology for Applications*, volume 7161 of *Lecture Notes in Computer Science*, pages 223–239. Springer, 2011.

[98] Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. Managing policy updates in security-typed languages. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 202–216, July 2006.

[99] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 67–82, New York, NY, USA, October 2018. Association for Computing Machinery.

[100] Franz Volland. Solidity patterns: A compilation of patterns and best practices for the smart contract programming language solidity. https://github.com/fravoll/solidity-patterns, 2019.

[101] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, 2009.

[102] W. G. Wood. Recovery control of communicating processes in a distributed system. Technical Report 158, University of Newcastle upon Tyne, 1980.

[103] Andrew C. Yao. Protocols for secure computations. In *23rd annual IEEE Symposium on Foundations of Computer Science*, pages 160–164, 1982.

[104] Siqiu Yao, Haobin Ni, Andrew C. Myers, and Ethan Cecchetti. SCIF: A language for compositional smart contract security. Technical Report abs/2407.01204, arXiv, July 2024.

[105] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. Using information flow to design an ISA that controls timing channels. In *32nd IEEE Computer Security Foundations Symp. (CSF)*, June 2019.

[106] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 15–23, June 2001.

[107] Steve Zdancewic and Andrew C. Myers. Secure information flow and CPS. In David Sands, editor, *10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 46–61. Springer Berlin Heidelberg, 2001.

[108] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. on Computer Systems*, 20(3):283–328, August 2002.

[109] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

[110] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton Jones. SHErrLoc: A static holistic error locator. *ACM Trans. on Programming Languages and Systems*, 39(4):18, August 2017.

[111] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *IEEE Symp. on Security and Privacy*, pages 236–250, May 2003.

[112] Lantian Zheng and Andrew C. Myers. A language-based approach to

secure quorum replication. In *9th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, August 2014.