TOWARDS A SECURE FEDERATED INFORMATION SYSTEM

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

Mon Jed Liu

August 2012

TOWARDS A SECURE FEDERATED INFORMATION SYSTEM

Mon Jed Liu, Ph.D.

Cornell University 2012

We are entering an era in which federated information systems are widely used to share information and computation. Federated systems support new services and capabilities by integrating computer systems across independent administrative domains. Each domain has policies for security, but does not fully trust other domains to enforce them. This dissertation explores, in two parts, the challenge of designing and building federated information systems that are secure and reliable while supporting mutually distrusting participants.

First, this dissertation presents Fabric, a new system and language for building secure federated information systems. Fabric allows heterogeneous network nodes to securely share information and computation despite mutual distrust. It uses optimistic, nested transactions to ensure global consistency, and has a peer-to-peer dissemination layer for better availability and load balancing. Fabric's high-level programming language provides a rich, Java-like object model, and keeps distribution and persistence largely transparent to programmers. It supports data shipping and function shipping: both information and computation can move between nodes to meet security requirements or to improve performance. Confidentiality and integrity policies on objects are enforced through a combination of compile-time and run-time mechanisms. Results from building Fabric applications suggest that Fabric has a clean and concise programming model, offers good performance, and enforces security.

Next, this dissertation examines the security implications of providing referential integrity in a federated system. Referential integrity ensures that named resources can be accessed when needed. This is an important property for re-

liability and security. However, the attempt to provide referential integrity can itself lead to security vulnerabilities that are currently not well understood. This dissertation identifies three such *referential security* vulnerabilities, and formalizes security conditions corresponding to their absence. A language model captures key aspects of programming distributed systems with named, persistent resources in the presence of an adversary. A new type system is proved to enforce the conditions for referential security.

BIOGRAPHICAL SKETCH

Mon Jed Liu was born in Toronto, Canada to wonderful and loving parents. Jed first became interested in computers at a young age, after having been exposed to them at a summer camp. His first computer was a laptop he built at age seven. It had a stunning 10.1" display and was quite literally paper-thin, putting today's MacBook Air to shame. It supported video calling over a built-in wireless radio, long before 802.11 and Skype existed. It was a glorious contraption made out of loose-leaf paper and powered by imagination.

Jed's second computer was delicious and lasted only a few days—a birthday cake with an image of a computer printed on it. His third computer was the rather entertaining Family Computer, which introduced him to such timeless classics as *The Legend of Zelda* and *Dragon Warrior*.

Computers became a little more serious for Jed at the age of ten, with his fourth computer, an Apple IIGS. On it, between sessions of playing *Pool of Radiance*, *Carmen Sandiego*, and *Earl Weaver Baseball* with his brothers, he taught himself how to program, in Applesoft BASIC. Sadly, his masterful creations were only stored in RAM, and are now lost to the sands of time.

Jed has since built several more computers, not all of which were based on arboreal cellulose technology. Within his family, he has earned the title of Computer Assistant, and has an awesome nameplate to prove it. He graduated from Upper Canada College in 1997 and received the Governor General's Academic Medal. He received a Bachelor of Arts in Mathematics (*cum laude*) and Computer Science (*magna cum laude*), with distinction in all subjects, from Cornell University in 2001. The following year, he received a Master of Engineering in Computer Science, also from Cornell. As part of his doctoral studies at Cornell, he completed a graduate minor in mathematics.

For 阿公, 阿嬤, 爸爸, 媽媽, *Richard, and Rex.*

ACKNOWLEDGEMENTS

Many people deserve my thanks for their role in making this dissertation and my degree a reality. First and foremost, I wish to thank my family. To 阿公, thank you for instilling in me my love of mathematics, which forms the foundation of so much of my life, including my passion for computer science. To 阿嬤, thank you for being so supportive of my academic pursuits. You were always incredibly proud of me, Richard, and Rex. I wish you and 阿公 had lived to see us graduate. To Mom and Dad, I cannot thank you enough for always being so loving and supportive. You have worked very hard and sacrificed too much to ensure that I get the best possible education, and you have consistently encouraged me to do what I love, to the best of my ability. To Richard and Rex, you have been invariably proud and supportive of me. I am lucky to have you as brothers, and I wish we could spend more time together. To all six of you, I owe much gratitude, and it is to you that I dedicate this dissertation.

My advisor, Andrew Myers, has been invaluable in his guidance of my research. Andrew has been an inspiration from the moment I stepped into his CS 412 class in January of 2000. He is never afraid to tackle difficult questions, and his high standards have made him a wonderful role model. He has always been eager to share his sharp insight, superb advice, and keen enthusiasm for research. Andrew has unfailingly provided encouragement when I needed it most, and for this, I am truly indebted to him. I also thank the other members of my thesis committee, Robbert van Renesse and Ravi Ramakrishna, for their insightful feedback on this dissertation.

Research is much more rewarding when done collaboratively, and I have been fortunate to have worked with many talented individuals. I thank my fellow Fabricators, who have made working on Fabric a rich and engaging learn-

ing experience: Owen Arden, Aslan Askarov, Mike George, Nate Nystrom, Xin Qi, Krishnaprasad Vikram, Lucas Waye, and Xin Zheng. Mike was especially instrumental in the design and implementation of Fabric, and this dissertation would not have been possible without him. My thanks also to my non-Fabric collaborators: Aaron Kimball, Stephen Chong, and Lantian Zheng.

I owe a debt to my labmates in the Cornell Systems Lab (Syslab): Mahesh Balakrishnan, Hitesh Ballani, Tuan Cao, Saikat Guha, Oliver Kennedy, and Alan Shieh. They were invariably a good avenue of procrastination, be it through random discussions, movies, video games, or nerf guns. Becky Stewart and Stephanie Meik have always been available with a smile to provide friendly advice on navigating the Big Red Tape.

Many thanks to the Cornell Karate Club. Karate has provided a much-needed outlet for the frustrations that inevitably arise from difficult research. As sensei, Mike Eschenbrenner has provided insightful guidance in my training. My fellow karatekas Gabriella Bensur, José Delgado, and Marissa Giovino have been essential in making training fun. I especially thank Gabriella for being a wonderful friend. Our conversations have always brightened my day, whether they be about life, school, or just plain geekery.

Finally, I would like to add a big thanks to many others who have made my graduate life fun over the years: Kavita Bala, Mark Bushnell, Meghan Dowd, Marisa Genuardi, Ingrid Kiehl, Leeann Louis, Joe McCourt, Jeff Mermin, Ryan O'Neil, Derek Plotkowski, Lena Sawin, Brian Shore, Steve Sinnott, Tim Snapp, John Thacker, Barbara Varsanofieva, and Marielle Volz.

The research reported in this dissertation was supported in part by the National Science Foundation under grants 0430161, 0541217, 0627649, and 0964409; by a grant from Microsoft Corporation; by the Air Force Team for Research in

Ubiquitous Secure Technology (AF-TRUST), which receives support from the DAF Air Force Office of Scientific Research (FA9550-06-1-0244) and the NSF (0424422); by National Intelligence Community Enterprise Cyber Assurance Program (NICECAP) Grant FA8750-08-2-0079, monitored by Air Force Research Laboratories; and by the Office of Naval Research under award N00014-09-1-0652. This work does not necessarily represent the opinions, expressed or implied, of any of these sponsors.

The graphic representing the researcher in Figure 1.1 is the Aperture Science logo from the video game *Portal* by Valve Corporation. In the same figure, the graphic representing the pharmaceutical company is adapted from the logo for Prescott Pharmaceuticals, proud sponsor of *Cheating Death with Dr. Stephen T. Colbert, D.F.A.* on the television program *The Colbert Report*. "Prescott: the only medicine on the market with a cap so simple, even a child could open it."

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Distributed information systems network computers together to provide fast, efficient access to data and computation. We rely on complex, distributed information systems for many important activities. Government agencies, banks, hospitals, schools, and many other enterprises use distributed information systems to manage information and interact with the public.

These systems often have security requirements, of which there are three central goals: *confidentiality*, that information be protected from unauthorized disclosure; *integrity*, that information be protected from unauthorized modification; and *availability*, that information be protected from loss of use [18]. Violations of these requirements can have financial, legal, and ethical consequences. Current practice does not offer general, principled techniques for implementing the functionality of these systems while also satisfying their security requirements.

The trend towards integration makes the security problem ever more important and difficult. Whereas distributed information systems are operated by individual administrative domains, *federated* systems provide new services and capabilities by connecting these systems together. However, each domain has its own policies for security, and does not fully trust other domains to enforce them. Future information systems will need to function correctly and securely despite having code and data distributed across these trust boundaries.

This dissertation examines in two ways the challenge of designing and building federated information systems that are secure and reliable. The first half presents the design and implementation of Fabric, a platform for building federated information systems with confidentiality and integrity assurances.

While Fabric offers confidentiality and integrity assurances, these alone are not sufficient for building secure and reliable federated systems. The challenge is made more difficult because information is not merely bits—it has structure. Information resources use names to refer to other, perhaps remote, resources. For example, web pages can have hyperlinks to other pages, tuples in relational databases can have foreign keys that refer to other tuples in the database, and objects in distributed object systems can point to other objects in the system. The precise details of how references are represented are not germane, so we refer to all these constructs collectively as *references*. Similarly, we use the term *objects* to refer generically to information resources that are connected by references, whether web pages, database tuples, or distributed objects. Regardless of the kind of system, security and reliability vulnerabilities are created when references cross trust boundaries in a distributed system.

The second half of this dissertation identifies three kinds of *referential security* vulnerabilities, a class of availability vulnerabilities that appear in federated information systems with persistent information. It formally characterizes these vulnerabilities and explores a language-based approach for modelling, analyzing, and preventing them.

1.1 Example

Epidemiological research is one domain that would benefit from a secure, federated information system. Because of patient confidentiality concerns, researchers have limited access to epidemiological data in the US and Canada, especially for STIs such as HIV/AIDS. When available, data sets are highly regional—often restricted to a single city—and in some cases must be obtained in person or hand-delivered by courier. The data is incomplete because it does

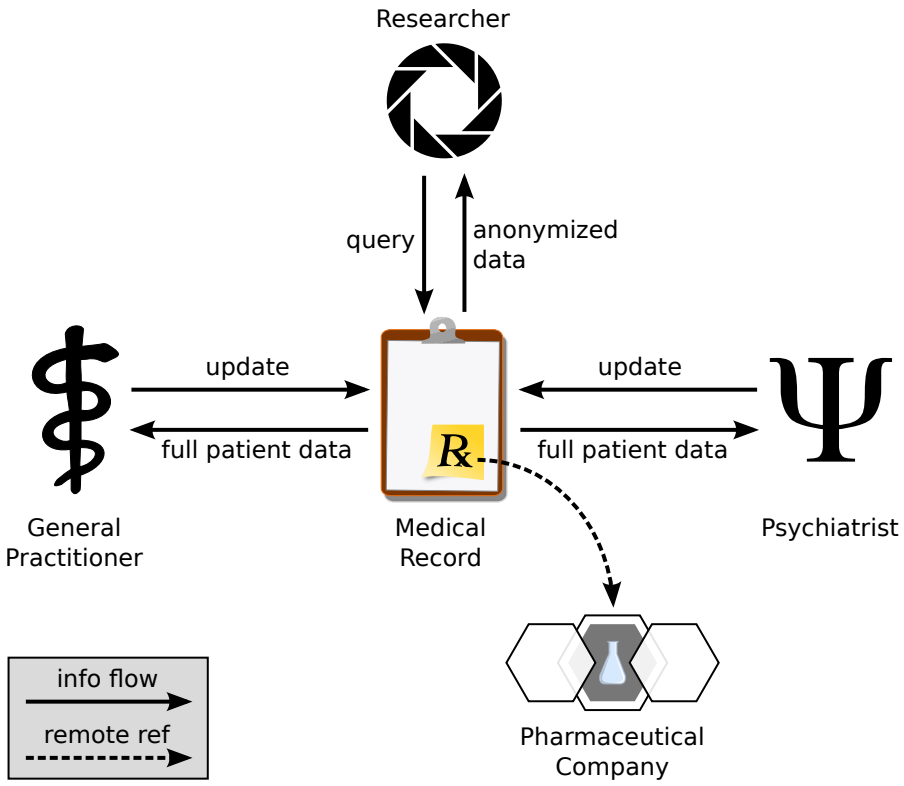


Figure 1.1: A medical example

not include full patient records. Instead, it is derived from population surveys and diagnosis forms that are unable to fully anticipate researchers' needs [75]. A secure, federated information system would allow researchers to directly mine patient records for real-time epidemiological trends while protecting patient confidentiality.

To illustrate the challenges, consider the scenario in Figure 1.1. A patient, Alice, has two doctors—a general practitioner and a psychiatrist—and has given consent for her medical record to be used anonymously for research. The medical record is kept in a federated information system. This not only provides researchers with direct access to her medical data, but her two doctors at separate medical institutions can securely and quickly share her medical information.[1]

[1]The ability of doctors to effectively collaborate on patient information is important. According to a 1999 Institute of Medicine study, at least 44,000 US deaths annually result from medical errors, with incomplete patient information identified as a leading cause [40].

Automated sharing of patient data poses difficulties. First, the security and privacy policies of the two institutions must be satisfied (as mandated by the Health Insurance Portability and Accountability Act (HIPAA) [34] in the US), restricting which information can be shared or modified by the two institutions. While Alice's doctors might have access to her complete medical record, the researchers' view of the record should not contain any personally identifying information, such as Alice's address or phone number.

Second, the patient record may be updated by both institutions as treatment progresses, yet the record should be consistent and up to date when viewed by the researcher and from the two institutions. It is inadequate to simply transmit a copy of the record in a common format such as XML, because the copy and the original are likely to diverge over time. Instead, the researcher and institutions should have easy, secure, consistent and efficient access to what is logically a single patient record.

Third, the record is likely composed of several objects, with references to other, perhaps remote, objects. For example, an object representing a prescription might refer to an object at a pharmaceutical company, identifying the prescribed drug and providing information about it, such as contraindications, side effects, and drug interactions. This is useful, because when the pharmaceutical company adds new information about the drug, it is instantly reflected in Alice's medical record. However, the *referential integrity* of Alice's record is now dependent on the pharmaceutical company: information in Alice's medical record becomes missing if the drug information is deleted or is temporarily unavailable. It is therefore important that either the pharmaceutical company be trusted to enforce this referential integrity, or that users of Alice's medical record be prepared for a potential failure in referential integrity.

1.2 Contributions of this dissertation

1.2.1 Secure federated computation and storage

This dissertation presents the design and implementation of Fabric, a federated system that supports secure, shared access to information and computation, despite distrust between cooperating entities. The goal of Fabric is to make secure distributed applications much easier to develop, and to enable the secure integration of information systems controlled by different organizations.

To achieve this goal, Fabric provides a shared computational and storage substrate implemented by an essentially unbounded number of Internet hosts. As with the Web, there is no notion of an "instance" of Fabric. Two previously non-interacting sets of Fabric nodes can interact and share information without prior arrangement. There is no centralized control over admission: new nodes, even untrustworthy nodes, can join the system freely.

Untrustworthy nodes pose a challenge for security. The guiding principle for security in Fabric is that one's security should never depend on components of the system that one does not trust. Fabric provides security assurance through a combination of mechanisms at the language and system levels.

Fabric gives programmers a high-level programming abstraction in which security policies and some distributed computing features are explicitly visible to the programmer. Programmers access Fabric objects in a uniform way, even though the objects may be local or remote, persistent or non-persistent, and object references may cross between Fabric nodes.

The Fabric programming language is an extension to the Jif programming language [53,56], in turn based on Java [30]. Fabric extends Jif with support for distributed programming and transactions. Like Jif, Fabric has several mech-

anisms, including access control and information-flow control, to prevent untrusted nodes from violating confidentiality and integrity. All objects in Fabric are labelled with policies from the *decentralized label model* (DLM) [54], which expresses security requirements in terms of principals (e.g., users and organizations). Object labels prevent a node that is not trusted by a given principal from compromising the security policies of that principal. Therefore, Fabric has fine-grained trust management that allows principals to control to what extent other principals (and nodes) can learn about or affect their information.

To achieve good performance while enforcing security, Fabric supports both *data shipping*, in which data moves to where computation is happening, and *function shipping*, in which computations move to where data resides. Data shipping enables Fabric nodes to compute using cached copies of remote objects, with good performance when the cache is populated. Function shipping enables computations to span multiple nodes. Inconsistency is prevented by performing all object updates within transactions, which are exposed at the language level. The availability of information, and scalability of Fabric, are increased by replicating objects within a peer-to-peer *dissemination layer*.

Of course, there has been much previous work on making distributed systems both easier to build and more secure. Prior mechanisms for remotely executing code, such as CORBA [61], Java RMI [62], SOAP [33] and web services [50], generally offer only limited support for information security, consistency, and data shipping. J2EE persistence (EJB) [23] provides a limited form of transparent access to persistent objects, but does not address distrust or distributed computation. Peer-to-peer content-distribution and wide-area storage systems (e.g., [20, 42, 65, 68]) offer high data availability, but do little to ensure that data is neither leaked to nor damaged by untrusted users, nor do they en-

sure consistency of mutable data. Prior distributed systems that enforce confidentiality and integrity in the presence of distrusted nodes (e.g., [12, 80, 81]) have not supported consistent computations over persistent data.

Fabric integrates many ideas from prior work, including compile-time and run-time information flow, access control, peer-to-peer replication, and optimistic transactions. This unique integration makes possible a higher-level programming model that simplifies reasoning about security and consistency. Indeed, it does not seem possible to provide a high-level programming model like that of Fabric by simply layering previous distributed-systems abstractions. Several new ideas were also needed to make Fabric possible:

- A programming language that integrates information flow, persistence, transactions, and distributed computation.
- A *trust ordering* on information-flow labels, supporting reasoning about information flow in distributed systems.
- An integration of function shipping and data shipping that also enforces secure information flows within and among network nodes.
- A way to manage transactions distributed among mutually distrusting nodes, and to propagate object updates while enforcing confidentiality and integrity.

Fabric does not require application developers to abandon other standards and methodologies; it seems feasible for Fabric to interoperate with other standards. In fact, Fabric already interoperates with existing Java application code. It seems feasible to implement many existing abstractions (e.g., web services) using Fabric. Conversely, it seems feasible to implement Fabric nodes by encapsulating other services such as databases. We leave further work on interoperability to the future.

1.2.2 Referential security

Fabric provides confidentiality and integrity through a combination of information-flow control and access control. However, experience building Fabric applications has revealed security and reliability vulnerabilities resulting from object references.

References create security issues because they introduce dependencies between different parts of the system. We say that a system has *referential integrity* if a reference can be relied upon to continue pointing to the same object. This is both an availability and an integrity property. Referential integrity fails when an object is deleted while a reference to it still exists, resulting in a *dangling reference* (an availability failure), or when the reference points to a different object altogether (an integrity failure). Because Fabric does not address availability, Fabric programs can encounter dangling references.

Referential integrity appears in many guises. We use the term in a more general sense than in the database literature, where referential integrity is an important aspect of the relational model [17]. The Web is a system whose lack of referential integrity is well known: the referent of a hyperlink can be deleted, leading to the familiar "404" error. Referential integrity is also an important property for programming language design; in programming languages that lack referential integrity, such as C, dangling pointers are a serious problem. Today, many languages have automatic garbage collection, allowing the automatic reclamation of memory while preserving referential integrity.

While absolute referential integrity is desirable, it cannot be achieved in a federated system such as Fabric: referential integrity is necessarily limited by the trustworthiness of the node (or nodes) storing the referent object. Therefore, we generalize referential integrity to systems where nodes are partially trusted.

In a federated system, referential integrity must be balanced against other security and reliability properties. Indeed, violations of referential integrity are only the first of three *referential security* vulnerabilities considered in this dissertation. In a system with referential integrity, a reference to an object is a promise to the referrer that the object will not move or disappear. It must be persistent. Therefore, reachability implies persistence, as in various object-oriented databases (e.g., [4,8]) and also in marshalling mechanisms such as Java serialization. However, if all reachable objects are persistent, objects can become *accidentally persistent* because they are unexpectedly reachable. Accidental persistence can inflate resource consumption, leading to poor performance and system failure. This problem is familiar to programmers who have used Java serialization. Avoiding accidental persistence is our second goal.

The third goal is preventing what we call *storage attacks*. Referential integrity prevents discarding reachable objects. But this gives an adversary a means to mount a denial-of-service attack. The adversary creates references to objects intended to be discarded, preventing reclamation and perhaps exhausting available storage space.

In summary, the second part of this dissertation studies the problem of programming in distributed object systems while preventing three kinds of referential vulnerabilities: dangling pointers that violate referential integrity, accidental persistence that leaks storage, and storage attacks that consume resources. While recent work has explored programming models and languages for building federated systems (e.g., [42]), these referential vulnerabilities have not been clearly identified or addressed in prior work.

This dissertation formalizes referential vulnerabilities in terms of security properties corresponding to their absence. These *referential security properties* are

formalized in the context of a simple programming language that captures the key elements of distributed programming in a federated system with persistent information and pointers. A new type system is defined to enforce these security properties, ensuring that the system is secure and reliable. Because these properties can be viewed as integrity properties, the language must also enforce integrity in the information-flow sense. The correctness of security enforcement by the type system has been proved.

1.3 Dissertation outline

The rest of this dissertation is structured as follows. Chapter 2 presents the design and implementation of Fabric. Chapter 3 turns to the problem of referential security and presents a type system for its enforcement. Chapter 4 concludes.

The material in Chapter 2 is joint work with Michael George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew Myers, and is adapted from [48]. The work on referential security is joint work with Andrew Myers.

CHAPTER 2

FABRIC: SECURE FEDERATED COMPUTATION AND STORAGE

Fabric is a system and language for building secure federated information systems. It is a decentralized system that supports secure, shared access to information and computation, despite mutual distrust between cooperating entities. It has a high-level programming language that makes distribution and persistence largely transparent to programmers. Fabric supports both data shipping and function shipping: data and computation can move between nodes to meet security requirements or to improve performance. Fabric provides a rich, Java-like object model. Objects are labelled with confidentiality and integrity policies that are enforced through a combination of compile-time and run-time mechanisms. Optimistic, nested transactions ensure consistency across all objects and nodes. A peer-to-peer dissemination layer helps to increase availability and to balance load. Results from applications built using Fabric suggest that Fabric has a clean, concise programming model, offers good performance, and enforces security.

2.1 System architecture overview

Each Fabric node takes on one of the three roles depicted in Figure 2.1:

- *Storage nodes* (or *stores*) store objects persistently and provide object data when requested.
- *Worker nodes* perform computation, using both their own objects and possibly copies of objects from storage nodes or other worker nodes.
- *Dissemination nodes* provide copies of objects, giving worker nodes lower latency access and offloading work from storage nodes.

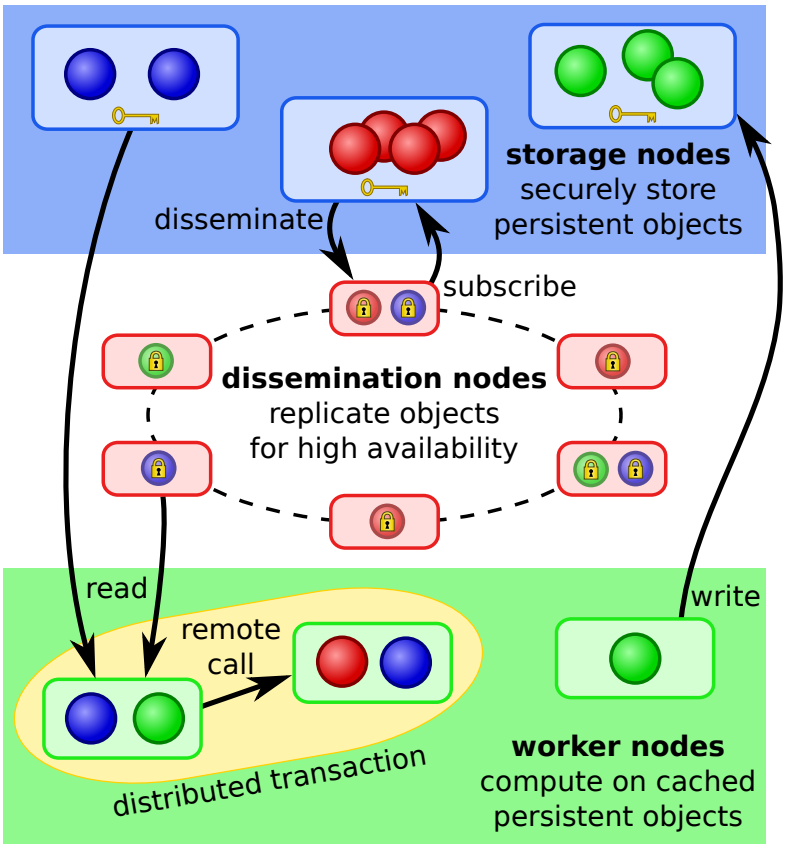


Figure 2.1: Fabric architecture

Although Fabric nodes serve these three distinct roles, a single host machine can have multiple Fabric nodes on it, typically colocated in the same Java VM. For example, a store typically has a colocated worker, allowing the store to invoke code at the worker with low overhead. This capability is useful, for example, when a store needs to evaluate a user-defined access-control policy to decide whether an object update is allowed (Section 2.3.5). It also gives the colocated worker the ability to efficiently execute queries against the store. Similarly, a worker node can be colocated with a dissemination node, making Fabric more scalable.

2.1.1 Security and assumptions

The design of Fabric is intended to allow secure sharing of computations and information, despite the presence of adversaries that control some Fabric nodes. The security goal of Fabric is the *decentralized security principle:* the security of a Fabric user should not depend on any part of the system that the user does not trust. (Equivalently, the security of a user should only depend on components of the system that the user trusts.)

Fabric users are able to express complete or partial trust in Fabric nodes. If a user expresses trust in a node, the compromise of that node might harm the security of that user. The goal of Fabric is to ensure that the degree of trust expressed bounds the degree to which security might be violated from the perspective of that user.

Assumptions

The security of any system depends on the assumptions made about the threats it is designed to defend against. The threat assumptions that Fabric makes are typical and weak, so the adversaries considered are typical, if not more powerful. This strengthens the security assurances of Fabric.

Compromised nodes are assumed to be malicious. They can give the outward appearance of a well-behaved node while behaving maliciously. Although Fabric provides a programming language with information-flow security, malicious behaviour is not constrained by the language. The runtime exposes more information than what is available at the language level, and malicious nodes can misuse the information they receive from the runtime, such as object identifiers, object version numbers, and cryptographic keys.

Misuse of information can include leaking confidential information or pro-

viding corrupt information to other nodes. However, without the appropriate cryptographic keys, nodes are assumed to be unable to learn encrypted content or forge digital signatures. As with most work on distributed systems, Fabric does not attempt to control read channels [78], timing channels, or termination channels.

Network adversaries are assumed to be unable to read or fabricate network messages on trusted channels. This assumption is justified by the use of SSL for all store–worker and worker–worker communication. Adversaries can learn information from the size, existence, and timing of network messages. As with most work on distributed systems, Fabric ignores these covert channels. Network adversaries can also prevent the delivery of messages. The availability of services written using Fabric depends on an assumption that network messages are eventually delivered.

2.1.2 Storage nodes

Storage nodes (stores) persistently store objects and, on request, provide copies of object data to worker nodes and dissemination nodes. Access control prevents nodes from obtaining data they should not see. The mechanism for this is described briefly here; details are given in Section 2.3.5.

Every object has an associated *label* describing the confidentiality and integrity requirements of the object's data (Section 2.2.3). When a worker requests a copy of an object from a store, the store examines the confidentiality part of the object's label. If the worker is trusted enough to read the object, then the store can securely send the worker an unencrypted copy of the object (though the network channel is of course encrypted by SSL).

This access-control mechanism works by treating each Fabric node as a prin-

cipal that tracks how much it trusts the nodes with which it interacts. Trust relationships are created by the delegation mechanisms described in Section 2.2.1.

After a worker fetches an object, it can perform computations using this cached copy, perhaps modifying its state. When the transaction containing these computations completes, the worker commits object updates to the stores that hold objects involved in the transaction. The transaction succeeds only if it is serializable with other transactions at those stores. As with object fetch requests, the store also enforces access control on update requests based on the degree of trust in the worker and the integrity policies in these objects' labels.

2.1.3 Worker nodes

Workers execute Fabric programs, which are typically written in the Fabric language. Programs may incorporate code written in other languages, such as the Fabric intermediate language, FabIL. However, such code is considered trusted: a worker running such code must trust it to maintain the security and consistency of the objects it uses. A worker only executes trusted code if it is stored on its local file system. In principle, *mobile code* (code provided by other nodes) can be downloaded and executed if the code is written in Fabric, and compiled and signed by a node that the worker trusts. The design described in this dissertation has been extended in [3] to add partial support for mobile code.

Fabric could, in principle, provide certifying compilation [58], allowing Fabric nodes to check that compiled code obeys the Fabric type system—and therefore that it correctly enforces access control and information-flow control—without relying on trusting the compiler or the node that runs it. The design and implementation of this feature are left to future work.

Fabric programs modify objects only inside transactions, which the Fabric

programming language exposes to the programmer as a simple `atomic` construct. Transactions can be nested, which is important for making code compositional. During transactions, object updates are logged in an undo/redo log, and are rolled back if the transaction fails. Such failures can happen because of inconsistency, deadlock, or an application-defined failure.

A Fabric program may be run entirely on a single worker that issues requests to stores (or to dissemination nodes) for objects that it needs. This *data-shipping* approach makes sense if the cost of moving data is small compared to the cost of computation, and if the objects' security policies permit the worker to compute using them.

When data shipping does not make sense, *function shipping* may be used instead. Execution of a Fabric program may be distributed across multiple workers, by using *remote method calls* to transfer control to other workers. Remote method calls in Fabric differ from related mechanisms such as Java RMI [62] or CORBA [61]:

- The *receiver object* on which the method is invoked need not be located at the remote worker (more precisely, cached at it) at the time of the call. In fact, the receiver object could be cached at the caller, at the callee, or at neither. Invocation causes the callee worker to cache a copy of the receiver object if it does not yet have a copy.
- The entire method call is executed in its own nested transaction. The effects of this transaction are not visible to other code running on the remote node until the commit of the top-level transaction containing the nested transaction. The commit protocol (Section 2.4.3) causes all workers participating in the top-level transaction to commit the sub-transactions that they executed as part of it.

- Remote method calls are subject to compile-time and run-time access-control checks. The caller side is checked at compile time to determine if the callee is trusted enough to enforce security for the method; the callee checks at run time that the calling node is trusted enough to invoke the method that is being called and to see the results of the method (Section 2.2.5).

Fabric workers are multithreaded and can concurrently serve requests from other workers. Pessimistic concurrency control (locking) is used to isolate transactions in different threads from each other.

One important use of remote calls is to invoke an operation on a worker colocated with a store. Since a colocated worker has low-cost access to persistent objects, this can improve performance substantially. This idea is analogous to a conventional application issuing a database query for low-cost access to persistent data. In Fabric, a remote call to a worker that is colocated with a store can be used to achieve this goal, with two advantages compared to database queries: the worker can run arbitrary Fabric code, and information-flow security is enforced.

2.1.4 Dissemination nodes

Objects are cached at *dissemination nodes*, to prevent stores with popular objects from becoming bottlenecks. Rather than requesting objects from remote or heavily loaded stores, workers can request objects from dissemination nodes. Objects are disseminated at the granularity of *object groups*, thereby amortizing the costs associated with fetching remote objects (Section 2.3.2).

On request, stores provide object data in encrypted form to dissemination nodes. Receiving encrypted objects does not require as much trust, because the

fields of the object are not visible without the object's encryption key, which dissemination nodes do not possess in general.

Fabric has no prescribed dissemination layer; workers may use any dissemination nodes they choose, and dissemination nodes may use whatever mechanism they want to find and provide objects. In the dissemination layer provided with the current Fabric implementation, the dissemination nodes form a peer-to-peer content distribution network based on FreePastry [69]. However, other dissemination architectures can be substituted if the interface to workers and stores remains the same.

To help keep caches up to date, workers and dissemination nodes are implicitly subscribed to any object group they read from a store. When any object in the group is updated, the store sends the updated group to its subscribers. The dissemination layer is responsible for relaying group updates to workers that have read them. A transaction that has read out-of-date data can then be aborted and retried by its worker on receipt of the updated group. Updated groups are delivered to their subscribers on a best-effort basis.

2.2 The Fabric language

Fabric offers a high-level language for building distributed programs with information-flow security. It is an extension to the Jif programming language [53,56], which also enforces secure information flow and has been used to build significant systems (e.g., [12, 14, 16, 36]). Fabric adds three major features to Jif:

- Nested transactions ensure that computations observe and update objects consistently, and provide clean recovery from failures.

- Remote method calls (remote procedure calls to methods) allow distributed computations that span multiple workers.
- Remote objects are accessed transparently, as if they are local objects.

While these features may seem unusual, they are not new.[1] The contribution of Fabric, however, is in combining these features with information-flow security. This requires new mechanisms to ensure that, for example, transactions do not leak confidential information, and remote calls are properly authorized. To support compile-time and run-time enforcement of secure distributed computation, Fabric also adds a new trust ordering on information-flow labels.

2.2.1 Principals

Principals in Fabric represent entities with authority, privilege, or trust. This includes users, roles, groups, organizations, privileges, and Fabric nodes. As in Jif [53], they are manifested in the Fabric programming language as objects with the built-in type `principal`.

Expressing trust: acts-for

Trust relationships between principals are represented by the *acts-for* relation [55]. If a principal p acts for principal q, any action by principal p can be considered to come from principal q as well. These actions include statements made by principal p. Thus, this acts-for relationship means q trusts p completely. We write this relationship more compactly as $p \succcurlyeq q$. The acts-for relation $\succcurlyeq$ is a pre-order (transitive and reflexive).

There is a most-trusted *top principal* $\top$ that acts for all other principals, and a least-trusted *bottom principal* $\bot$ that all principals act for. The operators $\wedge$ and $\vee$

[1] Argus [47] has the first two, for example.

can be used to form conjunctions and disjunctions of principals. The *conjunctive principal* $p \wedge q$ represents the joint authority of p and q, and acts for them both: $p \wedge q \succcurlyeq p$ and $p \wedge q \succcurlyeq q$. The *disjunctive principal* $p \vee q$ represents the disjoint authority of p and q. Both p and q act for the disjunctive principal $p \vee q$ (i.e., $p \succcurlyeq p \vee q$ and $q \succcurlyeq p \vee q$). The conjunctive and disjunctive principal operators are both commutative and associative.

Object representation of principals

The Fabric model of principals is similar to the model in Jif 3.0 [14]. Principals are represented as objects that inherit from the abstract class `fabric.lang.security.Principal`.[2] Instances of any subclass can be used as principals. Like other Fabric objects, principals can be distributed and persistent.

Principals control their acts-for relationships by implementing a method `p.delegatesTo(q)`, which tests whether `q` directly acts for `p`. This allows a principal to say who can directly act for it. The Fabric runtime system at each worker node automatically computes and stores the transitive closure of these direct acts-for relationships in a *security cache* (Section 2.3.8). The runtime system also exposes operations for notifying it that acts-for relationships have been added, which causes the acts-for cache to be updated conservatively to remove any information that might be stale. In general, worker nodes may have different partial views of the acts-for relation. The monotonicity of the label system ensures that security decisions based on these partial views are sound: any acts-for relationships that a worker has not observed would only make its security decisions more permissive [55].

[2]Instances of `fabric.lang.security.Principal` can be implicitly cast to the built-in type `principal`, and vice versa.

The Fabric runtime also supports the revocation of acts-for relationships. Supporting revocation involves a trade-off between security and performance (or availability), a challenge commonly encountered in the design of public-key infrastructures. To ensure sound authorization decisions, revocation notifications must rapidly propagate to all who might rely on the revoked authority. However, rapid propagation comes at a performance cost, and the propagation mechanism itself can be vulnerable to denial-of-service attack.

Fabric does not guarantee immediate notification of revoked acts-for relationships. A worker with a cached copy of principal `p` will not see a revocation of "`q` acts for `p`" until it receives an updated copy of `p`. This can happen through object subscriptions (Section 2.3.10). However, the subscription mechanism only operates on a best-effort basis. In the worst case, the worker will not receive the updated `p` until a transaction attempts to commit, after having observed the revoked acts-for relationship. As with any transaction that observes stale data, the transaction is rolled back and retried with the new version of `p`.

Node principals

Principals can use acts-for relationships to specify the degree to which they trust Fabric nodes. Fabric nodes are represented as first-class objects in Fabric that can be implicitly converted to principal objects that represent the node. For example, a storage node might be represented as a variable `s` of type `fabric.worker.Store`. The test `s actsfor p` would then test whether a principal `p` trusts `s`. This would always be the case if the principal `p` were stored at store `s`.

Fabric has a built-in way to authenticate worker nodes as corresponding to their Fabric worker-node objects. This is accomplished using X.509 certifi-

cates [37] that include the node's public key and the oid of its principal object (Section 2.3.4). Whether the certificates of a given certificate authority are accepted is decided by the Fabric node receiving them.

Authority

When running, Fabric code can possess the *authority* of a principal, and may carry out actions permitted to that principal, such as declassifying information to lower its confidentiality, or endorsing information to raise its integrity. As in Jif, code can obtain the authority of a principal p in two ways. In both cases, the code must be compiled and signed by a node that acts for p.

First, a class can declare it has authority by using a clause `authority(`p`)`. A method of the class can then claim this authority with a clause `where authority(`p`)`. Second, authority can be delegated via a method call, if the method being called is annotated with a clause `where caller(`p`)`. Such methods with delegated authority can only be called from code that possesses the authority of p. While this model of delegating authority has similarities to Java stack inspection [76], it differs in that authority is statically checked except at remote method calls, where the receiver checks that the calling node is sufficiently trusted (Section 2.2.5).

Principals express their security concerns by labelling information with information-flow policies.

2.2.2 Labels

Information security in Fabric is provided by information-flow control. All information is labelled with information-flow policies. These labels are propagated through computation using compile-time type checking, but run-time

checks are used for dynamic policies and to deal with untrusted nodes. There are two kinds of policies: confidentiality policies and integrity policies. While these policies are similar to those in Jif, they have a subtly different interpretation in Fabric because of the federated nature of the system.

Confidentiality and integrity policies

Confidentiality policies and integrity policies are built up from reader policies and writer policies, respectively. These, in turn, have two components: owners and subjects. The *owner* of a policy specifies the principal whose information is being governed by the policy. The *subject* specifies the principal who can act on (learn or affect) that information.

The *reader policy* `alice→bob`, for example, says that principal `alice` owns the policy and that she permits principal `bob` to indirectly learn about (or directly read) information that is labelled with the policy. If `bob` is trustworthy, however, he will not leak this information to untrusted third parties, because the information is not his to disclose. Therefore, `alice` is trusting `bob` to not inappropriately leak information labelled with the policy `alice→bob`.

Similarly, the *writer policy* `alice←bob` means that `alice` permits `bob` to indirectly affect (or directly modify) the labelled information; she is trusting `bob` to not inappropriately taint that information with data from untrusted sources.

Owners are implicitly subjects in their own policies. A principal is a *reader* for a reader policy $o \to r$ if the principal acts for $o \vee r$. A principal is a *writer* for a writer policy $o \leftarrow w$ if the principal acts for $o \vee w$. So, the policy `alice→bob` is equivalent to `alice→bob∨alice`, also written as `alice→bob,alice`. Similarly, `alice←bob` is equivalent to `alice←bob∨alice`, also written `alice←bob,alice`.

A *confidentiality policy* is a set of reader policies, all of which are enforced

simultaneously; a principal can learn about a value only when it is a reader for *all* reader policies in the value's confidentiality policy. An *integrity policy* is a set of writer policies; a principal can affect a value only when it is a writer for *any* writer policy in the value's integrity policy.

A *label* is simply a set of confidentiality and integrity policies, such as {`alice→bob;bob←alice`}. These *decentralized labels* [55] keep track of *whose* security is being enforced, which is useful for Fabric, where principals need to cooperate despite mutual distrust.

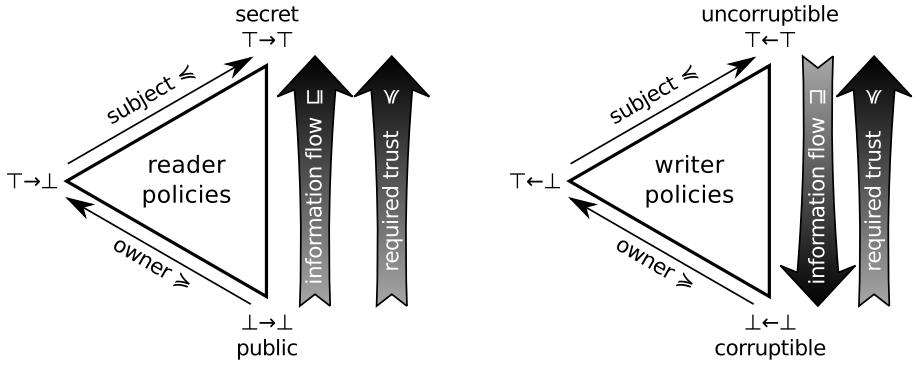
Declassification and endorsement

Information-flow security policies are expressed in terms of principals, which is important because it enables the integration of access control and information-flow control. A key use of this integration is for authorizing the downgrading of information-flow policies through declassification (for confidentiality) and endorsement (for integrity).

For example, on a worker `w` trusted by `alice` (i.e., `w` $\succcurlyeq$ `alice`), information labelled with the policy `alice→bob` can be explicitly declassified by code that is running with the authority of `alice`, removing that policy from the label.

Information-flow ordering

The Fabric compiler checks information flows at compile time to ensure that both explicit and implicit [25] information flows are secure. To do this, it uses Jif's *information-flow ordering* $\sqsubseteq$, which captures when information flow is secure: if $L_1 \sqsubseteq L_2$, then information labelled L_1 can securely flow to (or be relabelled with) L_2. The relationship $L_1 \sqsubseteq L_2$ can be read "L_1 flows to L_2".



(a) Reader policies (b) Writer policies

Figure 2.2: Orderings on reader and writer policies

Reader policies Figure 2.2a depicts how the acts-for relation and information-flow ordering relate for reader policies. Information flows upwards in the diagram. The information-flow ordering is covariant in both the owner and the subject. For example, we have $\{\texttt{alice} \to \texttt{bob}\} \sqsubseteq \{\texttt{charlie} \to \texttt{dora}\}$ exactly when $\texttt{charlie} \succcurlyeq \texttt{alice}$ (owner covariance) and $\texttt{dora} \succcurlyeq \texttt{bob} \vee \texttt{alice}$[3] (subject covariance).

As a value flows through a program, the set of principals that can declassify or read should only get smaller (unless the value is declassified or the principal hierarchy changes). Owner covariance is important because it ensures control of declassification is not lost. The new owner acts for the old owner, so any principal having the authority to declassify under the new policy also had that authority under the old policy. Subject covariance ensures information is not leaked; the new subject acts for the old subject, so any principal able to read under the new policy also had that ability under the old policy.

Therefore, the least reader policy, at the bottom of Figure 2.2a, describes information that is completely public: $\{\bot \to \bot\}$. The greatest reader policy, at the top of the figure, describes information that is completely secret: $\{\top \to \top\}$.

[3]The subject covariance condition is technically $\texttt{dora} \vee \texttt{charlie} \succcurlyeq \texttt{bob} \vee \texttt{alice}$, but the disjunction with $\texttt{dora}$ is omitted because it is satisfied by owner covariance.

Writer policies Integrity works the opposite way, because integrity policies allow flow from trusted sources to untrusted recipients. Integrity policies reflect the set of principals that could have endorsed or modified a value. To be safe, this should only get larger (unless the value is further endorsed or the principal hierarchy changes).

Figure 2.2b shows how the acts-for relation and information-flow ordering relate for writer policies. Information flows downwards in the diagram. The information-flow ordering is contravariant in both the owner and the subject. The relationship $\{\texttt{alice} \leftarrow \texttt{bob}\} \sqsubseteq \{\texttt{charlie} \leftarrow \texttt{dora}\}$ holds exactly when we have $\texttt{alice} \succcurlyeq \texttt{charlie}$ (owner contravariance) and $\texttt{bob} \succcurlyeq \texttt{dora} \vee \texttt{charlie}$[4] (subject contravariance).

Owner contravariance ensures those principals who have endorsed the labelled information are reflected in the new owner. The old owner acts for the new owner, so any principal who may have endorsed to the old policy could also have endorsed to the new policy. Subject contravariance ensures those principals who have affected the labelled information are reflected in the new subject. The old subject acts for the new subject, so any principal able to modify under the old policy also has that ability under the new policy.[5]

Therefore, the least writer policy, at the top of Figure 2.2b, describes information that is completely uncorrupted: $\{\top \leftarrow \top\}$. The greatest writer policy, at the bottom of the figure, describes information that is completely corrupted: $\{\bot \leftarrow \bot\}$. While this ordering is opposite of the intuitive ordering for integrity, we still refer to information labelled $\{\bot \leftarrow \bot\}$ as having *low integrity*, and information labelled $\{\top \leftarrow \top\}$ as having *high integrity*. Similarly, phrases such as *lower integrity* and *higher integrity* refer to the intuitive ordering.

[4]The subject contravariance condition is technically $\texttt{bob} \vee \texttt{alice} \succcurlyeq \texttt{dora} \vee \texttt{charlie}$, but the disjunction with $\texttt{alice}$ is omitted because it is satisfied by owner contravariance.

[5]See [54] for more justification of these rules.

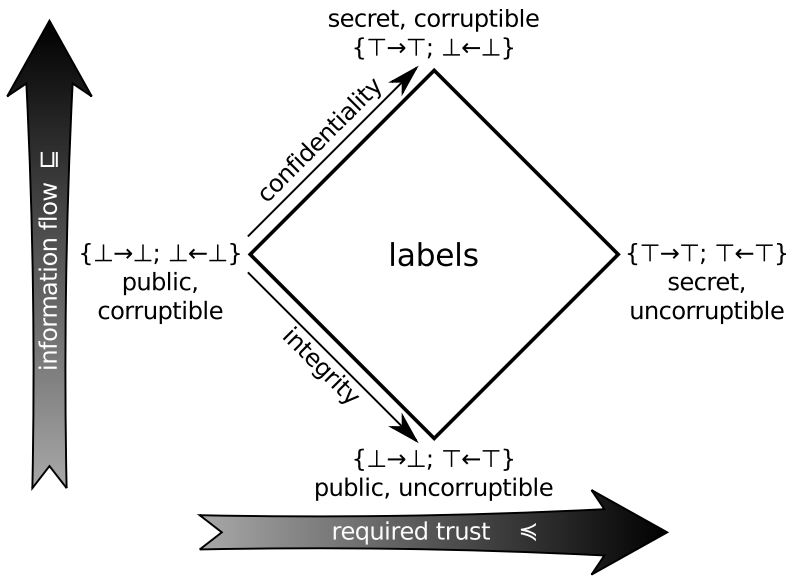


Figure 2.3: Orderings on the space of labels

Labels Information labelled L_1 can flow to another label L_2 if every reader policy in L_1 can flow to every reader policy in L_2, and every writer policy in L_1 can flow to every writer policy in L_2.

Figure 2.3 depicts the information-flow ordering on labels.[6] Information flows upwards in the diagram. The least label, at the bottom of the figure, describes information that can flow anywhere, because it is public and completely uncorrupted: $\{\bot \to \bot; \top \leftarrow \top\}$. The greatest label, at the top of the figure, describes information that can flow nowhere, because it is completely secret and completely corrupted: $\{\top \to \top; \bot \leftarrow \bot\}$.

Formally, the information-flow ordering is a pre-order over the set of labels. The equivalence classes of labels form a bounded join-semilattice, where lifted set unions are joins.[7] The equivalence classes of confidentiality policies form a sub-semilattice, as do the equivalence classes of integrity policies.

[6] Whereas Figure 2.2 shows each of the spaces of confidentiality and integrity policies in two dimensions, in Figure 2.3 they are each collapsed to one dimension for clarity.

[7] A meet operator can be defined as well to obtain a full lattice. See [54] for details.

```
1 int {alice→bob} x;
2 int {alice→bob, charlie} y;
3 x = y; // OK: bob ⊵ (bob ∨ charlie)
4 y = x; // Invalid
5 if (charlie actsfor bob) {
6   y = x; // OK: charlie ⊵ bob, so (bob ∨ charlie) ⊵ bob
7 }
```

Figure 2.4: Code example illustrating information-flow rules

Example The code in Figure 2.4 illustrates these rules. The assignment from `y` to `x` (line 3) is secure because the information in `y` can be learned by fewer readers (only `bob` rather than both `bob` and `charlie`). The assignment from `x` to `y` (line 4) is rejected by the compiler, because it permits `charlie` to read the information. However, the second assignment from `x` to `y` (line 6) is allowed because it occurs in a context where `charlie` is known to act for `bob`, and can therefore already read any information that `bob` can.

Trust ordering

Fabric extends the DLM by defining a second ordering on labels, the *trust ordering*, which is useful for reasoning about the enforcement of policies by a partially trusted platform. A label L_1 may require at least as much *trust* as a label L_2, which we write as $L_1 \succcurlyeq L_2$ by analogy with the trust ordering on principals. If L_1 requires at least as much trust as L_2, then any platform trusted to enforce L_1 is also trusted to enforce L_2. This happens when L_1 describes confidentiality and integrity policies that are at least as strong as those in L_2; unlike in the information-flow ordering, integrity is not opposite to confidentiality in the trust ordering.

Therefore, both confidentiality and integrity use the same rules in the trust ordering. As shown in Figure 2.2, they are covariant in both the owner and the subject; required trust increases upwards in the diagram. Both $\{\texttt{alice} \to \texttt{bob}\} \succcurlyeq$

$\{\texttt{charlie} \to \texttt{dora}\}$ and $\{\texttt{alice} \leftarrow \texttt{bob}\} \succcurlyeq \{\texttt{charlie} \leftarrow \texttt{dora}\}$ are true exactly when $\texttt{alice} \succcurlyeq \texttt{charlie}$ and $\texttt{bob} \succcurlyeq \texttt{dora} \vee \texttt{charlie}$.

Figure 2.3 depicts the trust ordering on labels, and how it relates to the information-flow ordering. Required trust increases rightwards in the diagram. In the trust ordering, the least label (leftmost in the figure) describes information that requires no trust to enforce its security, because it is completely public and completely corrupted: $\{\bot \to \bot; \bot \leftarrow \bot\}$. The greatest label (rightmost in the figure) is for information that is completely secret and completely uncorrupted: $\{\top \to \top; \top \leftarrow \top\}$.

2.2.3 Object labels

Like in Jif, each field in a Fabric object can have a different label that the compiler uses to control the flow of the information contained in that field. For efficiency, these field labels are summarized into a single *object label* that governs the use of information in that object at run time. This label determines which storage nodes can store the object persistently and which worker nodes can cache and compute directly on the object. It also controls which object groups an object may be part of and which key objects may be used to encrypt it for dissemination.

An object with label L_o may be stored securely on a store n if the store is trusted to enforce L_o. Recalling that n can be used as a principal, this condition is captured formally using the trust ordering on labels:

$$\{\top \to n; \top \leftarrow n\} \succcurlyeq L_o \tag{2.1}$$

To see this, suppose L_o has a confidentiality policy $\{p \to q\}$, which is equivalent to $\{p \to p \vee q\}$. Condition 2.1 holds exactly when $\top \succcurlyeq p$ (always true) and $n \succcurlyeq p \vee q$,

which implies $n \succcurlyeq p$ or $n \succcurlyeq q$—either p must trust n, or p must believe that n is allowed to read things that q is allowed to read. Conversely, if L_o has an integrity policy $\{p \leftarrow q\}$, we require the same condition, $n \succcurlyeq p \vee q$—either p trusts n, or p believes that n is allowed to affect things that q is. Therefore we can write $L(n)$ to denote the label corresponding to node n, which is $\{\top \rightarrow n; \top \leftarrow n\}$, and express condition 2.1 simply as $L(n) \succcurlyeq L_o$.

The object label is defined as the trust-ordering join of the object's field labels. This is a safe but conservative summary that can prohibit some secure flows at run time; however, per-field precision can be recovered by introducing an additional level of indirection in the object graph.

Fabric classes may be parameterized with respect to labels or principals, so different instances of the same class may have different labels. This feature, inherited from Jif, allows implementation of reusable classes, such as data structures that can hold information with different labels.

By design, Fabric does not provide *persistence by reachability* [4] because it can lead to unintended persistence. Therefore, constructors are annotated to indicate the store on which the newly created object should be made persistent. The call `new C@s(...)` creates a new object of class `C` on the store identified by the variable `s`. Except for reserving oids,[8] no communication with the store is needed until commit. If the store of an object is omitted, the new object is created at the same store as the object whose method calls `new`. Objects may have non-final fields that are marked `transient`. These transient fields are not saved persistently, which is similar to their treatment by Java serialization [32].

[8]This is done in batch ahead of time, so the communication cost for this can be amortized across many objects.

2.2.4 Tracking implicit flows

Information can be conveyed by program control flow. If not controlled, these *implicit flows* can allow adversaries to learn about confidential information from control flow, or to influence high-integrity information by affecting control flow.

Fabric controls implicit flows through the *program-counter label*, written pc, which captures the confidentiality and integrity of control flow. The program-counter label works by constraining side effects; to assign to a variable x with label L_x, Fabric requires $pc \sqsubseteq L_x$. If this condition does not hold, either information with a stronger confidentiality policy could leak into x, or information with a weaker integrity policy could affect x.

Implicit flows cross method-call boundaries, both local and remote. To track these flows, object methods are annotated with a *begin label* that constrains the program-counter label of the caller, as well as the effects of the method. The pc of the caller must flow to the begin label, which in turn must flow to the label of any variables assigned by the method. This ensures that the caller's pc can flow to the method's assignments. Implicit flows via exceptions and other control-flow mechanisms are also tracked [53].

Because implicit flows are controlled, untrusted code and untrusted data cannot affect high-integrity control flow unless an explicit downgrading action is taken, using the authority of the principals whose integrity policies are affected. Furthermore, because Fabric enforces robustness [13], untrusted code and untrusted data cannot affect information release. Thus, Fabric provides general protection against a wide range of security vulnerabilities.

```
1 void m1{alice←} () {
2   Worker rw = findWorker("bob.example.org");
3    if (rw actsfor bob) {
4      int{alice→bob} data = 1;
5      int{alice→} y = m2@rw(data);
6    }
7 }
8
9 int{alice→bob} m2{alice←} (int{alice→bob} x) {
10    return x+1;
11 }
```

Figure 2.5: A remote call in Fabric

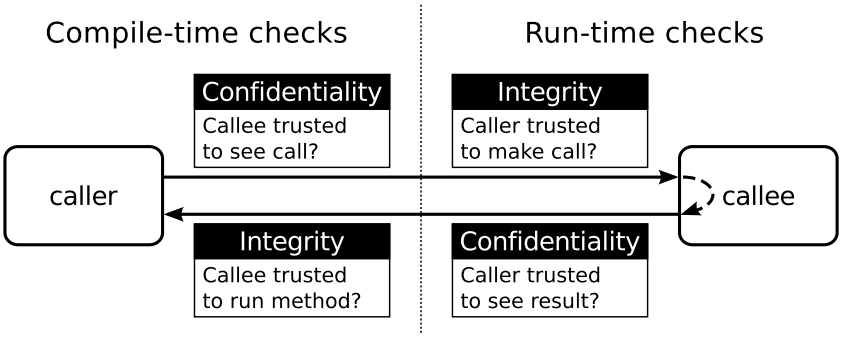


Figure 2.6: Compile-time and run-time checks for remote calls

2.2.5 Remote calls

Distributed control transfers are always explicit in Fabric. Fabric introduces the syntax `o.m@w(`a_1`,...,`a_n`)` to signify a remote method call to the worker node identified by variable `w`, invoking the method `m` of object `o`. If the syntax `@w` is omitted, the method call is always local, even if the object `o` is not cached on the current node (in which case, the object will be fetched and the method invoked locally). Figure 2.5 shows example code in which, at line 5, a method `m1` calls a method `m2` on the same object, but at a remote worker that is dynamically looked up using its hostname.

Remote method calls are subject to both compile-time and run-time checking. Figure 2.6 illustrates the checks made. The compiler permits a call to a re-

mote method only if it can statically determine that the call is secure, as shown on the left side of the figure. Information sent to a receiver worker `rw` can be read by `rw`, so all information sent in the call (the object, the arguments, and the implicit flow) must have labels L_s where $L_s \sqsubseteq \{\top \rightarrow \texttt{rw}\}$. For example, in Figure 2.5, the variable `data`, with label $\{\texttt{alice} \rightarrow \texttt{bob}\}$, can be passed to method `m2` only because the call happens in a context where it is known that $\texttt{rw} \succcurlyeq \texttt{bob}$, and hence $\{\texttt{alice} \rightarrow \texttt{bob}\} \sqsubseteq \{\top \rightarrow \texttt{rw}\}$.

Information received from `rw` can be affected by it, so by a similar argument, all information returned from the call must have labels L_r where $\{\top \leftarrow \texttt{rw}\} \sqsubseteq L_r$.

The recipient of a remote method call has no *a priori* knowledge that the caller is to be trusted, so run-time checking is needed. When a call occurs from caller worker `cw` to receiver worker `rw`, the receiver checks all information sent or received at label L (including implicit flows), to ensure that $\{\top \leftarrow \texttt{cw}\} \succcurlyeq L$. For example, in Figure 2.5, the method `m2` has a begin label that requires the integrity $\{\texttt{alice} \leftarrow\}$. Therefore, when `bob.example.org` receives the remote call to `m2`, it will check that the calling worker has the authority of `alice`, thereby ensuring $\{\top \leftarrow \texttt{cw}\} \succcurlyeq \{\texttt{alice} \leftarrow\}$. The compiler makes additional checks to ensure that these run-time checks themselves do not leak information.

2.2.6 Transactions

All changes to Fabric objects take place inside transactions, to provide concurrency control and ensure consistency of reads and writes. A transaction is indicated in Fabric code by the construct `atomic { `S` }`, where the transaction body S is a sequence of statements. The semantics is that the statement S is executed atomically and in isolation from all other computations in Fabric. In other words, Fabric enforces serializability of transactions.

Accesses to mutable fields of Fabric objects are not permitted outside transactions. Reads from objects that occur outside transactions are each treated as its own transaction.

If a transaction body throws an exception, the transaction is considered to have failed, and is aborted. If the body terminates successfully, its side effects become visible outside its transaction. Failure due to conflict with other transactions causes the atomic block to be retried automatically with exponential backoff. If the maximum number of retries is exceeded, the transaction is terminated.

Transactions may also be explicitly retried or aborted by the programmer. A `retry` statement rolls back the enclosing atomic block and restarts it from the beginning; an `abort` statement also rolls back the enclosing atomic block, but results in throwing the exception `UserAbortException`. Aborting a transaction creates an implicit flow; therefore, Fabric statically enforces that the pc of the `abort` is lower than or equal to the pc of the atomic block: $pc^{\texttt{abort}} \sqsubseteq pc^{\texttt{atomic}}$. Exceptions generated by the transaction body are checked similarly.

Atomic blocks may be used even during a transaction, because Fabric allows nested transactions. This allows programmers to enforce atomicity without worrying about whether their abstractions are at "top level" or not. Atomic blocks can also be used as a way to cleanly recover from application-defined failures, via `abort`.

Multi-worker computations (i.e., computations with remote calls) take place in atomic, isolated transactions that span all the workers involved. The Fabric runtime system ensures that when multiple workers use the same object within a transaction, updates to the object are propagated between them as necessary (Section 2.4.1).

Transactions are single-threaded; new threads cannot be started inside a

transaction, though a worker may run multiple transactions concurrently. This choice was made largely to simplify the implementation, though it maps well onto many of the applications for which Fabric is intended.

Fabric uses a mix of optimistic and pessimistic concurrency control. In the distributed setting, it is optimistic, because worker nodes compute on cached copies of objects that may be out of date, and a distributed two-phase commit protocol [31] ensures consistency at commit time. However, to coordinate threads running on the same worker, Fabric uses pessimistic concurrency control, in which threads acquire locks on objects.

Though distributed deadlocks may occur in Fabric, there are standard techniques (e.g, edge chasing [11]) for detecting and avoiding them in a non-federated context. We leave to future work the design and implementation of a secure deadlock-detection mechanism for a federated system like Fabric.

2.2.7 Java interoperability

Fabric programs can be written using a mixture of Java, Fabric, and FabIL (the Fabric intermediate language). FabIL is an extension to Java that supports transactions and remote calls, but not information-flow labels or static information-flow control. More concretely, FabIL supports the `atomic` construct and gives the ability to invoke methods and constructors with annotations `@w` and `@s` respectively. Transaction management is performed on Fabric and FabIL objects but not on Java objects, so the effects of failed transactions on Java objects are not rolled back.

FabIL and Java code is considered trusted, and workers only execute trusted code that is stored on their local file system. Therefore, the use of FabIL or Java code in Fabric programs offers lower assurance to principals who trust the

nodes running this code. This is compatible with the decentralized security principle, because the effects of trusted code is confined to these principals.

FabIL can be convenient for code whose security properties are not accurately captured by static information-flow analysis, making the labels of the full Fabric language counterproductive. An example is code implementing cryptography.

2.3 The Fabric runtime system

This section describes the features of the Fabric runtime system for supporting single-worker transactions. Section 2.4 extends these features with support for multi-worker transactions and remote calls.

2.3.1 Object model

Information in Fabric is stored in objects. Fabric objects are similar to Java objects; they are typically small and can be manipulated directly at the language level. Fabric also has array objects, to support larger data aggregates. Like Java objects, Fabric objects are mutable and are equipped with a notion of identity.

Naming

Objects are named throughout Fabric by object identifiers (*oids*). An object identifier has two parts: a store identifier, which is a fully qualified DNS hostname, and a 64-bit object number (*onum*), which identifies the object on that node. An object identifier can be transmitted through channels external to Fabric, by writing it as a uniform resource locator (URL) with the form `fab://`$store$`/`$onum$, where $store$ is a fully qualified DNS hostname and $onum$ is the object number.

An object identifier is permanent in the sense that it continues to refer to the same object for the lifetime of that object, and Fabric nodes always can use the identifier to find the object. If an object moves to a different store, acquiring an additional oid, the original oid still works because the original store is responsible for keeping a forwarding pointer in a *surrogate object*. Long forwarding chains of surrogate objects can reduce performance and reliability; path compression can be used to avoid this [22, 28, 38].

Knowing the oid of an object gives the power to name that object, but not the power to access it: oids are not capabilities [26]. If object names were capabilities, knowing the name of an object would confer the power to access any object reachable from it. To prevent covert channels that might arise because adversaries can see object identifiers, object numbers are generated by a cryptographically strong pseudo-random number generator. Therefore, an adversary cannot probe for the existence of a particular object, and an oid conveys no information other than the name of the node that persistently stores the object.

Fabric uses DNS to map hostnames to IP addresses, but relies on X.509 certificates to verify the identity of the named hosts and to establish secure SSL connections to them. Therefore, certificate authorities are the roots of trust and naming, as in the Web.

Fabric applications can implement their own naming schemes using Fabric objects. For example, a naming scheme based on directories and path names is easy to implement using a persistent hash map.

Labels

Every object has an associated object label that summarizes the confidentiality and integrity requirements associated with the object's data. It is used for

information-flow control and to control access to the object by Fabric nodes. The object label is defined as the trust-ordering join of the labels of the fields in the object's class. This join is computed by the compiler, where possible. However, some of this computation is performed by the Fabric runtime system. For example, if the compiler determines that an object label depends on a class parameter, which can happen when a field label uses a class parameter, then part of the object-label computation must be done at run time, when the object is constructed.

Classes

Every Fabric object, including array objects, contains a `ClassRef`, which is a reference to the object's class that is paired with the SHA-256 hash of the class's code. For classes stored in Fabric, the reference is the oid of a *class object*, a Fabric object that represents the object's class in the Fabric language and contains the class's code. For other classes, the reference is simply the Java fully qualified name of the class.

The `ClassRef` creates an unforgeable binding between each object and the correct code for implementing that object. When objects are received over the network, the expected hash in the object's `ClassRef` is checked against the actual hash of the object's class.

Versions

Fabric objects can be mutable. Each object has a *current version number*, which is incremented when a transaction that updates the object is committed. The version number distinguishes current and old versions of objects. If worker nodes try to compute with out-of-date object versions, the transaction will fail

on commit and will be retried with the current versions. The version number is an information channel with the same confidentiality and integrity as the fields of the object; therefore, it is protected by the same mechanisms.

2.3.2 Object groups

On a store, objects are associated with *object groups* containing a set of related objects. Object groups are the unit of object distribution: when an object is requested by a worker or dissemination node, the entire group is pre-fetched from the store, amortizing the cost of store operations over multiple objects. Every object in the object group is required to have the same security policy, so that the entire group can be treated uniformly with respect to access control, confidentiality, and integrity. The binding between an object and its group is not permanent; the store constructs object groups as needed and discards infrequently used object groups. To improve locality, the store tries to create object groups from objects connected in the object graph.

2.3.3 Dissemination and encryption

To avoid placing trust in the dissemination layer, disseminated object groups are encrypted using a symmetric key and signed with the public key of the originating store. The symmetric encryption key is stored in a *key object* that is not disseminated and must be fetched directly from its store. When an object group is fetched, the dissemination node sends the oid of the key object and the random initialization vector needed for decryption. Key objects are ordinarily shared across many disseminated object groups, so workers should not need to fetch them often.

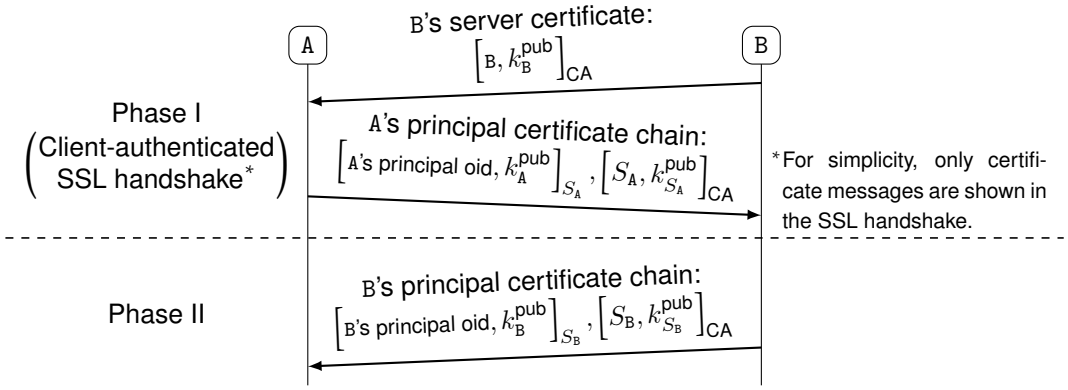


Figure 2.7: Authentication protocol sequence. Node A is connecting to node B.

Disseminated object groups are identified by dissemination nodes based on the oid of a contained object called the *head object*. The oid of the head object is exposed in the object group, but other oids in the object group (and the contents of all objects) are hidden by encryption.

2.3.4 Node authentication

Workers and stores authenticate each other when communicating over the network. The goal of authentication is to establish the principal oid (i.e., the oid of the principal object) of the remote host, so that it can be used in authorization checks. Figure 2.7 shows a protocol sequence diagram for the mutual authentication of nodes A and B. Authentication occurs in two phases.

In the first phase, A connects to B and performs a *client-authenticated* SSL handshake. Each node B that is capable of receiving connections has a *server certificate*, which is an X.509 certificate signed by a certificate authority (CA). This certificate binds B's DNS hostname to B's public key, and is therefore similar to an SSL certificate for the web. Whether the certificates of a given CA are accepted is decided by the Fabric node receiving them.

Every node A has a *principal certificate* for performing SSL client authentica-

tion. This is an X.509 certificate that binds A's principal oid to A's public key. The certificate is signed by the storage node that stores the principal object. With the store's server certificate, this gives a CA-rooted certificate chain for the oid of A's principal object.

After a successful SSL handshake, A knows it has contacted the correct node, because it knows the remote node has B's private key. Similarly, B knows A's principal oid, because of SSL client authentication, so A is authenticated to B.

In the second phase, B completes the mutual authentication by sending its principal certificate to A. This authenticates B if A can validate the certificate's signature, and can match the public key in the certificate with the one in B's server certificate.

Once the principal object of a remote node is established, it can be used in authorization checks.

2.3.5 Authorization checks

A Fabric node performs authorization checks to ensure confidentiality and integrity are maintained when sending or receiving data over the network. Stores perform authorization checks in a fresh top-level transaction on a colocated worker. When a worker w requests an object with label L, the store ensures that the worker is trusted to enforce the confidentiality part of L by checking $L \sqsubseteq \{\top \rightarrow \mathtt{w}\}$. Similarly, when the worker commits an update to the object, the store ensures that the worker is trusted to enforce the integrity part of L by checking $\{\top \leftarrow \mathtt{w}\} \sqsubseteq L$.

Workers perform analogous checks when receiving an object from a store, and when committing an object to a store. The authorization checks for remote calls are the run-time checks described in Section 2.2.5.

2.3.6 Transaction management and object locking

Every thread in the worker has a transaction manager that holds transaction state. The copy of each Fabric object at a worker has a reader-writer lock for isolating transactions in different threads from each other.

Each object also contains a version number that its store increments when it commits an update to the object. These are used to ensure consistency at commit time, as described below.

During computation, the transaction manager logs the version numbers of objects that are read or written, and the identities of the objects that are created. It acquires read locks for objects that are read, and write locks for objects that are written or created. The first write to an object during a transaction also logs the prior state of the object in an *object history* so that the transaction manager can restore the object's state in case the transaction aborts.

Because transactions can be nested, transaction logs and object histories are hierarchical. When a local sub-transaction is created, it inherits the locks held by its parent. When the sub-transaction commits, its log is merged with the parent transaction log, and its locks are transferred to the parent transaction. If the sub-transaction aborts, it discards its log, relinquishes the locks it has acquired, and restores the state of the objects it has modified.

To reduce logging overhead, the copy of each object at a worker has a *reader stamp*, which is a reference to the last transaction that read the object. No logging needs to be done for a read access if the current transaction matches the reader stamp. Similarly, each object has a *writer stamp* for the last transaction that modified the object, and no logging is needed if the current transaction matches the writer stamp. Obtaining the write lock clears the reader stamp.

When a worker commits a top-level transaction, it initiates a two-phase com-

mit protocol [31] with the stores for the objects accessed during the transaction. The information sent to each store includes the version numbers of objects accessed during the transaction and the new data for written objects. The store checks that its authoritative version numbers match the version numbers reported by the worker, to ensure the transaction used up-to-date information. For security, the store also performs the authorization checks described in Section 2.3.5 to ensure that the worker is trusted to modify the written objects.

2.3.7 Memory management

To conserve memory, cached objects may be *evicted* if they have no uncommitted changes. In the current implementation, cached objects are evicted automatically by the Java runtime system, because they are referenced using a Java `SoftReference` object. When an object is modified or created, its eviction is prevented by creating a hard reference to the object in the transaction log. This hard reference is destroyed when the transaction aborts or commits.

2.3.8 The security cache

The Fabric runtime system caches the results of acts-for tests and label comparisons. This memoization mechanism is adapted from Jif [14] and reduces the overhead of these dynamic tests.

The contents of the cache is the same as in Jif. Separate caches are kept for positive and negative results of acts-for tests and label comparisons. As with Jif, soundness is maintained by clearing the negative caches when a principal delegation is added, and by removing the positive-cache entries that depend on any principal delegation that is removed.

In Fabric, the security cache is tied to the transaction manager, to ensure that the use of the security cache does not introduce unsoundness. Each transaction has its own security cache, and because transactions can be nested, the security cache is hierarchical. When a sub-transaction is created, it inherits the cache entries from its parent. When the sub-transaction commits, its cache is merged with the parent cache; if the sub-transaction aborts, its cache is discarded. This ensures that the security cache of the parent transaction is isolated from changes to the principal hierarchy made in an aborted sub-transaction.

2.3.9 Handling failures of optimism

Computations on workers run transactions optimistically, which means that a transaction can fail in various ways. The worker has enough information to roll the transaction back safely in each case. At commit time, the system can detect inconsistencies that have arisen because another worker has updated an object that was accessed during the transaction. The stores inform the workers which objects involved in the transaction were out of date; the workers then flush their caches of the stale objects before retrying the transaction.

Another possible failure is that the objects read by the transaction are already inconsistent, breaking invariants on which the application code relies. Broken invariants can lead to errors in the execution of the application. Incorrectly computed results are not an issue because they will be detected and rolled back at commit time. Exceptions may also result, but as discussed earlier, exceptions also cause transaction failure and rollback. Finally, an application's computation might diverge rather than terminate. Fabric handles divergence by retrying transactions that are running too long. On retry, the transaction is given more time in case it is genuinely a long-running transaction. By geometrically grow-

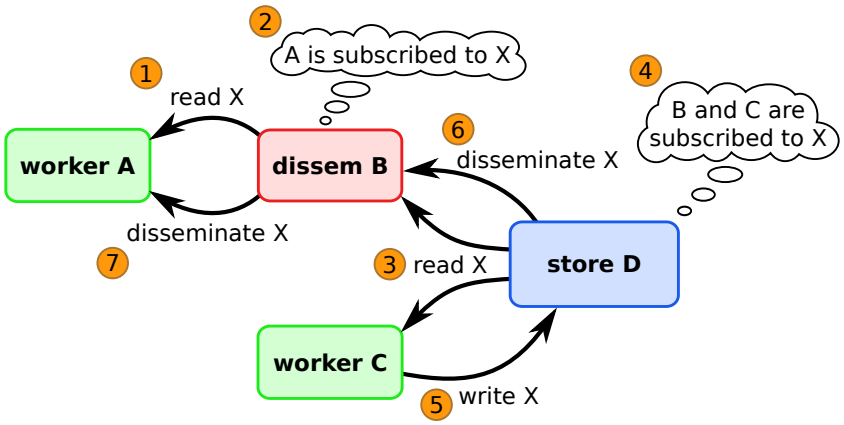


Figure 2.8: The object-subscription mechanism

ing the retry timeout, the expected running time is inflated by only a constant factor.

Because Fabric has subscription mechanisms for refreshing workers and dissemination nodes with updated objects, the object cache at a worker should tend to be up-to-date, and inconsistent computations in an application can be detected before a transaction completes.

2.3.10 Object subscriptions

To help keep caches up to date, workers and dissemination nodes are implicitly subscribed to any object group they read. Figure 2.8 illustrates this subscription mechanism. When a worker reads an object group from a dissemination node (1), it becomes subscribed to the group (2). Workers and dissemination nodes that read directly from a store (3) are similarly subscribed (4).

When any object in the group is updated (5), the store sends the updated group to its subscribers (6). The dissemination layer is responsible for relaying group updates to workers that have read them (7). Group updates are delivered on a best-effort basis. On receipt of the updated group, a worker can abort and retry any transaction that has read out-of-date data.

2.4 Support for distributed computation

Fabric transactions can be distributed across multiple workers by executing remote calls within a transaction. The whole transaction runs in isolation from other Fabric transactions, and its side effects are committed atomically. The ability to distribute transactions is crucial for reconciling expressiveness with security. Although some workers are not trusted enough to read or write some objects, it is secure for them to perform these updates by calling code on a sufficiently trusted worker. This section describes the features of the runtime system that support secure distributed transactions.

2.4.1 Writer maps

In a distributed transaction, an object can be shared and updated by multiple workers. This is challenging. For consistency, workers need to compute on the latest version of the shared object as it is updated. For performance, workers should be able to locally cache objects that are shared but not updated. For security, updates to an object with confidentiality L should not be learned by a worker $\mathtt{w}$ unless $L \sqsubseteq \{\top \to \mathtt{w}\}$. To allow workers to efficiently check for updates to objects they are caching, without revealing information to workers not trusted to learn about updates, Fabric introduces *writer maps*.

Every object used during a distributed transaction has a *writer*, which is the worker that last updated the object during the transaction. The object's writer, therefore, stores the definitive copy of the object for the transaction. Every transaction has a writer map that records the writer for each object used in the transaction. The writer map is passed through the distributed computation along with control flow.

When a worker $\mathtt{w}$ updates an object, the object's writer $\mathtt{w}'$ is notified and relinquishes the role. The notifying worker $\mathtt{w}$ becomes the new writer, and this change is recorded in the writer map. Notification of the old writer $\mathtt{w}'$ is not a covert channel, because the program-counter label pc of the write must be lower than the object's label L, which the old writer is already trusted to read:

$$pc \sqsubseteq L \sqsubseteq \{\top \to \mathtt{w}'\}$$

The writer map contains two kinds of mappings: writer mappings and label mappings. An update to object $\mathtt{o}$ at worker $\mathtt{w}$ adds a writer mapping with the form $hash(\mathtt{oid}, \mathtt{tid}, \mathtt{key}) \mapsto \{\mathtt{w}\}_{\mathtt{key}}$, where $\mathtt{oid}$ is the oid of object $\mathtt{o}$, $\mathtt{tid}$ is the identifier for the top-level transaction, and $\mathtt{key}$ is the encryption key for $\mathtt{o}$, stored in $\mathtt{o}$'s key object. This mapping permits a worker that has the ability to read or write $\mathtt{o}$—and therefore has the encryption key for $\mathtt{o}$—to learn whether there is a corresponding entry in the writer map, and to determine which node is currently the object's writer. Nodes lacking the key cannot exploit the writer mapping because without the key, they cannot verify the hash. Because the top-level transaction id is included in the hash, they also cannot watch for the appearance of the same writer mapping across multiple transactions.

Label mappings support object creation. The creation of a new object with oid $\mathtt{oid}$ adds an entry with the form $hash(\mathtt{oid}) \mapsto \mathtt{oid}_{\mathtt{label}}$, where $\mathtt{oid}_{\mathtt{label}}$ is the oid of the object's label, which contains the object's encryption key. This second kind of mapping allows a worker to find the encryption key for newly created objects, and then to check the writer map for a mapping of the first kind.

The writer map is an append-only structure, so if an untrusted worker fails to maintain a mapping, it can be restored. The size of the writer map is a side channel, but the capacity of this channel is bounded by always padding out the number of writer map entries added by each worker to the next largest power

of 2, introducing dummy entries containing random data as needed. Therefore a computation that modifies n objects leaks at most $\lg \lg n$ bits of information.

The writer map is threaded through the distributed computation along with control flow: it is included in every remote-call request, and is returned with the result of the call. Each worker in the computation keeps a local version number for the writer map, and increments this version number when incorporating new writer-map information from a remote-call request or a remote-call result.

During computation, while logging an object access, the transaction manager checks the writer map. If a writer is found, the latest version of the object is fetched from the writer, and for write accesses, the writer role is transferred.

To reduce overhead, the copy of each object at a worker has a *writer-map stamp*, which records the version number of the writer map seen during the previous access. No fetch needs to be done if the current writer-map version number matches the writer-map stamp.

2.4.2 Distributed transaction management

To maintain consistency, transaction management must in general span multiple workers. A worker maintains transaction logs for each top-level transaction it is involved in. These transaction logs must be stored on the workers where the logged actions occurred, because the logs may contain confidential information that other workers may not see. For example, in the code below, the existence of `a` or `b` in the transaction log can reveal the value of `secret`.

```
int x;
if (secret) x = a.f;
else x = b.f;
```

Figure 2.9 illustrates the log structures that could result during a distributed transaction involving three workers. Each transaction, including nested transactions, is identified by a randomly generated transaction id (tid). (For clarity, the figure uses sequential tids.) Each remote-call request includes the tids for the call's entire transactional context. This allows the receiving worker to synchronize its transaction state with that of the calling worker.

In Figure 2.9a, a transaction (`tid=01`) starts on worker A, then calls code on worker B, which starts a nested sub-transaction (`tid=02`) there. Because the request from worker A includes the context `ctxt=01`, worker B knows that `tid=02` occurs within `tid=01`. The code then calls to worker C, starting another sub-transaction (`tid=03`), which finally calls back to worker B, starting sub-transaction `tid=04`. Conceptually, all the transaction logs together form a single log that is distributed among the participating workers, as shown at the bottom of the figure.

When worker B returns to worker C, it commits `tid=04`, resulting in the state shown in Figure 2.9b. The procedure for committing a sub-transaction with a distributed transaction log is the same as for a transaction with a non-distributed log. To commit `tid=04`, worker B merges its portion of the log for `tid=04` with that of `tid=03`. Though worker C also has a portion of the log for `tid=03`, the two parts are kept separate.

Figure 2.9c shows the state of the distributed transaction log after worker C returns to worker B, and `tid=03` has committed. Before returning, worker C commits its portion of `tid=03`, so it merges its log for `tid=03` with that of `tid=02`.

When control returns from a remote call, the worker always commits up to the context in which the remote call occurred. Therefore, when worker B receives control, it also commits its portion of `tid=03` so it can continue working

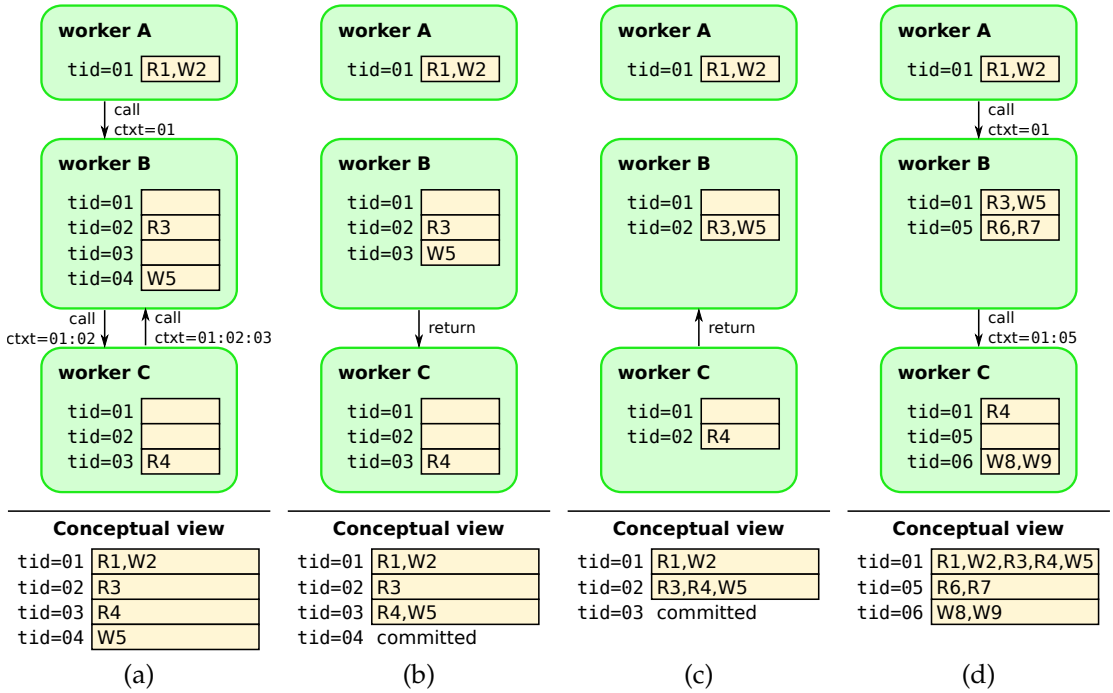


Figure 2.9: Distributed transaction logs

within `tid=02`. This merges worker B's portion of `tid=03` (which includes entries from `tid=04`) with `tid=02`.

The return from worker B to worker A is elided. The diagram looks like that in Figure 2.9c, except worker B has merged `tid=02` into `tid=01`. Although a similar merge does not happen at worker C, this is not a problem, because the merge will occur when worker C receives control again, or when the top-level transaction commits. Figure 2.9d shows the former case. Worker A calls worker B again, which starts `tid=05` and calls worker C, starting `tid=06`.

When a worker receives a remote call, it compares its transactional context with the one it receives. The worker synchronizes its transaction log by committing up to the most recent common ancestor of the two contexts, and starting any transactions it has missed. Therefore, when worker C receives the remote call, it compares its transactional context (`01:02`) with the one it receives (`01:05`).

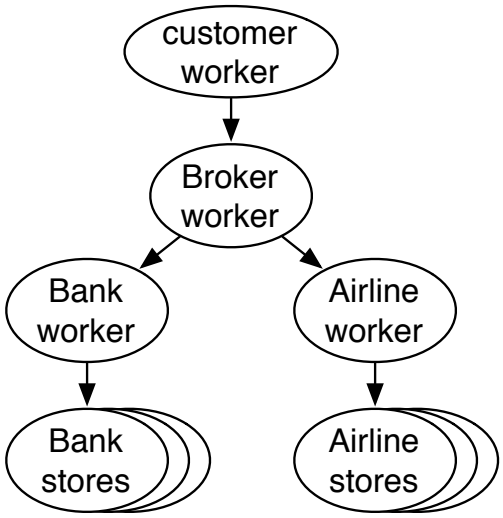


Figure 2.10: A hierarchical, distributed transaction

It commits up to the most recent common ancestor, `tid=01`, and starts the transaction it has missed, `tid=05`, before starting `tid=06` for the incoming call.

When the top-level transaction commits, workers A, B, and C participate in a hierarchical commit protocol to communicate with the stores of the objects they have accessed, using their respective parts of the logs.

2.4.3 Hierarchical commit protocol

In general, a transaction may span worker nodes that do not trust each other. This creates both integrity and confidentiality concerns. An untrusted node cannot be relied to commit its part of a transaction correctly. More subtly, the commit protocol might also cause an untrusted node to learn information it should not. Just learning the identities of other nodes that participated in a transaction could allow sensitive information to be inferred. Fabric's hierarchical two-phase commit protocol avoids these problems.

For example, consider a transaction that updates objects owned by a bank and other objects owned by an airline, perhaps as part of a transaction in which

a customer purchases an air ticket (see Figure 2.10). The bank and the airline do not necessarily trust each other; nor do they trust the customer purchasing the ticket. Therefore some computation is run on workers managed respectively by the bank and the airline. When the transaction is to be committed, some updates to persistent objects are recorded on these different workers.

Because the airline and the bank do not trust the customer, their workers will reject remote calls from the customer—the customer's worker lacks sufficient integrity. Therefore, this scenario requires the customer to find a trusted third party. As shown in the figure, a third-party broker can receive requests from the customer, and then invoke operations on the bank and airline. Because the broker runs at a higher integrity level than the customer that calls it, Fabric's endorsement mechanism must be used to boost integrity. This reflects a security policy that anyone is allowed to make requests of the broker. It is the responsibility of the broker to sanitize and check the customer request before endorsing it and proceeding with the transaction.

The hierarchical commit protocol begins with the worker that started the top-level transaction. It initiates commit by contacting all the stores for whose objects it is the current writer in the writer map, and all the other workers to which it has issued remote calls. These other workers then recursively do the same, constructing a *commit tree*. This process allows all the stores involved in a transaction to be informed about the transaction commit, without relying on untrusted workers to choose which workers and stores to contact and without revealing to workers which other workers and stores are involved in the transaction lower down in the commit tree. The two-phase commit protocol then proceeds as usual, except that messages are passed up and down the commit tree rather than directly between a single coordinator and the stores.

Of course, a worker in this tree could be compromised and fail to correctly carry out the protocol, causing some stores to be updated in a way that is inconsistent with other stores. However, a worker that could do this could already have introduced this inconsistency by simply failing to update some objects or by failing to issue some remote method calls. In our example above, the broker could cause payment to be rendered without a ticket being issued, but only by violating the trust that was placed in it by the bank and airline. The customer's power over the transaction is merely to prevent it from happening at all, which is not a security violation.

Once a transaction is prepared, it is important for the availability of the stores involved that the transaction is committed quickly. The transaction coordinator should remain available, and if it fails after the prepare phase, it must recover in a timely way. An unavailable transaction coordinator could become an availability problem for Fabric, and the availability of the coordinator is therefore a trust assumption. To prevent denial-of-service attacks, prepared transactions are timed out and aborted if the coordinator is unresponsive. In the example given, the broker can cause inconsistent commits by permanently failing after telling only the airline to commit, in which case the bank will abort its part of the transaction. This failure is considered a violation of trust, but in keeping with the security principles of Fabric, the failing coordinator can only affect the consistency of objects whose integrity it is trusted to enforce. This design weakens Fabric's consistency guarantees in a circumscribed way, in exchange for stronger availability guarantees.

2.5 Implementation

The Fabric implementation uses a mixture of Java, FabIL, and Fabric code. Not counting code ported to FabIL from earlier Java and Jif libraries, the implementation includes a total of 35k lines of code.

In addition to a common base of 7.7k lines of code supporting the worker, store, and dissemination nodes, the worker is implemented as 3.8k lines of Java code and 3.1k lines of FabIL code; the store is 1.9k lines of Java; and the dissemination layer is 1.2k lines of Java code. In addition, some of the GNU Classpath collection libraries have been ported to FabIL for use by Fabric programs (another 6.3k lines of code),

The Fabric compiler, supporting both Fabric and FabIL source files, is a 11k-line extension to the Jif 3.3 compiler [56], itself a 30k-line extension to the Polyglot compiler framework [59].

Implementing Fabric in Java has the advantage that it supports integration with and porting of legacy Java applications, and access to functionality available in Java libraries. However, it limits control over memory layout and prevents the use of many implementation techniques. In an ideal implementation, the virtual machine and JIT would be extended to support Fabric directly. For example, the Java `SoftReference` capability that is used for eviction could be implemented with fewer indirections. We leave VM extensions to future work.

2.5.1 Store

The current store implementation uses Berkeley DB [60] as a backing store in a simple way: each object is entered individually with its onum as its key and its serialized representation as the corresponding value. Because stores cache

both object groups and object versions in memory, and because workers are able to aggressively cache objects, the performance of this simple implementation is reasonable for the applications we have studied. For write-intensive workloads, object clustering at the backing store is likely to improve performance; we leave this to future work.

It is important for performance to keep the representation of an object at a store and on the wire compact. Therefore, references from one object to another are stored as onums rather than as full oids. A reference to an object located at another Fabric node is stored as an onum that is bound at that store to the full oid of the referenced object. This works well assuming most references are to an object in the same store.

2.5.2 Dissemination layer

The current dissemination layer is built using FreePastry [69], extended with proactive popularity-based replication based on Beehive [64]. The popularity-based replication algorithm replicates objects according to their popularity, with the aim of achieving a constant expected number of hops per lookup.

One standard configuration of Fabric worker nodes includes a colocated dissemination node to which dissemination-layer requests are directed; with this configuration, the size of the dissemination layer scales in the number of worker nodes.

2.5.3 Memory management

The current implementation uses Java's `SoftReference` feature for memory management. To achieve this, the FabIL compiler translates each class into a

pair of classes, a `_Proxy` class and an `_Impl` class, that follow the delegation design pattern. Conceptually, Fabric objects refer to each other through `_Proxy` instances. Each such instance has a `SoftReference` to its corresponding `_Impl`, and contains code that delegates to the `_Impl`. The `_Impl` class contains the actual class code, and its instances are the actual objects.

If an object has not been modified, its `_Impl` object will only be accessible through the `SoftReferences` in its `_Proxy` objects. This gives the JVM discretion to collect the `_Impl` when there is memory pressure. When an object is created or modified, a direct reference to its `_Impl` is added to transaction log, preventing the `_Impl` from being garbage-collected.

Figure 2.11 shows a FabIL class `C` and its Java translation. The class `C` has two fields `x` and `y`, and a method `m` that copies `y` to `x` and assigns a new `C` instance to `y`. The class is translated into an interface `C` with a pair of nested classes, `_Proxy` and `_Impl`, which implement the interface. The interface exposes a getter-setter pair for each field (lines 2–3), and a method corresponding to each method declared in the source class (line 4).

The `_Proxy` class (lines 6–8) extends the superclass's `_Proxy` class. All `_Proxy` classes ultimately inherit from `fabric.lang.Object._Proxy`, which holds the `SoftReference` to the corresponding `_Impl` object. Each `_Proxy` method delegates to the `_Impl` object. Line 7 shows the translation for the method `m`. It first obtains the `_Impl` object by calling `fetch()`. This will fetch the `_Impl` object if the JVM has evicted it from memory, or if the object has been updated by another node in the transaction. Once it has the `_Impl` object, the code then delegates to it by calling the appropriate method.

The `_Impl` class (lines 10–20) extends the superclass's `_Impl` class, and has the actual field data (line 11) and the actual implementations of `C`'s methods.

```
class C extends D implements I {
  C x,y;
  void m(Store s) { x = y; y = new C@s(); }
}
```

(a) FabIL class

```
1  interface C extends D, I {
2    C get$x(); C set$x(C val);
3    C get$y(); C set$y(C val);
4    void m(Store s);
5
6    static class _Proxy extends D._Proxy implements C {
7      void m(Store s) { ((C._Impl) fetch()).m(s); } ...
8    }
9
10   static class _Impl extends D._Impl implements C {
11     C x,y;
12     C get$x() {
13       TransactionManager.getInstance().registerRead(this);
14       return this.x;
15     }
16     void m(Store s) {
17       set$x(get$y());
18       set$y((C) new C._Impl(s).$getProxy());
19     } ...
20   } ...
21 }
```

(b) Java translation

Figure 2.11: A FabIL class and its Java translation

Field-accessor methods call into the transaction manager to register read/write operations (line 13). Field accesses are translated into calls to the appropriate accessor methods (line 17). New object instances are created by constructing an `_Impl` object, and immediately calling a `$getProxy` method to obtain its `_Proxy` (line 18).

2.5.4 Unimplemented features

Most of the Fabric design described in this dissertation has been implemented in the current prototype. A few features are not, though no difficulties are foreseen in implementing them: distributed deadlock detection via edge chasing [11], timeout-based abort of possibly divergent computations, timeout-based abort of prepared transactions for availability, subscriptions, `retry` and `abort` statements, path compression for pointer chains created by mobile objects, and avoidance of read channels at dissemination nodes.

2.6 Evaluation

2.6.1 Course Management System

To examine whether Fabric can be used to build real-world programs, and how its performance compares to common alternatives, we ported a portion of a course management system (CMS) [7] to FabIL. CMS is a 54k line J2EE web application written using EJB 2.0 [23], backed by an Oracle database. It has been used for course management at Cornell University since 2005; at present, it is used by more than 40 courses and more than 2,000 students.

Implementation CMS uses the model/view/controller design pattern; the model is implemented with Enterprise JavaBeans using Bean-Managed Persistence. For performance, hand-written SQL queries are used to implement lookup and update methods, while generated code manages object caches and database connections. The model contains 35 Bean classes encapsulating students, assignments, courses, and other abstractions. The view is implemented using Java Server Pages.

We ported CMS to FabIL in two phases. First, we replaced the Enterprise JavaBean infrastructure with a simple, non-persistent Java implementation based on the Collections API. We ported the entire data schema and partially implemented the query functionality of the model, focusing on the key application features. Of the 35 Bean classes, five have been fully ported. By replacing complex SQL queries with object-oriented code, we were able to simplify the model code a great deal: the five fully ported classes were reduced from 3,100 lines of code to 740 lines, while keeping the view and controller mostly unchanged. This intermediate version, which we will call the Java implementation, took one developer a month to complete and contains 23k lines of code.

Porting the Java implementation to FabIL required only superficial changes, such as replacing references to the Java Collections Framework with references to the corresponding Fabric classes, and adding label and store annotations. The FabIL version adds fewer than 50 lines of code to the Java implementation, and differs in fewer than 400 lines. The port was done in less than two weeks by an undergraduate initially unfamiliar with Fabric. These results suggest that porting web applications to Fabric is not difficult and results in shorter, simpler code.

A complete port of CMS to Fabric would have the benefit of federated, secure sharing of CMS data across different administrative domains, such as different universities, assuming that information is assigned labels in a fine-grained way. It would also permit secure access to CMS data from applications other than CMS. We leave this to future work.

Performance The performance of Fabric was evaluated by comparing five different implementations of CMS: the production CMS system based on EJB 2.0, the in-memory Java implementation (a best case), the FabIL implementation, the

	Page Latency (ms)		
	Course	Students	Update
EJB	305	485	473
Hilda	432	309	431
FabIL	35	91	191
FabIL/memory	35	57	87
Java	19	21	21

Table 2.1: CMS page load times (ms) under continuous load

FabIL implementation running with an in-memory store ("FabIL/memory"), and a fifth implementation developed earlier using the Hilda language [77]. Comparing against the Hilda implementation is useful because it is the best-performing prior version of CMS. The performance of each of these systems was measured for some representative user actions on a course containing 55 students: viewing the course overview page, viewing information about all students enrolled in the course, and updating the final grades for all students in the course. All three of these actions are compute- and data-intensive.

All Fabric and Java results were acquired with the app server on a 2.6 GHz single-core Intel Pentium 4 machine with 2 GB RAM. The Hilda and EJB results were acquired on slightly better hardware: the Hilda machine had the same CPU and 4 GB of memory; EJB results were acquired on the production configuration, a 3 GHz dual-core Intel Xeon with 8 GB RAM.

Table 2.1 shows the median time to perform three user actions under continuous load, for each of the measured systems. The first three measurements in Table 2.1 show that the Fabric implementation of CMS runs faster than the previous implementations of CMS. The comparison between the Java and non-persistent FabIL implementations illustrates that much of the run-time overhead of Fabric comes from transaction management and from communication with the remote store.

2.6.2 Travel example

Fabric can be used to build secure distributed applications, in which transactions span mutually distrusting workers. To evaluate this use, we built a simple prototype of the bank–airline example described in Section 2.4.3.

This application models an interaction between mutually distrusting users, banks, and merchants. Each principal has security concerns: banks are concerned that users and merchants only modify account balances in allowed ways, and users and merchants are concerned that their accounts are modified only when they decide to participate in a transaction.

These concerns are reflected in the labels placed on the objects implementing banks, accounts, users, and merchants. These labels restrict the possible placement of the data in the system. For example, a bank account is conceptually represented as follows:[9]

```
class Account[principal bank] {
  final Principal{bank←bank} user;
  int{bank→user; bank←bank} balance; ...
}
```

To satisfy the integrity policy, accounts must be stored at a bank machine.

Some operations in the example require trusted code. For example, the `createAccount` operation should be executable by any user, yet it must modify the list of accounts, which the bank considers to be high integrity. Implementing such operations requires information downgrading; static and runtime checks force such code to be explicitly approved by the affected principals, and to execute on workers trusted by those principals.

[9] In actuality, the account balance is broken out into a separate object, to prevent the confidentiality of the balance from tainting the account's object label.

	total	app	tx	log	fetch	store
Cold	9,153	10%	2%	12%	74%	2%
Warm	6,043	27%	3%	6%	61%	3%
Hot	840	46%	14%	24%	0%	17%

Table 2.2: Breakdown of OO7 traversal time (times in ms)

The application core is about 400 lines of Fabric code. Surrounding this core is another 1,000 lines of Fabric code to provide a web interface built on a Fabric port of SIF [14]. The labels and the trust relationships ensure its code and data are mapped securely onto the available nodes. Because of the mutual distrust in this example, transactions are committed using the hierarchical commit protocol described in Section 2.4.3.

2.6.3 Run-time overhead

To evaluate the overhead of Fabric computation at the worker when compared to ordinary computation on non-persistent objects, and to understand the effectiveness of object caching at both the store and the worker, we used the OO7 object-oriented database benchmark [9]. We measured the performance of a read-only (T1) traversal on an OO7 small database, which contains 153k objects totalling 24 MB. Performance was measured in three configurations: (1) cold; (2) warm, with stores caching object groups; and (3) hot, with both the store and worker caches warmed up.

Table 2.2 summarizes these measurements and breaks down the running times into time spent on application code (app), on local transaction processing (tx), on logging reads and writes (log), on fetching objects from the store (fetch), and on waiting for the store to process transaction messages (store).

The results show that caching is effective at both the worker and the store.

However the plain in-memory Java implementation of OO7 runs in 66 ms, which is about 10 times faster than the worker-side part of the hot traversal. Because Fabric is designed for computing on persistent data, this is an acceptable overhead for many, though not all, applications. For computations that require lower overhead, Fabric applications can always incorporate ordinary Java code, though that code must implement its own failure recovery.

2.7 Related work

Fabric provides a higher-level abstraction for programming distributed systems. Because it aims to help with many different issues, including persistence, consistency, security, and distributed computation, it overlaps with many systems that address a subset of these issues. However, none of these prior systems addresses all the issues tackled by Fabric.

OceanStore [66] shares the goal with Fabric of a federated, distributed object store. OceanStore is more focused on storage than on computation. It provides consistency only at the granularity of single objects, and does not help with consistent distributed computation. OceanStore focuses on achieving durability via replication. Fabric stores could be replicated but currently are not. Unlike OceanStore, Fabric provides a principled model for declaring and enforcing strong security properties in the presence of distrusted workers and stores.

Prior distributed systems that use language-based security to enforce strong confidentiality and integrity in the presence of distrusted participating nodes, such as Jif/split [79], SIF [14], and Swift [12], have had more limited goals. They do not allow new nodes to join the system, and they do not support consistent, distributed computations over shared persistent data. They do use program analysis to control read channels [79], which Fabric does not.

DStar [80] controls information flow in a distributed system using run-time taint tracking at the OS level, with Flume-style decentralized labels [41]. Like Fabric, DStar is a decentralized system that allows new nodes to join, but does not require certificate authorities. DStar has the advantage that it does not require language support, but controls information flow more coarsely. DStar does not support consistent distributed computations or data shipping.

Some previous distributed storage systems have used transactions to implement strong consistency guarantees, including Mneme [52], Thor [46] and Sinfonia [1]. Cache management in Fabric is inspired by that in Thor [10]. Fabric is also related to other systems that provide transparent access to persistent objects, such as ObjectStore [43] and GemStone [8]. These prior systems do not focus on security enforcement in the presence of distrusted nodes, and do not support consistent computations spanning multiple compute nodes.

Distributed computation systems with support for consistency, such as Argus [47] and Avalon [35], usually do not have a single-system view of persistent data and do not enforce information security. Emerald [6] gives a single-system view of a universe of objects while exposing location and mobility, but does not support transactions, data shipping or secure federation. InterWeave [74] synthesizes data- and function-shipping in a manner similar to Fabric, and allows multiple remote calls to be bound within a transaction, remaining atomic and isolated with respect to other transactions. However, InterWeave has no support for information security. The work of Shrira et al. [73] on exo-leases supports nested optimistic transactions in a client–server system with disconnected, multi-client transactions, but does not consider information security. MapJAX [57] provides an abstraction for sharing data structures between the client and server in web applications, but does not consider security. Other

recent language-based abstractions for distributed computing such as X10 [71] and Live Objects [63] also raise the abstraction level of distributed computing but do not support persistence or information-flow security.

Some distributed storage systems such as PAST [68], Shark [2], CFS [20], and Boxwood [49] use distributed data structures to provide scalable file systems, but offer weak consistency and security guarantees for distributed computation.

CHAPTER 3

DEFINING AND ENFORCING REFERENTIAL SECURITY

Referential integrity guarantees that named resources can be accessed when referenced. This an important property for reliability and security. In distributed systems, however, the attempt to provide referential integrity can itself lead to security vulnerabilities that are not currently well understood.

In this chapter, we identify three kinds of *referential security* vulnerabilities related to the referential integrity of distributed, persistent information. Security conditions corresponding to the absence of these vulnerabilities are formalized. A language model is used to capture the key aspects of programming distributed systems with named, persistent resources in the presence of an adversary. The referential security of distributed systems is proved to be enforced by a new type system.

3.1 Language model

3.1.1 Modelling distributed computing as a language

We model referentially secure distributed computing using a core programming language that we call $\lambda_{persist}$. One motivation for a language-based model is the popularity of high-level language-based models for distributed computing. Similarly to $\lambda_{persist}$, widely used middleware-based systems make distributed and persistent information appear to be ordinary language objects: they aim to make persistence and distribution more *transparent*. These systems include CORBA [61], Java RMI [62], and J2EE/EJB [23]. Fabric and various other research systems also take this approach (e.g., [5,6]). Prior work has also compiled high-level programs to distributed realizations. This approach has been devel-

oped for web applications (e.g., [12, 19, 72]) and for more general distributed applications (e.g., [27,78]). We expect our language model will help with understanding the security of all such language-based systems.

In addition, $\lambda_{persist}$ should give insight into referential security in systems that do *not* attempt to make persistence and distribution transparent, because $\lambda_{persist}$ faithfully models referential security in such systems as well.

In $\lambda_{persist}$, persistence, distribution, and communication are implicit but are constrained by policy annotations. Programs in $\lambda_{persist}$ are assumed to be mapped onto distributed host nodes in some way that agrees with these annotations. This mapping could be done manually by the programmer, or automatically by a compiler.

This implicit translation to a distributed implementation means that some apparently ordinary source-level operations may be implemented using distributed communication and computation, much in the same manner as in Fabric. For example, function application may be implemented as a remote procedure call. Similarly, following references at the language level may involve communication between nodes to fetch referenced objects.

Although the concrete mapping from source-level constructs onto host nodes is left implicit, we can nevertheless faithfully evaluate the security of source-level computations. The key is to ensure that the system is secure under *any* possible concrete mapping that is consistent with the policy annotations in the source program. That is, any given computation or information might be located on any host that satisfies the source-level security constraints. The technical contribution of this chapter is to develop an effective system of such source-level constraints, expressed as a type system.

Although we refer to $\lambda_{persist}$ as a source language, little attempt is made to

make this language congenial to actual programming. In particular, the type annotations introduced would be onerous in practice. They could be inferred automatically using standard techniques, but we leave this to future work. One can view the type system as describing a program (or system) analysis, and the formal results of this chapter as a demonstration that this analysis achieves its security goals.

3.1.2 Objects and references

Persistent objects are modelled in $\lambda_{persist}$ as records with mutable fields. The fields of an object can point to other objects through references. References contain the names of these mutable objects. References are not assignable as in ML [51]; imperative updates are achieved by assigning to mutable fields.

The language has two types of references: hard and soft. A *hard reference* is a reference with referential integrity: a promise that the referenced object will not be destroyed if its host is trustworthy. Because of this promise, hard references can only be created by trusted code. A *soft reference* does not create an obligation to maintain the referenced object. When following a soft reference or an untrusted hard reference, a program must be prepared to handle a failure in case the referenced object no longer exists. For example, a garbage collector may destroy objects reachable only via soft references. Hard links in Unix and references in Java are examples of hard references. URLs, Unix symbolic links, and Java `SoftReference` objects are examples of soft references.

This simple data model can represent many different kinds of systems, such as distributed objects, databases, and the Web. The shared directory structure shown in Figure 3.1 serves as a running example. Alice and Bob are travelling together and are using the system to share photos and itineraries. The root

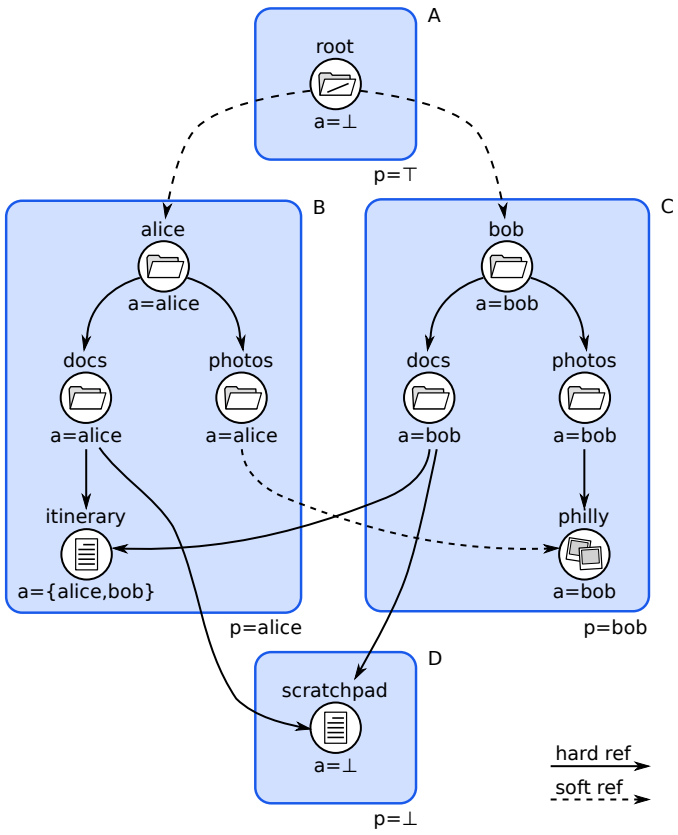


Figure 3.1: Directory example

directory is kept on a host `R`. Alice and Bob keep their directory objects on their own hosts, `A` and `B`, respectively. To share sightseeing ideas, they use a common scratchpad stored on host `U`. Solid arrows in the figure represent hard references, and dashed arrows are soft references. The `a` and `p` annotations are policies, which we now explain.

3.2 Policies for persistent programming

3.2.1 Persistence policies

Referential integrity ensures that a pointer can be followed to its referent—that there are no dangling pointers. In a federated system, referential integrity can-

not be absolute, because the referenced object may be located on an untrusted, perhaps maliciously controlled, host machine. Therefore, referential integrity must be constrained by the degree of trust in the referenced host. This constraint is expressed by assigning each object a *persistence policy* expressing how much it can be trusted to remain in existence.

The precise form of the persistence policy is left abstract. Persistence policies p are assumed to be drawn from a bounded lattice $(\mathcal{L}, \preccurlyeq)$ of *policy levels* with least element $\bot$ and greatest element $\top$. If $p_1 \preccurlyeq p_2$ for two persistence policies p_1 and p_2, then p_2 describes an object at least as persistent as that described by p_1.

While this description might seem to leave persistence policies too abstract to be meaningful, they have a simple, concrete interpretation. Absent replication, objects are located only on host nodes that are trusted to enforce their persistence policies, so a persistence policy p corresponds to a set of sufficiently trusted host nodes $H(p)$. Therefore, if $p_1 \preccurlyeq p_2$, then p_2 must be enforceable by a smaller set of hosts: $H(p_1) \supseteq H(p_2)$. In fact, it is reasonable to think of a policy p as simply a set of hosts.

For example, in Figure 3.1, the root directory has persistence policy $\top$ and is kept on host `R`, which is trusted to enforce this policy. Alice and Bob each have a user directory with their own persistence policy (`alice` and `bob`, respectively) and is stored on their own host (`A` and `B`, respectively). The shared scratchpad is kept on an untrusted host `U`, which enforces the persistence policy $\bot$.

Persistence policies are integrated into the type system of $\lambda_{persist}$. The type of an object reference includes a lower bound on the persistence policy of the object it refers to; the type system ensures that the persistence of the object is always at least as high as that of any reference pointing to it. Programs can therefore use the persistence of a reference to determine whether the reference

can be trusted to be intact. This rule enables sound reasoning about persistence and referential integrity as the graph of objects is traversed.

For example, in Figure 3.1, while Alice and Bob both have a hard reference to the scratchpad, they must be prepared for a persistence failure when using the reference. The type system of $\lambda_{persist}$ will ensure their code handles such a failure. Any reference to the scratchpad must have a type with $\perp$ persistence, because it can be no higher than the $\perp$ persistence of the scratchpad itself.

In $\lambda_{persist}$, persistence is defined not by reachability, but by policy. This resolves by fiat one of the three problems identified earlier: accidental persistence. Accidents are avoided by allowing programmers to express their intention explicitly. An object that is not intended to be persistent is prevented from being treated as a persistent object.

3.2.2 Characterizing the adversary

Security involves an adversary, and is always predicated on assumptions about the power of the adversary. In the kind of decentralized, federated system under consideration, the adversary is assumed to control some of the nodes in the system.

Different participants in a distributed system may have their own viewpoints about who the adversary is, yet all participants need security assurance. Therefore, a given adversary is modelled as a point α in the lattice of persistence policy levels. In the host-set interpretation of persistence policies, α defines the set of trusted hosts that the adversary does not control. If an object's persistence is not at least as high in the lattice as α, the adversary is assumed to have the power to delete (i.e., violate the persistence of) that object, because it is potentially stored at a host node controlled by the adversary.

The formal results for the security properties enforced by $\lambda_{persist}$ treat the adversary as an arbitrary parameter. Therefore, these properties hold for any adversary.

3.2.3 Storage attacks and authority policies

We introduce the idea of *storage attacks*, in which a malicious adversary tries to prevent reclamation of object storage by exploiting the enforcement of referential integrity. For example, in Figure 3.1, Bob has shared with Alice an album containing the photos he has so far taken during their trip. Bob doesn't consider the album to be private, so others may create references to his album, as Alice has done. However, an adversary that creates a hard reference to this album can prevent Bob from reclaiming its storage.

To prevent such storage attacks, we ensure that hard references can be created only in sufficiently trusted code. We introduce *creation authority* to abstractly define this power to create new references. This is the only action requiring some form of authority in our discussion, so for brevity, we refer to creation authority simply as *authority*.

Like persistence policies, authority policies a are assumed to be drawn from a bounded lattice $(\mathcal{L}, \preccurlyeq)$ of policy levels. Without loss of expressive power, they are assumed to be drawn from the same lattice as persistence policies. Authority prevents storage attacks because hard references can only be created to objects whose authority policy a is less than or equal to the authority of the process a_p. That is, we require $a \preccurlyeq a_p$.

A hard reference is a reference that should have referential integrity, so creating hard references requires authority. The adversary is assumed to have some ability to create hard references, described by its authority level α. Soft refer-

ences do not keep an object alive, so no creation authority is required to create a soft reference.

In Figure 3.1, the root directory has the authority policy $\bot$, so anyone can create a hard reference to it. Bob's `philly` album is large, so he has given it the authority policy `bob`; only he can create hard references that prevent the album from being deleted. Therefore, Alice's reference to the album must be soft. Alice has drafted an itinerary, giving it the authority policy $\{\mathsf{alice}, \mathsf{bob}\}$ to indicate she will persist the document for as long as Bob requires. Bob's reference to the itinerary, therefore, can be hard.

It may sound odd to posit control over creation of references. But a reference with referential integrity is a contract between the referrer and the referent. For example, the node containing the referent is obligated to notify the referrer if the object moves. Entering into a contract requires agreement by both parties, so it is reasonable for the node containing the referent to refuse the creation of a reference.

3.2.4 Integrity

Thus far, the enumerated powers of the adversary include creating references to low-authority objects and destroying objects with low persistence. Because the adversary may control some nodes, the adversary can also change the state of objects located at these nodes. This may in turn affect code running on nodes not controlled by the adversary, if the adversary supplies inputs to that code, or if the adversary affects the decision to run that code.

Integrity levels describe limitations on these effects of the adversary. An integrity level w is drawn from a bounded lattice $(\mathcal{L}, \preccurlyeq)$ of policy levels; without loss of expressive power, it is assumed to be the same lattice as for persistence

and authority policies. The ordering $\preccurlyeq$ corresponds to increasing integrity. If $w_1 \preccurlyeq w_2$, an information flow from level w_2 to w_1 would be secure: more-trusted information would be affecting less-trusted information.[1]

In $\lambda_{persist}$, each variable and each field of an object has an associated integrity level describing how trusted it is, and hence how powerful an adversary must be to damage it. The integrity of a reference is the integrity of the field or variable it was read from. Each object also has a persistence and an authority level, which we can think of as the integrity of other, implicit attributes of the object. For persistence, this implicit attribute is the existence of the object itself. For authority, the attribute is the set of incoming references to the object. This unifying view of different policies as different aspects of integrity explains why all three kinds of policies can come from the same lattice.

The interpretation of these policies for integrity, authority, and persistence is summarized in Figure 3.3.

3.2.5 Integrity of dereferences and garbage collection

An adversary can directly affect the result of a dereference in two ways. First, if the reference has low integrity, the adversary can alter it to point to a different object. Second, if the referent has low persistence, the adversary can delete it. Therefore, the integrity of any dereference can be no higher than the integrity and persistence annotations on the reference. So, in Figure 3.1, if Alice follows the reference from her `docs` directory to the `scratchpad`, she obtains an untrusted result; the untrusted host `U` influences the result by choosing whether to delete the `scratchpad` object.

More subtly, the adversary can manipulate hard references to influence the

[1] This ordering corresponds to the trust ordering described in Section 2.2.2.

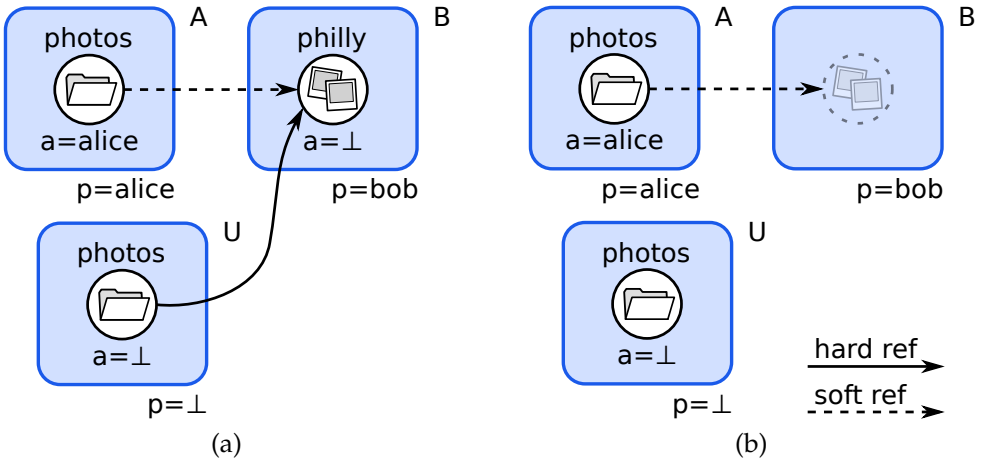


Figure 3.2: Authority affects integrity of dereferences. Alice is following her soft reference to the philly album. An adversary can affect the outcome of the dereference, because the album has low authority. (a) The untrusted host U has a hard reference preventing philly from being garbage collected; Alice's dereference succeeds. (b) Host U has removed its hard reference, allowing philly to be garbage collected; Alice's dereference fails.

garbage collector, and thereby *indirectly* affect the result of a dereference. For example, in Figure 3.2a, Alice is following her soft reference to Bob's `philly` album. Bob has marked `philly` as only requiring low authority, allowing the untrusted, adversarial host `U` to create a hard reference, and thereby preventing `philly` from being garbage-collected. Therefore, Alice's dereference must succeed.

However, in Figure 3.2b, the adversary `U` has removed its reference. Subsequently, `philly` has been garbage-collected, and Alice's dereference fails. The adversary has indirectly affected the outcome of the dereference. To account for this, the integrity of Alice's dereference must be no higher than the authority required by `philly`.

	Integrity	Authority	Persistence	Set of hosts
$\top$ "High"	Trusted, Untainted: No one can affect data	"root": No one can make a hard reference	Persistent: No one can delete object	No host nodes
$\bot$ "Low"	Untrusted, Tainted: Anyone can affect data	"anyone": Anyone can make a hard reference	Transient: Anyone can delete object	All host nodes

Figure 3.3: Interpretations of the extremal policy labels

Variables	$x, y \in \mathsf{Var}$	Policy levels	$w, a, p, \ell \in \mathcal{L}$
Memory locations	$m \in \mathsf{Mem}$	PC labels	$pc ::= w$
Labelled record types	$S ::= \{\overrightarrow{x_i : \tau_i}\}_s$	Storage labels	$s ::= (a, p)$
Labelled reference types	$R ::= \{\overrightarrow{x_i : \tau_i}\}_r$	Reference labels	$r ::= (a^+, a^-, p)$
Base types	$b ::= \mathsf{bool} \mid \tau_1 \xrightarrow{pc} \tau_2 \mid R \mid \mathsf{soft}\ R$	Types	$\tau ::= b_w \mid \mathbf{1}$
Values	$v, u ::= x \mid \mathsf{true} \mid \mathsf{false} \mid * \mid m^S \mid \mathsf{soft}\ m^S \mid \lambda(x : \tau)[pc].\, e\ (\mid \bot_p)$		
Terms	$e ::= v \mid v_1\ v_2 \mid \mathsf{if}\ v_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \{\overrightarrow{x_i = v_i}\}^S \mid v.x$		
	$\mid v_1.x := v_2 \mid \mathsf{soft}\ e \mid e_1 \| e_2 \mid \mathsf{exists}\ v\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2$		
	$\mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$		

Figure 3.4: Syntax of $\lambda^0_{persist}$

3.2.6 Security properties

We now informally summarize how $\lambda_{persist}$ prevents the three referential vulnerabilities discussed earlier: accidental persistence, referential integrity, and storage attacks. The adversary is described by a single point in the lattice of policy levels, describing the adversary's ability to destroy, modify, and link to objects. Accidental persistence is prevented because persistence is determined by policies expressing the programmer's intent, rather than by reachability. Referential integrity is maintained by a $\lambda_{persist}$ program with respect to a particular adversary if following hard references whose persistence and integrity are above the level of the adversary never leads to an object that has been destroyed by the adversary or garbage-collected. Storage attacks are prevented if the adversary is unable to change the set of high-authority objects that are reachable through hard references.

3.3 Types for persistent programming

To formalize the ideas introduced in the previous section, we introduce the $\lambda_{persist}$ language, an extension to the simply typed lambda calculus. Its type system prevents the referential vulnerabilities identified above. Figure 3.4 gives the formal syntax of $\lambda_{persist}$. We first focus on how it integrates policies for persistence, authority, and integrity into its types.

3.3.1 Labels

We assume a bounded lattice $(\mathcal{L}, \preccurlyeq)$ of *policy levels* with least element $\bot$ and greatest element $\top$, from which integrity (w), authority (a), and persistence policies (p) are drawn.

Objects and reference values are annotated with *storage labels* consisting of a creation authority policy and a persistence policy. All non-unit types τ consist of a base type b along with an integrity policy annotation w; fields and variables thereby acquire integrity policies, because they are part of their types. Objects do not have their own integrity labels because all of their state is in their fields, which do.

The program-counter label pc [24] is an integrity level indicating the degree to which the program's control flow has been tainted by untrusted data. This label restricts the side effects of code.

3.3.2 Example

Suppose we want to create a hierarchical, distributed directory structure, such as in Figure 3.1. Each directory maps names to either strings, representing ordinary files, or to other directories, and contains a reference to its parent direc-

tory (elided in the figure). To faithfully model ordinary file systems, directories higher in the hierarchy should be more persistent: if they are destroyed, so is everything below.

A fully general directory structure would require augmenting $\lambda_{persist}$ with recursive and dependent types; for simplicity, these features have been omitted from $\lambda_{persist}$ because they do not appear to add interesting issues. However, we can capture the security of a general directory structure by using $\lambda_{persist}$ records to build a fixed-depth directory structure with a fixed set of entry names for each directory.

3.3.3 Modelling objects and references

The security policies of $\lambda_{persist}$ are about objects and references to them. Therefore, $\lambda_{persist}$ extends the lambda calculus with records that represent the content of objects. The record $\{\overrightarrow{x_i = v_i}\}$ comprises a set of fields x_i with corresponding values v_i. Records are not values in the language; instead, they are accessed via references m^S, where m is the identity of the object and $S = \{\overrightarrow{x_i : \tau_i}\}_s$ gives its base type. The *storage label* s is a pair (a, p). The *authority label* a is an upper bound on the authority required to create a new reference to the referent object.

References to objects have labelled reference types $\{\overrightarrow{x_i : \tau_i}\}_r$. A reference label r is a triple (a^+, a^-, p) that gives upper and lower bounds on the authority required by the referent, and a lower bound on the persistence of the referent. The *upper authority label* a^+ prevents storage attacks. The *lower authority label* a^- prevents the adversary from exploiting garbage collection to damage integrity.

3.3.4 Modelling distributed systems

The goal of the $\lambda_{persist}$ language is to model a distributed system in which code is running at different host nodes. A single program written in $\lambda_{persist}$ is intended to represent such a system. The key to modelling distributed, federated computation faithfully is that different parts of the program can be annotated with different integrity labels, representing the trust that has been placed in that part of the code. To model a set of computations (subprograms $\overrightarrow{e_i}$) executing at different nodes, the individual computations are composed in parallel ($e_1 \| \cdots \| e_n$) into a single $\lambda_{persist}$ program.

From the viewpoint of a given principal in the system, code with a low integrity label, relative to that principal, can be replaced by any code at all. For the purposes of evaluating the security of the system, this code is in effect erased and replaced by the adversary. Therefore the single-program representation faithfully models a distributed system containing an adversary.

3.4 Accidental persistence and storage attacks

We present $\lambda_{persist}$ in two phases. In this section, we present $\lambda^0_{persist}$, a simplified subset of $\lambda_{persist}$ that prevents accidental persistence and storage attacks.

3.4.1 Syntax of $\lambda^0_{persist}$

Figure 3.4 gives the syntax of $\lambda^0_{persist}$. The names x and y range over variable names Var; m ranges over a space of memory addresses Mem; w, a, p, and ℓ range over the lattice $\mathcal{L}$ of policy levels; s ranges over the space of storage labels $\mathcal{L}^2$; and r ranges over the space of reference labels $\mathcal{L}^3$.

Types in $\lambda^0_{persist}$ consist of base types with an integrity label (b_w), and also

the unit type **1**, which needs no integrity label. Base types include booleans, functions, and two kinds of references to mutable records: hard (R) and soft (soft R). The metavariable R denotes a labelled reference type.

The type $\tau_1 \xrightarrow{pc} \tau_2$ is an ordinary function type with a pc annotation that is a lower bound on the pc label of the caller. It gives an upper bound on the integrity of data the function affects, on the authority level of references the function creates, and on the authority level of any references in the function body.

Values include variables x, booleans true and false, the unit value $*$, typed memory locations (references) m^S, soft references soft m^S, functions $\lambda(x : \tau)[pc].\, e$, and p-persistence failures $\bot_p$. A function $\lambda(x : \tau)[pc].\, e$ has one argument x with type τ. The pc component has the same meaning as that in function types.

Terms include values v and u, applications $v_1\ v_2$, if expressions if v_1 then e_2 else e_3, record constructors $\{\overrightarrow{x_i = v_i}\}^S$, field selections $v.x$, field assignments $v_1.x := v_2$, soft references soft e, parallel composition $e_1 \| e_2$, soft-reference tests exists v as x : e_1 else e_2, and let expressions let $x = e_1$ in e_2.

3.4.2 Example

Returning to the directory example in Figure 3.1, Bob can add to the itinerary with the code below. It starts at the root of the directory structure, traverses down to the itinerary, and invokes an `add` method to add a museum.

```
let home = root.bob
in exists home as bob:
     let docs = bob.docs
         itin = docs.itinerary
     in itin.add "Rodin Museum"
   else: …
```

The soft reference `home` to Bob's home directory may have been snapped by the garbage collector, so `exists` is used to determine whether the reference is still valid. If so, the body of the `exists` is evaluated with `bob` bound to a hard reference to the home directory. (This reference can be created because the pc label at this point has sufficient creation authority.) The second select expression, `bob.docs`, dereferences the hard reference.

3.4.3 Operational semantics of $\lambda^0_{persist}$

The small-step operational semantics of $\lambda^0_{persist}$ is given in Figure 3.5. The notation $e\{v/x\}$ denotes capture-avoiding substitution of value v for variable x in expression e. The value of a memory location that has failed or been garbage-collected is $\perp$.

Let M represent a memory that is a finite partial map from typed memory locations m^S to closed record values, and let $\langle e, M\rangle$ be a system configuration. A small evaluation step is a transition from $\langle e, M\rangle$ to another configuration $\langle e', M'\rangle$, written $\langle e, M\rangle \rightarrow \langle e', M'\rangle$.

To avoid using undefined memory locations, we restrict the form of $\langle e, M\rangle$. Let $\mathsf{locs}(e)$ represent the set of locations appearing explicitly in e. A memory M is well-formed only if every address m appears at most once in $\mathsf{dom}(M)$, and for any location m^S in $\mathsf{dom}(M)$, $\mathsf{locs}(M(m^S)) \subseteq \mathsf{dom}(M)$. A configuration $\langle e, M\rangle$ is well-formed only if M is well-formed, $\mathsf{locs}(e) \subseteq \mathsf{dom}(M)$, and e has no free variables. Evaluation preserves well-formed configurations (see Lemma 10 in Section 3.7.3).

Most of the operational semantics rules are straightforward, but a few deserve more explanation.

The record constructor $\{\overrightarrow{x_i = v_i}\}^S$ (rule CREATE) creates a new memory lo-

$$[\textsc{Apply}] \quad \langle (\lambda(x{:}\tau)[pc].\, e)\ v, M\rangle \xrightarrow{\mathsf{e}} \langle e\{v/x\}, M\rangle$$

$$[\textsc{Let}] \quad \frac{\forall p.\ v \neq \bot_p}{\langle \mathsf{let}\ x = v\ \mathsf{in}\ e, M\rangle \xrightarrow{\mathsf{e}} \langle e\{v/x\}, M\rangle}$$

$$[\textsc{If-True}] \quad \langle \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2, M\rangle \xrightarrow{\mathsf{e}} \langle e_1, M\rangle$$

$$[\textsc{If-False}] \quad \langle \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2, M\rangle \xrightarrow{\mathsf{e}} \langle e_2, M\rangle$$

$$[\textsc{Create}] \quad \frac{m = \mathsf{newloc}(M)}{\left\langle \{\overrightarrow{x_i = v_i}\}^S, M\right\rangle \xrightarrow{\mathsf{e}} \left\langle m^S, M[m^S \mapsto \{\overrightarrow{x_i = v_i}\}]\right\rangle}$$

$$\left[\begin{array}{c}\textsc{Parallel-}\\ \textsc{Result}\end{array}\right] \quad \langle v_1 \| v_2, M\rangle \xrightarrow{\mathsf{e}} \langle *, M\rangle \qquad [\textsc{Select}] \quad \frac{M(m^S) = \{\overrightarrow{x_i = v_i}\}}{\left\langle m^S.x_c, M\right\rangle \xrightarrow{\mathsf{e}} \langle v_c, M\rangle}$$

$$[\textsc{Assign}] \quad \frac{M(m^S) \neq \bot \qquad \forall p.\ v \neq \bot_p}{\left\langle m^S.x_c := v, M\right\rangle \xrightarrow{\mathsf{e}} \left\langle *, M[m^S.x_c \mapsto v]\right\rangle}$$

$$\left[\begin{array}{c}\textsc{Dangle-}\\ \textsc{Select}\end{array}\right] \quad \frac{M(m^S) = \bot \qquad p = \mathsf{persist}(m^S)}{\left\langle m^S.x_c, M\right\rangle \xrightarrow{\mathsf{e}} \langle \bot_p, M\rangle} \qquad \left[\begin{array}{c}\textsc{Dangle-}\\ \textsc{Assign}\end{array}\right] \quad \frac{M(m^S) = \bot \qquad p = \mathsf{persist}(m^S)}{\left\langle m^S.x_c := v, M\right\rangle \xrightarrow{\mathsf{e}} \langle \bot_p, M\rangle}$$

$$\left[\begin{array}{c}\textsc{Exists-}\\ \textsc{True}\end{array}\right] \quad \frac{M(m^S) \neq \bot}{\left\langle \mathsf{exists\ soft}\ m^S\ \mathsf{as}\ x :\ e_1\ \mathsf{else}\ e_2, M\right\rangle \xrightarrow{\mathsf{e}} \left\langle e_1\{m^S/x\}, M\right\rangle}$$

$$\left[\begin{array}{c}\textsc{Exists-}\\ \textsc{False}\end{array}\right] \quad \frac{M(m^S) = \bot}{\left\langle \mathsf{exists\ soft}\ m^S\ \mathsf{as}\ x :\ e_1\ \mathsf{else}\ e_2, M\right\rangle \xrightarrow{\mathsf{e}} \langle e_2, M\rangle}$$

$$\left[\begin{array}{c}\textsc{Eval-}\\ \textsc{Context}\end{array}\right] \quad \frac{\langle e, M\rangle \xrightarrow{\mathsf{e}} \langle e', M'\rangle}{\langle E[e], M\rangle \xrightarrow{\mathsf{e}} \langle E[e'], M'\rangle} \qquad \left[\begin{array}{c}\textsc{Fail-}\\ \textsc{Prop}\end{array}\right] \quad \langle F\,[\bot_p], M\rangle \xrightarrow{\mathsf{e}} \langle \bot_p, M\rangle$$

$$E ::= \mathsf{soft}\ [\,\cdot\,] \mid \mathsf{let}\ x = [\,\cdot\,]\ \mathsf{in}\ e \mid [\,\cdot\,] \| e \mid e \| [\,\cdot\,]$$
$$F ::= \mathsf{soft}\ [\,\cdot\,] \mid \mathsf{let}\ x = [\,\cdot\,]\ \mathsf{in}\ e$$

$$[\textsc{Prog-Step}] \quad \frac{\langle e, M\rangle \xrightarrow{\mathsf{e}} \langle e', M'\rangle}{\langle e, M\rangle \to \langle e', M'\rangle} \qquad [\mathrm{GC}] \quad \frac{\mathsf{gc}(G, \langle e, M\rangle)}{\langle e, M\rangle \to \langle e, M[G \mapsto \bot]\rangle}$$

Figure 3.5: Small-step operational semantics for ordinary (non-adversarial) execution of $\lambda^0_{persist}$

cation m^S to hold the record. The component S specifies the base type and storage label of the record. The storage label governs at what nodes the object can be created. The function $\mathsf{newloc}(M)$ deterministically generates a fresh memory location. If $\mathsf{address\text{-}space}(M)$ represents the set of location names in M (i.e., $\{m \ : \ \exists S.\, m^S \in \mathsf{dom}(M)\}$), then $\mathsf{newloc}(M) \notin \mathsf{address\text{-}space}(M)$ and $\mathsf{newloc}(M') = \mathsf{newloc}(M)$ if $\mathsf{address\text{-}space}(M') = \mathsf{address\text{-}space}(M)$.

Parallel composition expressions $e_1 \| e_2$ evaluate to the unit value (rule Parallel-Result).

The field-selection expression $v.x$ (rules SELECT and DANGLE-SELECT) evaluates v to a memory location m^S. If the location has not failed, the result of the selection is the value of the field x of the record at that location. Otherwise, a p-persistence failure occurs, where p is the persistence level of m^S, written $p = \mathsf{persist}(m^S)$.

The field-assignment expression $v_1.x := v_2$ evaluates v_1 to a memory location m^S (rules ASSIGN and DANGLE-ASSIGN) If the location has not failed, v_2 is assigned into the field x of the record at that location; otherwise, a p-persistence failure occurs (where $p = \mathsf{persist}(m^S)$). The notation $M[m^S.x_c \mapsto v]$ denotes the memory resulting from updating with value v the field x_c of the record at location m^S.

Persistence failures propagate outward dynamically (FAIL-PROP) until the whole program fails. The full $\lambda_{persist}$ language, defined in Section 3.5, can handle these failures.

In rule EVAL-CONTEXT, E represents an ordinary evaluation context, whereas F, in rule FAIL-PROP, specifies the contexts from which persistence failures propagate. Contexts are given as a term with a single hole (denoted by $[\cdot]$) in redex position. The syntax of E specifies the evaluation order.

The soft-reference expression $\mathsf{soft}\ e$ evaluates e to a hard reference and turns it into a soft reference. The soft-reference test ($\mathsf{exists}\ v\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2$) promotes the soft reference v (if valid) to a hard reference bound to x and evaluates e_1. If the reference is invalid, e_2 is evaluated instead.

In rule GC, the notation $\mathsf{gc}(G, \langle e, M\rangle)$ means that G is a set of locations that is *collectible*. G is considered collectible if it has no GC roots (i.e., hard references in e), and no location outside G has a hard reference into G.

Definition 1 (GC roots)**.** *A location m^S is a* GC root *in an expression e, written*

$\mathsf{root}(m^S, e)$*, if it is a hard reference in* e*. This is formally defined by the following inference rules:*

$$[\text{R1}]\quad \frac{}{\mathsf{root}(m^S, m^S)} \qquad [\text{R2}]\quad \frac{\mathsf{root}(m^S, e) \qquad \forall m_0^{S_0}.\ e \neq m_0^{S_0}}{\mathsf{root}(m^S, \mathsf{soft}\ e)}$$

$$[\text{R3}]\quad \frac{\exists i.\ \mathsf{root}(m^S, v_i)}{\mathsf{root}(m^S, \{\overrightarrow{x_i = v_i}\}^{S'})} \qquad [\text{R4}]\quad \frac{\mathsf{root}(m^S, v)}{\mathsf{root}(m^S, v.x)}$$

$$[\text{R5}]\quad \frac{\exists i.\ \mathsf{root}(m^S, v_i)}{\mathsf{root}(m^S, v_1.x := v_2)} \qquad [\text{R6}]\quad \frac{\mathsf{root}(m^S, e)}{\mathsf{root}(m^S, \lambda(x{:}\tau)[pc].\ e)}$$

$$[\text{R7}]\quad \frac{\exists i.\ \mathsf{root}(m^S, v_i)}{\mathsf{root}(m^S, v_1\ v_2)} \qquad [\text{R8}]\quad \frac{\exists i.\ \mathsf{root}(m^S, e_i)}{\mathsf{root}(m^S, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)}$$

$$[\text{R9}]\quad \frac{\exists i.\ \mathsf{root}(m^S, e_i)}{\mathsf{root}(m^S, \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)} \qquad [\text{R10}]\quad \frac{\exists i.\ \mathsf{root}(m^S, e_i)}{\mathsf{root}(m^S, \mathsf{exists}\ e_1\ \mathsf{as}\ x :\ e_2\ \mathsf{else}\ e_3)}$$

$$[\text{R11}]\quad \frac{\exists i.\ \mathsf{root}(m^S, e_i)}{\mathsf{root}(m^S, e_1 \| e_2)}$$

$\lambda_{persist}$, defined in Section 3.5, adds R12; and [$\lambda_{persist}$], defined in Section 3.6, adds R13:

$$[\text{R12}]\quad \frac{\exists i.\ \mathsf{root}(m^S, e_i)}{\mathsf{root}(m^S, \mathsf{try}\ e_1\ \mathsf{catch}\ p{:}\ e_2)} \qquad [\text{R13}]\quad \frac{\mathsf{root}(m^S, e)}{\mathsf{root}(m^S, [e])}$$

Definition 2 (Collectible groups)**.** *A set of locations* G *is a* collectible group *in a configuration* $\langle e, M\rangle$*, written* $\mathsf{gc}(G, \langle e, M\rangle)$*, if it does not contain any roots of* e*, and no location outside* G *has a hard reference into* G*.*

$$\begin{aligned}
&\mathsf{gc}(G, \langle e, M\rangle) \overset{def.}{\iff} \\
&\quad G \subseteq \mathsf{dom}(M) \\
&\quad \wedge\ (\nexists m^S \in G.\ \mathsf{root}(m^S, e)) \\
&\quad \wedge\ \forall m_0^{S_0} \in \mathsf{dom}(M).\ (M(m_0^{S_0}) \neq \bot \wedge \exists m^S \in G.\ \mathsf{root}(m^S, M(m_0^{S_0}))) \Rightarrow m_0^{S_0} \in G
\end{aligned}$$

3.4.4 Subtyping in $\lambda^0_{persist}$

The subtyping judgement $\vdash \tau_1 \leq \tau_2$ states that any value of type τ_1 can be treated as a value of type τ_2. Subtyping in $\lambda^0_{persist}$ is the least reflexive and transitive

$$[\text{S1}]\quad \frac{n > m}{\vdash \{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}_r \leq \{x_1{:}\tau_1, \ldots, x_m{:}\tau_m\}_r}$$

$$[\text{S2}]\quad \frac{\vdash R_1 \leq R_2}{\vdash \mathsf{soft}\ R_1 \leq \mathsf{soft}\ R_2} \qquad [\text{S3}]\quad \frac{\vdash b_1 \leq b_2 \qquad \vdash w_2 \preccurlyeq w_1}{\vdash (b_1)_{w_1} \leq (b_2)_{w_2}}$$

$$[\text{S4}]\quad \frac{\vdash \tau_2 \leq \tau_1 \qquad \vdash \tau_1' \leq \tau_2' \qquad \vdash pc_1 \preccurlyeq pc_2}{\vdash \tau_1 \xrightarrow{pc_1} \tau_1' \leq \tau_2 \xrightarrow{pc_2} \tau_2'}$$

$$[\text{S5}]\quad \frac{\vdash a_1^+ \preccurlyeq a_2^+ \qquad \vdash a_2^- \preccurlyeq a_1^- \qquad \vdash p_2 \preccurlyeq p_1}{\vdash \{\overrightarrow{x_i : \tau_i}\}_{(a_1^+, a_1^-, p_1)} \leq \{\overrightarrow{x_i : \tau_i}\}_{(a_2^+, a_2^-, p_2)}}$$

Figure 3.6: Subtyping rules for $\lambda^0_{persist}$

relation consistent with the rules given in Figure 3.6. Rule S1 gives standard width subtyping on records. Because records are mutable, there is no depth subtyping.

Subtyping on soft references is covariant (rule S2). While hard references may be soundly used as soft references, this is omitted for simplicity.

Rule S3 gives contravariant subtyping on integrity labels. Rule S4 gives standard subtyping on functions; the additional pc component is covariant. These are the opposite of the rules typically seen in work on information-flow security, accounting for our use of the trust ordering.

Rule S5 gives subtyping for labelled reference types. Subtyping is covariant on the a^+ component of the reference label and contravariant on the other two components. This ensures the bounds specified by the reference label of the subtype are at least as precise as those of the supertype.

3.4.5 Static semantics of $\lambda^0_{persist}$

Typing rules for $\lambda^0_{persist}$ are given in Figure 3.7. The notation $\mathsf{auth}^+(r)$, $\mathsf{auth}^-(r)$, and $\mathsf{persist}(r)$ give the upper authority (a^+), lower authority (a^-), and persistence (p) component of a reference label r, respectively. The notation $\mathsf{auth}^+(s)$

$$[\text{T-Bool}]\quad \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma; pc \vdash b : \mathsf{bool}_\top} \qquad [\text{T-Unit}]\quad \Gamma; pc \vdash * : \mathbf{1}$$

$$[\text{T-Var}]\quad \frac{\Gamma(x) = \tau}{\Gamma; pc \vdash x : \tau} \qquad [\text{T-Bottom}]\quad \frac{p \neq \top}{\Gamma; pc \vdash \bot_p : \tau}$$

$$[\text{T-Loc}]\quad \frac{\vdash_{wf} S : \mathsf{rectype} \qquad S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}}{\Gamma; pc \vdash m^S : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top} \qquad [\text{T-Soft}]\quad \frac{\Gamma; pc \vdash e : R_w}{\Gamma; pc \vdash \mathsf{soft}\ e : (\mathsf{soft}\ R)_w}$$

$$[\text{T-If}]\quad \frac{\begin{array}{c}\Gamma; pc \vdash v : \mathsf{bool}_w \\ \Gamma; pc \sqcap w \vdash e_i : \tau \ ^{(\forall i)} \\ \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc \sqcap w\end{array}}{\Gamma; pc \vdash \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau \sqcap w} \qquad [\text{T-Parallel}]\quad \frac{\begin{array}{c}\Gamma; pc \vdash e_i : \tau_i \ ^{(\forall i)} \\ \vdash \mathsf{auth}^+(\tau_i) \preccurlyeq pc \ ^{(\forall i)}\end{array}}{\Gamma; pc \vdash e_1 \parallel e_2 : \mathbf{1}}$$

$$[\text{T-Abs}]\quad \frac{\begin{array}{c}\Gamma, x{:}\tau'; pc' \vdash e : \tau \\ \vdash_{wf} (\tau' \xrightarrow{pc'} \tau)_\top : \mathsf{type} \qquad \vdash pc' \preccurlyeq pc\end{array}}{\Gamma; pc \vdash \lambda(x{:}\tau')[pc'].\, e : (\tau' \xrightarrow{pc'} \tau)_\top} \qquad [\text{T-App}]\quad \frac{\begin{array}{c}\Gamma; pc \vdash v_1 : (\tau' \xrightarrow{pc'} \tau)_w \\ \Gamma; pc \vdash v_2 : \tau' \\ \vdash pc' \preccurlyeq pc \sqcap w\end{array}}{\Gamma; pc \vdash v_1\ v_2 : \tau \sqcap w}$$

$$[\text{T-Record}]\quad \frac{\begin{array}{c}\vdash_{wf} S : \mathsf{rectype} \qquad S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)} \qquad \Gamma; pc \vdash v_i : \tau_i' \ ^{(\forall i)} \\ \vdash \tau_i' \leq \tau_i \ ^{(\forall i)} \qquad \vdash \mathsf{auth}^+(\tau_i') \preccurlyeq pc \ ^{(\forall i)} \qquad \vdash \mathsf{integ}(\tau_i) \preccurlyeq pc \ ^{(\forall i)} \qquad \vdash p \preccurlyeq pc\end{array}}{\Gamma; pc \vdash \{\overrightarrow{x_i = v_i}\}^S : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top}$$

$$[\text{T-Select}]\quad \frac{\begin{array}{c}\Gamma; pc \vdash v : (\{\overrightarrow{x_i : \tau_i}\}_r)_w \\ \vdash \mathsf{auth}^+(r) \preccurlyeq pc \\ w' = w \sqcap \mathsf{persist}(r)\end{array}}{\Gamma; pc \vdash v.x_c : \tau_c \sqcap w'} \qquad [\text{T-Assign}]\quad \frac{\begin{array}{c}\Gamma; pc \vdash v_1 : (\{\overrightarrow{x_i : \tau_i}\}_r)_w \\ \vdash \mathsf{auth}^+(r) \preccurlyeq pc \\ \Gamma; pc \vdash v_2 : \tau \\ \vdash \tau \sqcap pc \sqcap w \leq \tau_c \\ \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc \sqcap w\end{array}}{\Gamma; pc \vdash v_1.x_c := v_2 : \mathbf{1}}$$

$$[\text{T-Exists}]\quad \frac{\begin{array}{c}\Gamma; pc \vdash v : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_r)_w \qquad \vdash \mathsf{auth}^+(r) \preccurlyeq pc \sqcap w \\ w' = \mathsf{auth}^-(r) \sqcap \mathsf{persist}(r) \sqcap w \qquad \Gamma, x{:}(\{\overrightarrow{x_i : \tau_i}\}_r)_w; pc \sqcap w' \vdash e_1 : \tau \\ \Gamma; pc \sqcap w' \vdash e_2 : \tau \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc \sqcap w'\end{array}}{\Gamma; pc \vdash \mathsf{exists}\ v\ \mathsf{as}\ x :\ e_1\ \mathsf{else}\ e_2 : \tau \sqcap w'}$$

$$[\text{T-Let}]\quad \frac{\begin{array}{c}\Gamma; pc \vdash e_1 : \tau' \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \\ w = \mathsf{integ}(\tau') \qquad pc' = pc \sqcap w \\ \Gamma, x{:}\tau'; pc' \vdash e_2 : \tau \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc'\end{array}}{\Gamma; pc \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau \sqcap w} \qquad [\text{T-Subsume}]\quad \frac{\Gamma; pc \vdash e : \tau' \qquad \vdash \tau' \leq \tau}{\Gamma; pc \vdash e : \tau}$$

Figure 3.7: Typing rules for $\lambda^0_{persist}$

$$[\text{WT1}]\ \vdash_{wf} \mathsf{bool}_w : \mathsf{type} \qquad [\text{WT2}]\ \frac{\vdash_{wf} R_\top : \mathsf{type}}{\vdash_{wf} (\mathsf{soft}\ R)_w : \mathsf{type}} \qquad [\text{WT3}]\ \vdash_{wf} \mathbf{1} : \mathsf{type}$$

$$[\text{WT4}]\quad \frac{\vdash pc \preccurlyeq w \qquad \vdash \mathsf{auth}^+(\tau_1) \sqcup \mathsf{auth}^+(\tau_2) \preccurlyeq pc \qquad \vdash_{wf} \tau_1 : \mathsf{type} \qquad \vdash_{wf} \tau_2 : \mathsf{type}}{\vdash_{wf} (\tau_1 \xrightarrow{pc} \tau_2)_w : \mathsf{type}}$$

$$[\text{WT5}]\quad \frac{\vdash_{wf} \tau_i : \mathsf{type}^{\ (\forall i)} \qquad \vdash \mathsf{auth}^+(\tau_i) \preccurlyeq a^{+\ (\forall i)} \qquad \vdash a^+ \preccurlyeq w \sqcap p \qquad \vdash a^- \preccurlyeq a^+}{\vdash_{wf} (\{\overrightarrow{x_i : \tau_i}\}_{(a^+,a^-,p)})_w : \mathsf{type}}$$

$$[\text{WT6}]\quad \frac{\vdash_{wf} (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top : \mathsf{type} \qquad \vdash \mathsf{integ}(\tau_i) \preccurlyeq p^{\ (\forall i)}}{\vdash_{wf} \{\overrightarrow{x_i : \tau_i}\}_{(a,p)} : \mathsf{rectype}}$$

Figure 3.8: Well-formedness of types

and $\mathsf{persist}(s)$ give the authority and persistence component of a storage label, respectively. The notation $\mathsf{auth}^+(\tau)$ gives the authority level needed to create a hard reference to a value of type τ, the integrity of τ is $\mathsf{integ}(\tau)$, and $\tau \sqcap \ell$ denotes the type obtained by tainting (meeting) the integrity of τ with ℓ:

$$\mathsf{auth}^+(\mathsf{bool}) = \mathsf{auth}^+(\mathbf{1}) = \mathsf{auth}^+(\mathsf{soft}\ R) = \bot$$

$$\mathsf{integ}(b_w) = w \qquad (b_w) \sqcap \ell = b_{w \sqcap \ell} \qquad \mathsf{auth}^+(\tau_1 \xrightarrow{pc} \tau_2) = pc$$

$$\mathsf{integ}(\mathbf{1}) = \top \qquad \mathbf{1} \sqcap \ell = \mathbf{1} \qquad \mathsf{auth}^+(\{\overrightarrow{x_i : \tau_i}\}_s) = \mathsf{auth}^+(s)$$

The typing context includes a *type assignment* Γ and the program-counter label pc. Γ is a finite partial map from variables x to types τ, expressed as a finite list of $x : \tau$ entries. We write $x : \tau \in \Gamma$ and $\Gamma(x) = \tau$ interchangeably. For an expression e that is well-typed in a context $\Gamma; pc$, the type checker produces a type τ. The typing assertion $\Gamma; pc \vdash e : \tau$, therefore, means that the expression e has type τ under type assignment Γ with program-counter label pc.

Most of the typing rules are standard rules, extended to ensure that the pc is sufficiently high to obtain any hard references that may result from evaluating subexpressions, and that the pc is suitably tainted where appropriate.

Rule T-BOTTOM says persistence failures can have any well-formed type.

Rule T-ABS checks function values. It ensures that the function's program-counter label pc' accurately summarizes the authority levels of the references contained in the closure, and that the pc is high enough to create this closure. The body is checked with program-counter label pc', so in rule T-APP, the function can only be used by code with sufficient integrity.

Rule T-RECORD checks the creation of records. It requires that the annotation S be well-formed. Also, the pc must be high enough to create any hard references that appear in the fields, and to write to the fields themselves.

When using a hard reference v_1, the pc must have sufficient authority to possess v_1 (rules T-SELECT and T-ASSIGN). When assigning through v_1, hard references contained in the assigned value v_2 also require authority. Since the integrity and persistence of v_1 can affect whether the assignment succeeds, we taint the pc with these labels before comparing with the authority requirement of v_2.

Rule T-EXISTS checks soft-reference validity tests. It ensures that the pc has the authority to promote the reference.

The rules for determining the well-formedness of types are given in Figure 3.8. In rule WT5, a reference type $(\{\overrightarrow{x_i : \tau_i}\}_{(a^+,a^-,p)})_w$ is well-formed only if the upper authority label a^+ is an upper bound on the authority levels of the field types τ_i. This ensures that the upper authority label is an accurate summary of the authority required by the fields. We also require a^+ be bounded from above by the integrity w of the reference, since low-integrity data should not influence the creation of high-authority references. To ensure that hosts are able to create hard references to the objects they store, we also require $\mathsf{auth}^+(r)$ to be bounded from above by the persistence level p of the record.

$$\left[\begin{smallmatrix}\textsc{Soft-}\\ \textsc{Select}\end{smallmatrix}\right] \quad \frac{\langle m^S.x_c, M\rangle \xrightarrow{\mathsf{e}} \langle v, M\rangle}{\langle (\mathsf{soft}\ m^S).x_c, M\rangle \xrightarrow{\mathsf{e}} \langle v, M\rangle} \qquad \left[\begin{smallmatrix}\textsc{Soft-}\\ \textsc{Assign}\end{smallmatrix}\right] \quad \frac{\langle m^S.x_c := v, M\rangle \xrightarrow{\mathsf{e}} \langle v', M'\rangle}{\langle (\mathsf{soft}\ m^S).x_c := v, M\rangle \xrightarrow{\mathsf{e}} \langle v', M'\rangle}$$

$$\left[\begin{smallmatrix}\textsc{Try-}\\ \textsc{Val}\end{smallmatrix}\right] \quad \frac{\forall p'.\ v \neq \bot_{p'}}{\langle \mathsf{try}\ v\ \mathsf{catch}\ p\colon e, M\rangle \xrightarrow{\mathsf{e}} \langle v, M\rangle} \qquad \left[\begin{smallmatrix}\textsc{Try-}\\ \textsc{Catch}\end{smallmatrix}\right] \quad \frac{p \preccurlyeq p'}{\langle \mathsf{try}\ \bot_{p'}\ \mathsf{catch}\ p\colon e, M\rangle \xrightarrow{\mathsf{e}} \langle e, M\rangle}$$

$$\left[\begin{smallmatrix}\textsc{Try-}\\ \textsc{Esc}\end{smallmatrix}\right] \quad \frac{p \not\preccurlyeq p'}{\langle \mathsf{try}\ \bot_{p'}\ \mathsf{catch}\ p\colon e, M\rangle \xrightarrow{\mathsf{e}} \langle \bot_{p'}, M\rangle} \qquad E ::= \ldots \mid \mathsf{try}\ [\,\cdot\,]\ \mathsf{catch}\ p\colon e$$

$$\left[\begin{smallmatrix}\textsc{T-Soft-}\\ \textsc{Select}\end{smallmatrix}\right] \quad \frac{\Gamma; pc; \mathcal{H} \vdash v : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_r)_w, \top \qquad \vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc \qquad p = \mathsf{auth}^-(r) \sqcap \mathsf{persist}(r) \sqcap w \qquad \vdash \mathcal{H} \preccurlyeq p}{\Gamma; pc; \mathcal{H} \vdash v.x_c : \tau_c \sqcap p, p}$$

$$\left[\begin{smallmatrix}\textsc{T-Soft-}\\ \textsc{Assign}\end{smallmatrix}\right] \quad \frac{\Gamma; pc; \mathcal{H} \vdash v_1 : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_r)_w, \top \qquad p = \mathsf{auth}^-(r) \sqcap \mathsf{persist}(r) \sqcap w \qquad \Gamma; pc; \mathcal{H} \vdash v_2 : \tau, \top \qquad \vdash \tau \sqcap pc \sqcap p \leq \tau_c \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc \sqcap p \qquad \vdash \mathcal{H} \preccurlyeq p}{\Gamma; pc; \mathcal{H} \vdash v_1.x_c := v_2 : \mathbf{1}, p}$$

$$[\textsc{T-Try}] \quad \frac{\Gamma; pc; \mathcal{H}, p \vdash e_1 : \tau, \mathcal{X}_1 \qquad w = \prod_{p' \in \mathcal{X}_1} (p \sqcup p') \qquad \Gamma; pc \sqcap w \sqcap \mathsf{integ}(\tau); \mathcal{H} \vdash e_2 : \tau, \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc}{\Gamma; pc; \mathcal{H} \vdash \mathsf{try}\ e_1\ \mathsf{catch}\ p\colon e_2 : \tau \sqcap w, (\mathcal{X}_1/p) \sqcap \mathcal{X}_2}$$

Figure 3.9: Additional small-step evaluation and typing rules for $\lambda_{persist}$

3.5 Ensuring referential integrity

In a distributed system, references can span trust domains, so to be secure and reliable, program code must in general be ready to encounter a dangling reference, one perhaps created by the adversary. Therefore, we extend $\lambda^0_{persist}$ with *persistence-failure handlers* to obtain the full $\lambda_{persist}$ language (see Appendix 3.A.1 for its full syntax). The type system of $\lambda_{persist}$ forces the programmer to be aware of and to handle all potential failures.

Because low-persistence references may be used frequently, handling persistent failures immediately at each use would be awkward. Instead, $\lambda_{persist}$ factors out failure-handling code from ordinary code by treating failures as a kind of exception.

The value of (try e_1 catch p: e_2) is the value of evaluating e_1. If a dangling pointer at persistence level p or higher is encountered, the error-handling expression e_2 is evaluated instead. A `try` expression creates a context (e_1) in which the programmer can write simpler code under the assumption that certain per-

sistence failures are impossible, yet without sacrificing the property that all failures are handled.

3.5.1 Persistence handler levels

To track the failures that can be handled in the current context, a *set* of persistence levels $\mathcal{H}$ is used. Formally, $\mathcal{H}$ is drawn from the bounded meet-semilattice given by the upper powerdomain of persistence levels. The elements of the powerdomain are the finitely generated subsets[2] of $\mathcal{L}$, modulo equivalence relation (3.1). The ordering on these elements is given by (3.2). If we choose maximal sets to represent the equivalence classes, then the meet operation is set union.

$$A \sim B \overset{def.}{\iff} \forall \ell \in \mathcal{L}.\ ((\exists a \in A.\ a \preccurlyeq \ell) \Leftrightarrow (\exists b \in B.\ b \preccurlyeq \ell)) \tag{3.1}$$

$$A \preccurlyeq_{\wp} B \overset{def.}{\iff} \forall b \in B.\ \exists a \in A.\ a \preccurlyeq b \tag{3.2}$$

Functions $\lambda(x:\tau)[pc;\mathcal{H}].\, e$ and function types $\tau_1 \xrightarrow{pc,\mathcal{H}} \tau_2$ are extended with an $\mathcal{H}$ component, which is an upper bound on the $\mathcal{H}$ label of the caller. It gives a set of lower bounds on the persistence levels of references that the function follows.

3.5.2 Example

Returning to the directory example in Figure 3.1, Alice can add a place to the list of sightseeing ideas with the code below. This code starts at Alice's `docs` directory, traverses the reference to the scratchpad, and invokes an `add` method to add a museum.

[2]Non-empty subsets that are either finite or contain $\bot$.

```
let pad = docs.scratchpad
in try pad.add "Rodin Museum"
   catch ⊥: …
```

The expression `pad.add` follows a hard reference to the scratchpad. Despite the hard reference, a `try` is needed because Alice does not trust host `U` to persist the scratchpad.

3.5.3 Operational semantics of $\lambda_{persist}$

The small-step operational semantics of $\lambda_{persist}$ extends that of $\lambda^0_{persist}$ with the rules in Figure 3.9. Two rules are for using soft references directly, and three are for persistence-failure handlers. Failures propagate outward dynamically (TRY-ESC) until either they are handled by a failure handler (TRY-CATCH), or the whole program fails.

The persistence-failure handler try e_1 catch p: e_2 handles p-persistence failures. (Failures that occur at persistence levels higher than p are also considered to be p-persistence failures.) If e_1 fails with such a failure, then e_2 is evaluated; otherwise the result of the try is that of e_1. Persistence-failure handlers enable e_1 to call functions that require more trust (lower $\mathcal{H}$) than provided by the context. Appendix 3.A.2 gives the full operational semantics for $\lambda_{persist}$.

3.5.4 Subtyping in $\lambda_{persist}$

The subtyping rules are the same as for $\lambda^0_{persist}$, except that function subtyping is also contravariant on the $\mathcal{H}$ component. Full subtyping rules are given in Appendix 3.A.3.

$$[\textsc{Create}] \quad \frac{m = \mathsf{newloc}(M) \qquad S = \{\overrightarrow{x_i : \tau_i}\}_s \qquad v_i' = v_i \blacktriangleright_\alpha \tau_i}{\left\langle \{\overrightarrow{x_i = v_i}\}^S, M \right\rangle \xrightarrow{\mathsf{e}} \left\langle m^S, M[m^S \mapsto \{\overrightarrow{x_i = v_i'}\}] \right\rangle}$$

$$[\textsc{Select}] \quad \frac{M(m^S) = \{\overrightarrow{x_i = v_i}\} \qquad S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}}{\left\langle m^S.x_c, M \right\rangle \xrightarrow{\mathsf{e}} \langle v_c \blacktriangleright_\alpha p, M \rangle}$$

$$[\textsc{Assign}] \quad \frac{M(m^S) \neq \bot \qquad \forall p'.\ v \neq \bot_{p'} \qquad S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)} \qquad M' = M[m^S.x_c \mapsto v \blacktriangleright_\alpha \tau_c]}{\left\langle m^S.x_c := v, M \right\rangle \xrightarrow{\mathsf{e}} \left\langle * \blacktriangleright_\alpha p, M' \right\rangle}$$

$$\left[\begin{array}{c}\textsc{Dangle-}\\ \textsc{Select}\end{array}\right] \quad \frac{M(m^S) = \bot \qquad S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}}{\left\langle m^S.x_c, M \right\rangle \xrightarrow{\mathsf{e}} \langle \bot_p \blacktriangleright_\alpha p, M \rangle}$$

$$\left[\begin{array}{c}\textsc{Dangle-}\\ \textsc{Assign}\end{array}\right] \quad \frac{M(m^S) = \bot \qquad S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}}{\left\langle m^S.x_c := v, M \right\rangle \xrightarrow{\mathsf{e}} \langle \bot_p \blacktriangleright_\alpha p, M \rangle}$$

$$\left[\begin{array}{c}\textsc{Soft-}\\ \textsc{Select}\end{array}\right] \quad \frac{\left\langle m^S.x_c, M \right\rangle \xrightarrow{\mathsf{e}} \langle v, M \rangle \qquad S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}}{\left\langle (\mathsf{soft}\ m^S).x_c, M \right\rangle \xrightarrow{\mathsf{e}} \langle v \blacktriangleright_\alpha (a \sqcap p), M \rangle}$$

$$\left[\begin{array}{c}\textsc{Soft-}\\ \textsc{Assign}\end{array}\right] \quad \frac{\left\langle m^S.x_c := v, M \right\rangle \xrightarrow{\mathsf{e}} \left\langle v', M' \right\rangle \qquad S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}}{\left\langle (\mathsf{soft}\ m^S).x_c := v, M \right\rangle \xrightarrow{\mathsf{e}} \left\langle v' \blacktriangleright_\alpha (a \sqcap p), M' \right\rangle}$$

$$\left[\begin{array}{c}\textsc{Exists-}\\ \textsc{True}\end{array}\right] \quad \frac{M(m^S) \neq \bot \qquad S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}}{\left\langle \mathsf{exists\ soft}\ m^S\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2, M \right\rangle \xrightarrow{\mathsf{e}} \left\langle (e_1\{m^S/x\}) \blacktriangleright_\alpha (a \sqcap p), M \right\rangle}$$

$$\left[\begin{array}{c}\textsc{Exists-}\\ \textsc{False}\end{array}\right] \quad \frac{M(m^S) \neq \bot \qquad S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}}{\left\langle \mathsf{exists\ soft}\ m^S\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2, M \right\rangle \xrightarrow{\mathsf{e}} \langle e_2 \blacktriangleright_\alpha (a \sqcap p), M \rangle}$$

$$\left[\begin{array}{c}\textsc{Bracket-}\\ \textsc{Select}\end{array}\right] \quad \left\langle [m^S].x_c, M \right\rangle \xrightarrow{\mathsf{e}} \left\langle [m^S.x_c], M \right\rangle$$

$$\left[\begin{array}{c}\textsc{Bracket-}\\ \textsc{Assign}\end{array}\right] \quad \left\langle [m^S].x_c := v, M \right\rangle \xrightarrow{\mathsf{e}} \left\langle [m^S.x_c := v], M \right\rangle$$

$$\left[\begin{array}{c}\textsc{Bracket-}\\ \textsc{Soft-Select}\end{array}\right] \quad \left\langle [\mathsf{soft}\ m^S].x_c, M \right\rangle \xrightarrow{\mathsf{e}} \left\langle [(\mathsf{soft}\ m^S).x_c], M \right\rangle$$

$$\left[\begin{array}{c}\textsc{Bracket-}\\ \textsc{Soft-Assign}\end{array}\right] \quad \left\langle [\mathsf{soft}\ m^S].x_c := v, M \right\rangle \xrightarrow{\mathsf{e}} \left\langle [(\mathsf{soft}\ m^S).x_c := v], M \right\rangle$$

$$\left[\begin{array}{c}\textsc{Bracket-}\\ \textsc{Soft}\end{array}\right] \quad \left\langle \mathsf{soft}\ [m^S], M \right\rangle \xrightarrow{\mathsf{e}} \left\langle [\mathsf{soft}\ m^S], M \right\rangle$$

$$\left[\begin{array}{c}\textsc{Double-}\\ \textsc{Bracket}\end{array}\right] \quad \langle [[v]], M \rangle \xrightarrow{\mathsf{e}} \langle [v], M \rangle$$

$$\left[\begin{array}{c}\textsc{Bracket-}\\ \textsc{Exists}\end{array}\right] \quad \langle \mathsf{exists}\ [v]\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2, M \rangle \xrightarrow{\mathsf{e}} \langle [\mathsf{exists}\ v\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2], M \rangle$$

$$\left[\begin{array}{c}\textsc{Bracket-}\\ \textsc{Apply}\end{array}\right] \quad \langle [\lambda(x{:}\tau)[pc; \mathcal{H}].\, e]\ v, M \rangle \xrightarrow{\mathsf{e}} \langle [(\lambda(x{:}\tau)[pc; \mathcal{H}].\, e)\ v], M \rangle$$

$$\left[\begin{array}{c}\textsc{Bracket-}\\ \textsc{Try}\end{array}\right] \quad \langle \mathsf{try}\ [v]\ \mathsf{catch}\ p{:}\ e, M \rangle \xrightarrow{\mathsf{e}} \langle [\mathsf{try}\ v\ \mathsf{catch}\ p{:}\ e], M \rangle$$

$$\left[\begin{array}{c}\textsc{Bracket-}\\ \textsc{If}\end{array}\right] \quad \langle \mathsf{if}\ [v]\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, M \rangle \xrightarrow{\mathsf{e}} \langle [\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2], M \rangle$$

$$\left[\begin{array}{c}\textsc{Bracket-}\\ \textsc{Let}\end{array}\right] \quad \frac{\forall p.\ v \neq \bot_p}{\langle \mathsf{let}\ x = [v]\ \mathsf{in}\ e, M \rangle \xrightarrow{\mathsf{e}} \langle [e\{[v]/x\}], M \rangle}$$

$$\left[\begin{array}{c}\textsc{Bracket-}\\ \textsc{Context}\end{array}\right] \quad \frac{\langle e, M \rangle \xrightarrow{\mathsf{e}} \left\langle e', M' \right\rangle}{\langle [e], M \rangle \xrightarrow{\mathsf{e}} \left\langle [e'], M' \right\rangle}$$

$$\left[\begin{array}{c}\textsc{Bracket-}\\ \textsc{Fail}\end{array}\right] \quad \langle F[[\bot_p]], M \rangle \xrightarrow{\mathsf{e}} \langle [\bot_p], M \rangle$$

Figure 3.10: Small-step operational semantics extensions for ordinary execution of $[\lambda_{persist}]$

3.5.5 Static semantics of $\lambda_{persist}$

The typing rules for $\lambda_{persist}$ extend those for $\lambda^0_{persist}$. They augment the typing context with a *handler environment* $\mathcal{H}$, indicating the set of persistence failures the evaluation context can handle. For an expression e that is well-typed in a context $\Gamma; pc; \mathcal{H}$, typing judgements additionally produce an effect $\mathcal{X}$, which is a set indicating the persistence failures e can produce during evaluation. The typing assertion $\Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}$, therefore, means that the expression e has type τ and effect $\mathcal{X}$ under type assignment Γ, current program-counter label pc, and handler environment $\mathcal{H}$.

The typing rules for $\lambda^0_{persist}$ are converted straightforwardly to thread $\mathcal{H}$ and $\mathcal{X}$ through typing judgements. Rules T-SELECT and T-ASSIGN gain premises to ensure the context has a suitable handler in case dereferences fail. Appendix 3.A.4 gives the full set of converted rules.

Figure 3.9 gives three new typing rules. T-SOFT-SELECT and T-SOFT-ASSIGN check direct uses of soft references. They taint the integrity of the dereference with $\mathsf{auth}^{-}(r)$ because the result of the dereference is affected by those able to create a hard reference and thereby prevent the referent from being garbage-collected (Section 3.2.5). Rule T-TRY checks try expressions. To reflect the installation of a p-persistence handler, p is added to the handler environment $\mathcal{H}$ when checking e_1. The value w in the typing rule is a conservative summary of the persistence errors that can occur while evaluating e_1 and not handled by the p-persistence handler. Because evaluation of e_2 depends on the result of e_1, the pc label for evaluating e_2 is tainted by w. In this rule, the notation $\mathcal{X}/p$ denotes the subset of persistence errors $\mathcal{X}$ not handled by p.

$$\mathcal{H}/p \triangleq \{p' \in \mathcal{H} : p \not\leq p'\}$$

$$[\alpha\text{-CREATE}]\quad \frac{m = \mathsf{newloc}(M) \qquad \varnothing; \top; \top \vdash \{\overrightarrow{x_i = [v_i]}\}^S : R_\top, \top \qquad \vdash^{\alpha}_{[wf]} M[m^S \mapsto \{\overrightarrow{x_i = [v_i]}\}] \qquad \alpha \notin \mathsf{persist}(S)}{\langle e, M\rangle \rightsquigarrow_\alpha \left\langle e, M[m^S \mapsto \{\overrightarrow{x_i = [v_i]}\}]\right\rangle}$$

$$[\alpha\text{-ASSIGN}]\quad \frac{m^S \in \mathsf{dom}(M) \qquad M(m^S) \neq \bot \qquad S = \{\overrightarrow{x_i : \tau_i}\}_s \qquad \varnothing; \top; \top \vdash [v] : \tau_c, \top \qquad \vdash^{\alpha}_{[wf]} M[m^S.x_c \mapsto [v]]}{\langle e, M\rangle \rightsquigarrow_\alpha \langle e, M[m^S.x_c \mapsto [v]]\rangle}$$

$$[\alpha\text{-FORGET}]\quad \frac{m^S \in \mathsf{dom}(M) \qquad \alpha \notin \mathsf{persist}(S)}{\langle e, M\rangle \rightsquigarrow_\alpha \langle e, M[m^S \mapsto \bot]\rangle}$$

Figure 3.11: Effects caused by the α-adversary

3.6 The power of the adversary

The power of the adversary is modelled by extending the operational semantics of Figure 3.5 with additional transitions. To support reasoning about what an adversary may have affected in a partially evaluated program, $\lambda_{persist}$ is augmented to include bracketed forms. We call the resulting augmented language $[\lambda_{persist}]$. We write $[e]$ to indicate an expression that may have been influenced by the adversary, and $[v]$ to indicate an influenced value. Doubly bracketed values are considered expressions and not values.

The extended syntax and the rule for typing bracketed forms appear below. Extensions for the operational semantics appear in Figure 3.10.

$$\text{Values} \quad v ::= \ldots \mid [v]$$

$$\text{Terms} \quad e ::= \ldots \mid [e]$$

$$[\text{T-BRACKET}]\quad \frac{\Gamma; pc \sqcap \ell; \mathcal{H} \vdash e : \tau, \mathcal{X} \qquad \alpha \notin \ell \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc \sqcap \ell}{\Gamma; pc; \mathcal{H} \vdash [e] : \tau \sqcap \ell, \mathcal{X}}$$

The operational semantics is extended by adding new rules that propagate brackets in the obvious manner. Rules CREATE, SELECT, DANGLE-SELECT,

ASSIGN, DANGLE-ASSIGN, SOFT-SELECT, SOFT-ASSIGN, EXISTS-TRUE, and EXISTS-FALSE are amended to ensure low-integrity expressions are bracketed. To do this, they use the auto-bracketing function $e \blacktriangleright_\alpha \ell$. The notation $e \blacktriangleright_\alpha \tau$ is shorthand for $e \blacktriangleright_\alpha \mathsf{integ}(\tau)$.

$$e \blacktriangleright_\alpha \ell = \begin{cases} e, & \text{if } \vdash \alpha \preccurlyeq \ell \text{ or } \exists e'.\ e = [e']; \\ [e], & \text{otherwise.} \end{cases}$$

Also, rules TRY-VAL and LET are amended to prevent a transition when v is bracketed, and rules that disallow transitions on bottom values ($\bot_p$) are amended to prevent transitions on bracketed bottom values.

It is important to know that any evaluation of a program in the original language can be simulated in the augmented language, which amounts to showing that the rules cover all the ways that brackets can appear. A straightforward induction proves this (see Lemma 1 in Section 3.7).

Rules for the adversary's transitions are given in Figure 3.11. Adversaries may create new records, modify existing records, or remove records from memory altogether, but their ability is bounded by an integrity label $\alpha \in \mathcal{L}$. Such an α-*adversary* has all creation authority except α and higher, can modify any record field except those with α (or higher) integrity, and can delete any record except those with α (or higher) persistence. A small evaluation step taken in the presence of an α-adversary is a transition from a machine configuration $\langle e, M\rangle$ to another configuration $\langle e', M'\rangle$, written $\langle e, M\rangle \rightarrow_\alpha \langle e', M'\rangle$.

While it is reasonable to allow the adversary to create ill-typed values, an implementation with run-time type checking can catch ill-typed values when they cross between hosts and replace them with well-typed default values. Therefore, the adversary's transitions embody a simplifying assumption that the adversary can only create well-typed values.

Rule α-CREATE lets the adversary create records at new memory locations. The premise $\varnothing; \top; \top \vdash \{\overrightarrow{x_i = [v_i]}\}^S : R_\top, \top$ ensures that the records are well-typed values and that new hard references satisfy the restrictions on the adversary. The premise $\vdash^{\alpha}_{[wf]} M[m^S \mapsto \{\overrightarrow{x_i = [v_i]}\}]$ ensures that the resulting memory is well-formed (formally defined in Section 3.7.1), so the adversary cannot create references to unknown memory locations.

Rule α-ASSIGN lets the adversary modify existing records. The premise $M(m^S) \neq \bot$ ensures that the record being modified still exists in memory. The premise $\varnothing; \top; \top \vdash [v] : \tau_c, \top$ ensures that the assignment is well-typed and that new hard references satisfy the restrictions on the adversary. The premise $\vdash^{\alpha}_{[wf]} M[m^S.x_c \mapsto [v]]$ ensures that the resulting memory is well-formed, so the adversary cannot create references to unknown memory locations.

Rule α-FORGET lets the adversary drop records from memory. The premise $\alpha \not\leq \mathsf{persist}(S)$ restricts the persistence level of dropped records.

3.7 Results

The goal of $\lambda_{persist}$ is to prevent accidental persistence and to ensure that the adversary cannot damage referential integrity or cause storage attacks. Accidental persistence is prevented by the use of persistence policies. We now show how to formalize the other security properties and prove that $\lambda_{persist}$ satisfies these properties.

3.7.1 Well-formedness

To ensure we consider only sensible memories and configurations, we define well-formedness for memories, and use that in the definition for configurations.

Definition 3 (Well-formed $\lambda_{persist}$ memory)**.** *A $\lambda_{persist}$ memory M is well-formed, written $\vdash_{wf} M$, if each record stored in M satisfies two conditions: the record's type corresponds to the type of the record's location in M, and every location mentioned in the record is valid in M.*

More precisely, whenever M maps a reference m^S to a record value $\{\overrightarrow{x_i = v_i}\}$,

- *S is well-formed,*
- *each v_i has the appropriate type (as specified by S),*
- *the locations mentioned in the field values v_i are in the domain of M:*

$$\begin{aligned} \vdash_{wf} M \overset{def.}{\iff} & \left(M(m^S) = \{\overrightarrow{x_i = v_i}\} \wedge S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}\right. \\ \Rightarrow \quad & \vdash_{wf} S : \mathsf{rectype} \\ & \wedge \left(\forall i.\ \varnothing; \top; \top \vdash v_i : \tau_i, \top\right) \\ & \left. \wedge\ \mathsf{locs}(\{\overrightarrow{x_i = v_i}\}) \subseteq \mathsf{dom}(M)\right) \end{aligned}$$

Our notion of a well-formed configuration relies on knowing when a location is non-collectible (cannot be garbage collected), which we now define.

Definition 4 (Non-collectible locations)**.** *A memory location m^S is* non-collectible *in a configuration $\langle e, M\rangle$, written $\mathsf{nc}(m^S, \langle e, M\rangle)$, if it is reachable from a GC root of e through a path of hard references.*

This is defined formally by the following induction rules:

$$[\text{NC1}]\ \frac{\mathsf{root}(m^S, e)}{\mathsf{nc}(m^S, \langle e, M\rangle)} \qquad [\text{NC2}]\ \frac{\mathsf{root}(m_1^{S_1}, e) \qquad M(m_1^{S_1}) = \{\overrightarrow{x_i = v_i}\} \qquad \exists c.\ \mathsf{nc}(m^S, \langle v_c, M\rangle)}{\mathsf{nc}(m^S, \langle e, M\rangle)}$$

Well-formedness of configurations is parameterized on an adversary α.

Definition 5 (Well-formed $\lambda_{persist}$ configuration)**.** *A $\lambda_{persist}$ configuration $\langle e, M\rangle$ is well-formed, written $\vdash^{\alpha}_{wf} \langle e, M\rangle$, if the following all hold:*

- *M is well-formed;*
- *the locations mentioned in e are valid in M;*
- *no non-collectible high-persistence location is deleted; and*
- *if G is a minimal collectible group in which a high-persistence location is deleted, then all locations in G are also deleted.*

Formally,

$$\begin{aligned}\vdash^{\alpha}_{wf} \langle e, M\rangle \overset{def.}{\iff} \vdash_{wf} M \wedge \mathsf{locs}(e) \subseteq \mathsf{dom}(M)\\ \wedge(\forall m^S.\mathsf{nc}(m^S, \langle e, M\rangle) \wedge \vdash \alpha \preccurlyeq \mathsf{persist}(S)\\ \Rightarrow M(m^S) \neq \bot)\\ \wedge(\forall G.\ \mathsf{gc}(G, \langle e, M\rangle) \wedge (\nexists G' \subseteq G.\ \mathsf{gc}(G', \langle e, M\rangle))\\ \wedge(\exists m_0^{S_0} \in G.\ \vdash \alpha \preccurlyeq \mathsf{persist}(S_0) \wedge M(m_0^{S_0}) = \bot)\\ \Rightarrow \forall m^S \in G.\ M(m^S) = \bot)\end{aligned}$$

A $\lambda_{persist}$ configuration is well-formed in a non-adversarial setting, written $\vdash_{wf} \langle e, M\rangle$, if it is well-formed with respect to the $\bot$ adversary.

Corresponding well-formedness conditions are defined similarly for $[\lambda_{persist}]$ memories, written $\vdash^{\alpha}_{[wf]} M$ and $\vdash_{[wf]} M$, and for $[\lambda_{persist}]$ configurations, written $\vdash^{\alpha}_{[wf]} \langle e, M\rangle$ and $\vdash_{[wf]} \langle e, M\rangle$. Well-formedness of $[\lambda_{persist}]$ memories is parameterized on an α-adversary, because values appearing in low-integrity record fields must be bracketed.

Definition 6 (Well-formed $[\lambda_{persist}]$ memory)**.** *A $[\lambda_{persist}]$ memory M is well-formed with respect to an adversary α (written $\vdash^{\alpha}_{[wf]} M$) if it is a well-formed $\lambda_{persist}$ memory and all low-integrity field values are bracketed:*

$$\begin{aligned}\vdash^{\alpha}_{[wf]} M \overset{def.}{\iff} \vdash_{wf} M \wedge (M(m^S) = \{\overrightarrow{x_i = v_i}\} \wedge S = \{\overrightarrow{x_i : \tau_i}\}_s \wedge \alpha \not\preccurlyeq \mathsf{integ}(\tau_i)\\ \Rightarrow \exists v'.\ v_i = [v'])\end{aligned}$$

Definition 7 (Well-formed $[\lambda_{persist}]$ configuration)**.** *A $[\lambda_{persist}]$ configuration $\langle e, M\rangle$ is well-formed if it is a well-formed $\lambda_{persist}$ configuration with a well-formed $[\lambda_{persist}]$ memory. A configuration is well-formed in a non-adversarial setting if it is well-formed in the presence of a $\bot$ adversary.*

$$\vdash^{\alpha}_{[wf]} \langle e, M\rangle \overset{def.}{\iff} \vdash^{\alpha}_{[wf]} M \wedge \vdash^{\alpha}_{wf} \langle e, M\rangle$$
$$\vdash_{[wf]} \langle e, M\rangle \overset{def.}{\iff} \vdash^{\bot}_{[wf]} \langle e, M\rangle$$

3.7.2 Completeness of $[\lambda_{persist}]$ evaluation

For $[\lambda_{persist}]$ to be adequate for reasoning about referential security properties of $\lambda_{persist}$, we must be able to simulate any $\lambda_{persist}$ execution in $[\lambda_{persist}]$. This is a completeness property. To do this, we first need to define what it means for a $[\lambda_{persist}]$ configuration to simulate a $\lambda_{persist}$ configuration. This involves defining a correspondence on expressions and heaps between the two languages.

We write $e_1 \lesssim e_2$ to denote that the $\lambda_{persist}$ term e_1 corresponds to the $[\lambda_{persist}]$ term e_2. The terms correspond if erasing brackets from e_2 produces e_1.

Definition 8 (Correspondence between $[\lambda_{persist}]$ and $\lambda_{persist}$ expressions)**.** *A $[\lambda_{persist}]$ expression e is related to a $\lambda_{persist}$ expression e', written $\vdash e \lesssim e'$, if the two are equal when brackets in e are removed. This is formally defined by the following induction rules.*

$$\frac{e = e}{\vdash e \lesssim e} \qquad \frac{\vdash e \lesssim e'}{\vdash [e] \lesssim e'} \qquad \frac{\vdash e \lesssim e'}{\vdash \lambda(x{:}\tau)[pc; \mathcal{H}].\, e \lesssim \lambda(x{:}\tau)[pc; \mathcal{H}].\, e'}$$

$$\frac{\vdash v_i \lesssim u_i \ ^{(\forall i)}}{\vdash v_1\ v_2 \lesssim u_1\ u_2} \qquad \frac{\vdash e_i \lesssim e_i' \ ^{(\forall i)}}{\vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \lesssim \mathsf{if}\ e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'} \qquad \frac{\vdash v_i \lesssim u_i \ ^{(\forall i)}}{\vdash \{\overrightarrow{x_i = v_i}\}^S \lesssim \{\overrightarrow{x_i = u_i}\}^S}$$

(rules continued on next page)

$$\frac{\vdash v \lesssim u}{\vdash v.x \lesssim u.x} \qquad \frac{\vdash v_i \lesssim u_i \ ^{(\forall i)}}{\vdash v_1.x := v_2 \lesssim u_1.x := u_2} \qquad \frac{\vdash e \lesssim e'}{\vdash \mathsf{soft}\ e \lesssim \mathsf{soft}\ e'} \qquad \frac{\vdash e_i \lesssim e_i' \ ^{(\forall i)}}{\vdash e_1 \| e_2 \lesssim e_1' \| e_2'}$$

$$\frac{\vdash e_i \lesssim e_i' \ ^{(\forall i)}}{\vdash \mathsf{exists}\ e_1\ \mathsf{as}\ x : e_2\ \mathsf{else}\ e_3 \lesssim \mathsf{exists}\ e_1'\ \mathsf{as}\ x : e_2'\ \mathsf{else}\ e_3'}$$

$$\frac{\vdash e_i \lesssim e_i' \ ^{(\forall i)}}{\vdash \mathsf{try}\ e_1\ \mathsf{catch}\ p\text{:}\ e_2 \lesssim \mathsf{try}\ e_1'\ \mathsf{catch}\ p\text{:}\ e_2'} \qquad \frac{\vdash e_i \lesssim e_i' \ ^{(\forall i)}}{\vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \lesssim \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2'}$$

This correspondence on expressions naturally induces a correspondence on heaps, and the two together give the correspondence on configurations.

Definition 9 (Correspondence between $[\lambda_{persist}]$ and $\lambda_{persist}$ memories)**.** *A $[\lambda_{persist}]$ memory M_1 is related to a $\lambda_{persist}$ memory M_2, written $\vdash M_1 \lesssim M_2$, if they map the same set of locations, and the memories map each location to values that are related.*

$$\begin{aligned} \vdash M_1 \lesssim M_2 \overset{def.}{\iff} & \mathsf{dom}(M_1) = \mathsf{dom}(M_2) \\ & \wedge \forall m^S \in \mathsf{dom}(M_1).\ M_1(m^S) = M_2(m^S) = \bot \\ & \qquad \vee \vdash M_1(m^S) \lesssim M_2(m^S) \end{aligned}$$

Definition 10 (Correspondence between $[\lambda_{persist}]$ and $\lambda_{persist}$ configurations)**.** *A $[\lambda_{persist}]$ configuration $\langle e_1, M_1\rangle$ is related to a $\lambda_{persist}$ configuration $\langle e_2, M_2\rangle$, written $\vdash \langle e_1, M_1\rangle \lesssim \langle e_2, M_2\rangle$, if both the expressions and the memories are related:*

$$\vdash \langle e_1, M_1\rangle \lesssim \langle e_2, M_2\rangle \overset{def.}{\iff} \vdash e_1 \lesssim e_2 \wedge \vdash M_1 \lesssim M_2$$

We can now state and prove the completeness of $[\lambda_{persist}]$.

Lemma 1 (Completeness of $[\lambda_{persist}]$ with respect to $\lambda_{persist}$)**.** *Let $\langle e_1, M_1\rangle$ be a well-formed $\lambda_{persist}$ configuration with e_1 well-typed. Let $\langle e_2, M_2\rangle$ be a well-formed $[\lambda_{persist}]$*

configuration corresponding to $\langle e_1, M_1\rangle$, with e_2 well-typed. If $\langle e_1, M_1\rangle$ takes a $\rightarrow$ transition to $\langle e_1', M_1'\rangle$, then there exists a configuration $\langle e_2', M_2'\rangle$ corresponding to $\langle e_1', M_1'\rangle$ such that $\langle e_2, M_2\rangle \rightarrow_\alpha^* \langle e_2', M_2'\rangle$.

$$\begin{aligned}
&\vdash_{wf}^{\alpha} \langle e_1, M_1\rangle \wedge \varnothing; pc; \mathcal{H} \vdash e_1 : \tau, \mathcal{X} \\
&\wedge \vdash_{[wf]}^{\alpha} \langle e_2, M_2\rangle \wedge \varnothing; pc; \mathcal{H} \vdash e_2 : \tau, \mathcal{X} \\
&\wedge \vdash \langle e_1, M_1\rangle \lesssim \langle e_2, M_2\rangle \wedge \langle e_1, M_1\rangle \rightarrow \langle e_1', M_1'\rangle \\
&\qquad \Rightarrow \exists e_2', M_2'. \langle e_2, M_2\rangle \rightarrow_\alpha^* \langle e_2', M_2'\rangle \wedge \langle e_1', M_1'\rangle \lesssim \langle e_2', M_2'\rangle
\end{aligned}$$

Proof. Induction on the derivation of $\langle e_1, M_1\rangle \rightarrow_\alpha \langle e_1', M_1'\rangle$. □

3.7.3 Soundness of [$\lambda_{persist}$] type system

We prove the type system sound via the usual method of proving type preservation (Lemma 9) and progress (Lemma 13). Because we are only concerned with well-formed configurations, it is important to know that they are preserved by the operational semantics. This is captured by Lemma 10.

We first show some preliminary results for weakening the typing context.

Lemma 2 (Type-environment weakening)**.** *Extra type assumptions can be safely added to typing contexts:* $\Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \wedge x \notin \mathsf{FV}(e) \Rightarrow \Gamma, x{:}\tau'; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}$.

Proof. By induction on the derivation of $\Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}$. □

Lemma 3 (pc weakening)**.** *The pc label in a typing context can be safely raised.*

$$\vdash pc \preccurlyeq pc' \wedge \Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \Rightarrow \Gamma; pc'; \mathcal{H} \vdash e : \tau, \mathcal{X}$$

Proof. By induction on the derivation of $\Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}$.

Rules T-BOOL, T-UNIT, T-LOC, T-BOTTOM, and T-VAR are trivial base cases. Rule T-ABS follows from the transitivity of $\preccurlyeq$. Rules T-SOFT, T-PARALLEL, and T-SUBSUME follow from the induction hypothesis.

Rules T-RECORD, T-SELECT, T-SOFT-SELECT, T-ASSIGN, T-SOFT-ASSIGN, T-EXISTS, T-APP, T-LET, T-TRY, T-IF, and T-BRACKET follow from the induction hypothesis and the transitivity of $\preccurlyeq$. In rules T-ASSIGN and T-SOFT-ASSIGN, we need to show $\vdash \tau \sqcap pc' \sqcap p \leq \tau_c$. By assumption, we have $\vdash \tau \sqcap pc \sqcap p \leq \tau_c$. Let $b_w = \tau$. Then using S3, we can show

$$\frac{\vdash b \leq b \qquad \vdash w \sqcap pc \sqcap p \preccurlyeq w \sqcap pc' \sqcap p}{\vdash \tau \sqcap pc' \sqcap p \leq \tau \sqcap pc \sqcap p} \quad ,$$

and the result follows from transitivity of subtyping. □

Lemma 4 (Handler weakening)**.** *Extra handler assumptions can be safely added to the typing context.*

$$\vdash \mathcal{H}' \preccurlyeq \mathcal{H} \wedge \Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \Rightarrow \Gamma; pc; \mathcal{H}' \vdash e : \tau, \mathcal{X}$$

Proof. By induction on the derivation of $\Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}$.

Rules T-BOOL, T-UNIT, T-LOC, T-BOTTOM, T-VAR, T-ABS, and T-PARALLEL are trivial base cases.

Rules T-SOFT, T-RECORD, T-SELECT, T-ASSIGN, T-SOFT-SELECT, T-SOFT-ASSIGN, T-EXISTS, T-APP, T-LET, T-IF, T-SUBSUME, and T-BRACKET follow from the induction hypothesis.

Rule T-TRY follows from the induction hypothesis and the fact that $\mathcal{H}' \cup \{p\} \preccurlyeq \mathcal{H} \cup \{p\}$. □

Corollary 5 summarizes these results.

Corollary 5 (Context weakening). *Suppose* $\Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}$ *with* $x \notin \mathsf{FV}(e)$, $pc \preccurlyeq pc'$ *and* $\mathcal{H}' \preccurlyeq \mathcal{H}$. *Then*

$$\Gamma, x{:}\tau'; pc'; \mathcal{H}' \vdash e : \tau, \mathcal{X}$$

We can now prove the substitution lemma.

Lemma 6 (Substitution). $\Gamma, x{:}\tau'; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \wedge \varnothing; pc; \mathcal{H} \vdash v : \tau', \top \Rightarrow \Gamma; pc; \mathcal{H} \vdash e\{v/x\} : \tau, \mathcal{X}$.

Proof. By induction on the derivation of $\Gamma, x{:}\tau'; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}$. Note that since v was typed in an empty type assignment ($\Gamma = \varnothing$), we must have $\mathsf{FV}(v) = \varnothing$.

Rules T-Bool, T-Unit, T-Loc, and T-Bottom are trivial base cases.

Rules T-Soft, T-Record, T-Select, T-Assign, T-Soft-Select, T-Soft-Assign, T-App, T-Parallel, T-Try, T-If, T-Subsume, and T-Bracket follow from the definition of substitution and the induction hypothesis.

Case T-Var:

Suppose $e = x$. Then $e\{v/x\} = v$ and $\tau' = \tau$ and the result holds in this case via Corollary 5 and T-Subsume. Alternatively, suppose $e = y \neq x$. In this case, $e\{v/x\} = e$ and the result holds trivially.

Case T-Abs ($e = \lambda(y{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1$):

If $y = x$, then $e\{v/x\} = e$ and the result holds trivially. Alternatively, suppose $y \neq x$. We have $\mathsf{FV}(v) = \varnothing$, so $e\{v/x\} = \lambda(y{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1\{v/x\}$. Let $\Gamma' = \Gamma, x{:}\tau'$. From the typing of e, we have

$$\frac{\Gamma, x{:}\tau', y{:}\tau_1; pc_1; \mathcal{H}_1 \vdash e_1 : \tau_2, \mathcal{H}_1 \qquad \vdash_{wf} (\tau_1 \xrightarrow{pc_1, \mathcal{H}_1} \tau_2)_\top : \mathsf{type} \qquad \vdash pc_1 \preccurlyeq pc}{\Gamma, x{:}\tau'; pc; \mathcal{H} \vdash \lambda(y{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1 : (\tau_1 \xrightarrow{pc_1, \mathcal{H}_1} \tau_2)_\top, \top\,,}$$

where $\tau = (\tau_1 \xrightarrow{pc_1, \mathcal{H}_1} \tau_2)_\top$ and $\mathcal{X} = \top$. So, we know $\Gamma, x{:}\tau', y{:}\tau_1; pc_1; \mathcal{H}_1 \vdash e_1 : \tau_2, \mathcal{H}_1$. Therefore, by the induction hypothesis, we have $\Gamma, y : \tau_1; pc_1; \mathcal{H}_1 \vdash e_1\{v/x\} : \tau_2, \mathcal{H}_1$. So the result holds in this case via an application of T-ABS:

$$\frac{\Gamma, y{:}\tau_1; pc_1; \mathcal{H}_1 \vdash e_1\{v/x\} : \tau_2, \mathcal{H}_1 \qquad \vdash_{wf} (\tau_1 \xrightarrow{pc_1, \mathcal{H}_1} \tau_2)_\top : \mathsf{type} \qquad \vdash pc_1 \preccurlyeq pc}{\Gamma; pc; \mathcal{H} \vdash \lambda(y{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1\{v/x\} : (\tau_1 \xrightarrow{pc_1, \mathcal{H}_1} \tau_2)_\top, \top\,.}$$

Case T-EXISTS (exists u as $y : e_1$ else e_2):

From the typing of e, we have

$$\frac{\begin{array}{c}\Gamma, x{:}\tau'; pc; \mathcal{H} \vdash u : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_r)_w, \top \\ \vdash \mathsf{auth}^+(r) \preccurlyeq pc \sqcap w \qquad w' = \mathsf{auth}^-(r) \sqcap \mathsf{persist}(r) \sqcap w \\ \Gamma, x{:}\tau', y{:}(\{\overrightarrow{x_i : \tau_i}\}_r)_w; pc'; \mathcal{H} \vdash e_1 : \tau'', \mathcal{X}_1 \\ \Gamma, x{:}\tau'; pc'; \mathcal{H} \vdash e_2 : \tau'', \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau'') \preccurlyeq pc'\end{array}}{\Gamma, x{:}\tau'; pc; \mathcal{H} \vdash \mathsf{exists}\ u\ \mathsf{as}\ y : e_1\ \mathsf{else}\ e_2 : \tau'' \sqcap w', \mathcal{X}_1 \sqcap \mathcal{X}_2\,,}$$

where $pc' = pc \sqcap w'$, $\tau = \tau'' \sqcap w'$, and $\mathcal{X} = \mathcal{X}_1 \sqcap \mathcal{X}_2$. By the induction hypothesis, we therefore have

$$\Gamma; pc; \mathcal{H} \vdash u\{v/x\} : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_r)_w, \top \tag{3.3}$$

$$\Gamma; pc'; \mathcal{H} \vdash e_2\{v/x\} : \tau'', \mathcal{X}_2 \tag{3.4}$$

Suppose $y = x$. Then $e\{v/x\} = \mathsf{exists}\ u\{v/x\}\ \mathsf{as}\ y : e_1\ \mathsf{else}\ e_2\{v/x\}$, so from (3.3) and (3.4), the result holds in this case via an application of T-EXISTS:

$$\frac{\begin{array}{c}\Gamma; pc; \mathcal{H} \vdash u\{v/x\} : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_r)_w, \top \\ \vdash \mathsf{auth}^+(r) \preccurlyeq pc \sqcap w \qquad w' = \mathsf{auth}^-(r) \sqcap \mathsf{persist}(r) \sqcap w \\ \Gamma, y{:}(\{\overrightarrow{x_i : \tau_i}\}_r)_w; pc'; \mathcal{H} \vdash e_1 : \tau'', \mathcal{X}_1 \\ \Gamma; pc'; \mathcal{H} \vdash e_2\{v/x\} : \tau'', \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau'') \preccurlyeq pc'\end{array}}{\Gamma; pc; \mathcal{H} \vdash \mathsf{exists}\ u\{v/x\}\ \mathsf{as}\ y : e_1\ \mathsf{else}\ e_2\{v/x\} : \tau'' \sqcap w', \mathcal{X}_1 \sqcap \mathcal{X}_2\,.}$$

Alternatively, suppose $y \neq x$. We have $\mathsf{FV}(v) = \varnothing$, so $e\{v/x\} = \mathsf{exists}\ u\{v/x\}\ \mathsf{as}\ y : e_1\{v/x\}\ \mathsf{else}\ e_2\{v/x\}$. From the typing of e, we know $\Gamma, x{:}\tau', y{:}(\{\overrightarrow{x_i : \tau_i}\}_r)_w; pc'; \mathcal{H} \vdash e_1 : \tau'', \mathcal{X}_1$. By the induction hypothesis, we have $\Gamma, y{:}(\{\overrightarrow{x_i : \tau_i}\}_r)_w; pc'; \mathcal{H} \vdash e_1\{v/x\} : \tau'', \mathcal{X}_1$. From this, (3.3), and (3.4), the result holds in this case via an application of T-EXISTS:

$$\frac{\begin{array}{c}\Gamma; pc; \mathcal{H} \vdash u\{v/x\} : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_r)_w, \top \\ \vdash \mathsf{auth}^+(r) \preccurlyeq pc \sqcap w \qquad w' = \mathsf{auth}^-(r) \sqcap \mathsf{persist}(r) \sqcap w \\ \Gamma, y{:}(\{\overrightarrow{x_i : \tau_i}\}_r)_w; pc'; \mathcal{H} \vdash e_1\{v/x\} : \tau'', \mathcal{X}_1 \\ \Gamma; pc'; \mathcal{H} \vdash e_2\{v/x\} : \tau'', \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau'') \preccurlyeq pc'\end{array}}{\Gamma; pc; \mathcal{H} \vdash \mathsf{exists}\ u\{v/x\}\ \mathsf{as}\ y : e_1\{v/x\}\ \mathsf{else}\ e_2\{v/x\} : \tau'' \sqcap w', \mathcal{X}_1 \sqcap \mathcal{X}_2}.$$

Case T-LET ($e = \mathsf{let}\ y = e_1\ \mathsf{in}\ e_2$):

From the typing of e, we have

$$\frac{\begin{array}{c}\Gamma, x{:}\tau'; pc; \mathcal{H} \vdash e_1 : \tau_1, \mathcal{X}_1 \qquad \vdash \mathsf{auth}^+(\tau_1) \preccurlyeq pc \qquad w = (\bigsqcap \mathcal{X}_1) \sqcap \mathsf{integ}(\tau_1) \\ pc' = pc \sqcap w \qquad \Gamma, x{:}\tau', y{:}\tau_1; pc'; \mathcal{H} \vdash e_2 : \tau_2, \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau_2) \preccurlyeq pc'\end{array}}{\Gamma, x{:}\tau'; pc; \mathcal{H} \vdash \mathsf{let}\ y = e_1\ \mathsf{in}\ e_2 : \tau_2 \sqcap w, \mathcal{X}_1 \sqcap \mathcal{X}_2},$$

where $\tau = \tau_2 \sqcap w$ and $\mathcal{X} = \mathcal{X}_1 \sqcap \mathcal{X}_2$. By the induction hypothesis, we therefore have

$$\Gamma; pc; \mathcal{H} \vdash e_1\{v/x\} : \tau_1, \mathcal{X}_1. \tag{3.5}$$

Suppose $y = x$. Then $e\{v/x\} = \mathsf{let}\ x = e_1\{v/x\}\ \mathsf{in}\ e_2$, so from (3.5), the result holds in this case via an application of T-LET:

$$\frac{\begin{array}{c}\Gamma; pc; \mathcal{H} \vdash e_1\{v/x\} : \tau_1, \mathcal{X}_1 \qquad \vdash \mathsf{auth}^+(\tau_1) \preccurlyeq pc \qquad w = (\bigsqcap \mathcal{X}_1) \sqcap \mathsf{integ}(\tau_1) \\ pc' = pc \sqcap w \qquad \Gamma, y{:}\tau_1; pc'; \mathcal{H} \vdash e_2 : \tau_2, \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau_2) \preccurlyeq pc'\end{array}}{\Gamma; pc; \mathcal{H} \vdash \mathsf{let}\ y = e_1\{v/x\}\ \mathsf{in}\ e_2 : \tau_2 \sqcap w, \mathcal{X}_1 \sqcap \mathcal{X}_2}.$$

Alternatively, suppose $y \neq x$. We have $\mathsf{FV}(v) = \varnothing$, so $e\{v/x\} = \mathsf{let}\ y = e_1\{v/x\}\ \mathsf{in}\ e_2\{v/x\}$. From the typing of e, we know $\Gamma, x : \tau', y : \tau_1; pc'; \mathcal{H} \vdash e_2 : \tau_2, \mathcal{X}_2$. By the induction hypothesis, we have $\Gamma, y : \tau_1; pc'; \mathcal{H} \vdash e_2\{v/x\} : \tau_2, \mathcal{X}_2$. From this and (3.5), the result holds in this case via an application of T-LET:

$$\frac{\Gamma; pc; \mathcal{H} \vdash e_1\{v/x\} : \tau_1, \mathcal{X}_1 \qquad \vdash \mathsf{auth}^+(\tau_1) \preccurlyeq pc \qquad w = (\bigsqcap \mathcal{X}_1) \sqcap \mathsf{integ}(\tau_1) \\ pc' = pc \sqcap w \qquad \Gamma, y : \tau_1; pc'; \mathcal{H} \vdash e_2\{v/x\} : \tau_2, \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau_2) \preccurlyeq pc'}{\Gamma; pc; \mathcal{H} \vdash \mathsf{let}\ y = e_1\{v/x\}\ \mathsf{in}\ e_2\{v/x\} : \tau_2 \sqcap w, \mathcal{X}_1 \sqcap \mathcal{X}_2} .$$

□

Lemma 7 (Effect bound)**.** *The effect of a well-typed expression is bounded from below by its handler environment.*

$$\Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \Rightarrow \mathcal{H} \preccurlyeq \mathcal{X}$$

Proof. By induction on the derivation of $\Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}$. Rule T-TRY relies on the easily proved fact that if $\mathcal{H} \cup \{p\} \preccurlyeq \mathcal{X}_1$, then $\mathcal{H} \preccurlyeq \mathcal{X}_1/p$. □

Lemma 8 (Value typing)**.** *The handler environment is irrelevant for typing non-bottom values. Such a value v can also be typed with any pc that has the authority of the references that appear in v, and can have any effect bounded from below by the handler environment. Formally,*

$$\begin{aligned} &\Gamma; pc; \mathcal{H} \vdash v : \tau, \mathcal{X} \wedge (\forall p.\ v \neq \bot_p \wedge v \neq [\bot_p]) \\ &\wedge \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc' \wedge \vdash \mathcal{H}' \preccurlyeq \mathcal{X}' \\ &\qquad \Rightarrow \Gamma; pc'; \mathcal{H}' \vdash v : \tau, \mathcal{X}' \end{aligned}$$

Proof. By induction on the derivation of $\Gamma; pc; \mathcal{H} \vdash v : \tau, \mathcal{X}$. □

We are now ready to prove type preservation for $[\lambda_{persist}]$.

Lemma 9 (Type preservation)**.** *Let M be a well-formed memory. Let e be an expression with type τ and effect $\mathcal{X}$. Let $\alpha \in \mathcal{L}$. If the configuration $\langle e, M\rangle$ takes an $\rightarrow_\alpha$ transition, then the new expression e' will also have type τ and effect $\mathcal{X}$:*

$$\begin{aligned}&\vdash^{\alpha}_{[wf]} M \wedge \varnothing; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}\\ &\wedge \langle e, M\rangle \rightarrow_\alpha \langle e', M'\rangle\\ &\qquad \Rightarrow \varnothing; pc; \mathcal{H} \vdash e' : \tau, \mathcal{X}.\end{aligned}$$

Proof. By induction on the derivation of $\varnothing; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}$.

Given $\langle e, M\rangle \rightarrow_\alpha \langle e', M'\rangle$, the proof proceeds by cases according to the evaluation rules. For cases GC, α-CREATE, α-ASSIGN, and α-FORGET, we have $e = e'$, so the result follows trivially.

By Lemma 7, we have $\mathcal{H} \preccurlyeq \mathcal{X}$.

Case CREATE ($\left\langle \{\overrightarrow{x_i = v_i}\}^S, M\right\rangle \rightarrow_\alpha \left\langle m^S, M[m^S \mapsto \{\overrightarrow{x_i = v_i'}\}]\right\rangle$, where m is fresh, $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$, and $v_i' = v_i \blacktriangleright_\alpha \tau_i$):

We have $\varnothing; pc; \mathcal{H} \vdash \{\overrightarrow{x_i = v_i}\}^S : R_\top, \top$ with $R = \{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)}$ and $\vdash_{wf} S$: rectype. We need to show $\varnothing; pc; \mathcal{H} \vdash m^S : R_\top, \top$. This is given by trivial application of T-LOC.

Case APPLY ($\langle (\lambda(x{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1)\ v, M\rangle \rightarrow_\alpha \langle e_1\{v/x\}, M\rangle$):

From the typing of e, we have

$$\varnothing; pc; \mathcal{H} \vdash \lambda(x{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1 : (\tau_1 \xrightarrow{pc_1, \mathcal{H}_1} \tau)_\top, \top \tag{3.6}$$

and

$$\varnothing; pc; \mathcal{H} \vdash v : \tau_1, \top \tag{3.7}$$

with $\vdash \mathcal{H} \preccurlyeq \mathcal{H}_1$ and $\mathcal{H}_1 = \mathcal{X}$. We need to show $\varnothing; pc; \mathcal{H} \vdash e_1\{v/x\} : \tau, \mathcal{H}_1$. From the derivation of (3.6), we know

$$x{:}\tau_1; pc_1; \mathcal{H}_1 \vdash e_1 : \tau, \mathcal{H}_1$$

with $\vdash pc_1 \preccurlyeq pc$. Applying Corollary 5 to this, we get

$$x{:}\tau_1; pc; \mathcal{H} \vdash e_1 : \tau, \mathcal{H}_1.$$

The result follows from this and (3.7) via Lemma 6.

Case SELECT ($\langle m^S.x_c, M\rangle \rightarrow_\alpha \langle v_c \blacktriangleright_\alpha p, M\rangle$, where $M(m^S) = \{\overrightarrow{x_i = v_i}\}$ and $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$):

We have $\varnothing; pc; \mathcal{H} \vdash m^S.x_c : \tau_c \sqcap p, p$ and need to show $\varnothing; pc; \mathcal{H} \vdash v_c \blacktriangleright_\alpha p : \tau_c \sqcap p, p$.

From the typing of e, we know $\varnothing; pc; \mathcal{H} \vdash m^S : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top$ and $\vdash a \preccurlyeq pc$ with $\vdash_{wf} S$: rectype. Therefore, the following holds:

- $\vdash \mathsf{auth}^+(\tau_c) \preccurlyeq a \preccurlyeq pc$ and
- $\vdash \mathsf{auth}^+(\tau_c) \preccurlyeq p$.

So, we know $\vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc \sqcap p$.

By Lemma 7, we have $\vdash \mathcal{H} \preccurlyeq p$. Since M is well-formed, we also know that $\varnothing; \top; \top \vdash v_c : \tau_c, \top$, so by Lemma 8, we have

$$\varnothing; pc \sqcap p; \mathcal{H} \vdash v_c : \tau_c, p. \tag{3.8}$$

Suppose $\vdash \alpha \preccurlyeq p$ or v_c is bracketed. Then $v_c \blacktriangleright_\alpha p = v_c$ and the result follows from (3.8) via Corollary 5 and T-SUBSUME.

Otherwise, $\alpha \not\preccurlyeq p$ and v_c is unbracketed. So $v_c \blacktriangleright_\alpha p = [v_c]$, and the result follows via T-BRACKET:

$$\frac{\varnothing; pc \sqcap p; \mathcal{H} \vdash v_c : \tau_c, p \qquad \alpha \not\preccurlyeq p \qquad \vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc \sqcap p}{\varnothing; pc; \mathcal{H} \vdash [v_c] : \tau_c \sqcap p, p}\ .$$

Case DANGLE-SELECT ($\langle m^S.x_c, M\rangle \to_\alpha \langle \bot_p \blacktriangleright_\alpha p, M\rangle$, where $M(m^S) = \bot$ and $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$):

We have $\varnothing; pc; \mathcal{H} \vdash m^S.x_c : \tau_c \sqcap p, p$ and need to show $\varnothing; pc; \mathcal{H} \vdash \bot_p \blacktriangleright_\alpha p : \tau_c \sqcap p, p$.

From the typing of e, we know $\varnothing; pc; \mathcal{H} \vdash m^S : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top$ and $\vdash a \preccurlyeq pc$ with $\vdash_{wf} S$: rectype. Therefore, the following holds:

- $\vdash \mathsf{auth}^+(\tau_c) \preccurlyeq a \preccurlyeq pc$ and
- $\vdash \mathsf{auth}^+(\tau_c) \preccurlyeq p$.

So, we know $\vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc \sqcap p$.

By Lemma 7, we know $\vdash \mathcal{H} \preccurlyeq p$, so by T-BOTTOM, we have

$$\varnothing; pc \sqcap p; \mathcal{H} \vdash \bot_p : \tau_c, p. \tag{3.9}$$

Suppose $\vdash \alpha \preccurlyeq p$. Then $\bot_p \blacktriangleright_\alpha p = \bot_p$ and the result follows from (3.8) via Corollary 5 and T-SUBSUME.

Otherwise, $\alpha \not\preccurlyeq p$. So $\bot_p \blacktriangleright_\alpha p = [\bot_p]$, and the result follows via T-BRACKET:

$$\frac{\varnothing; pc \sqcap p; \mathcal{H} \vdash \bot_p : \tau_c, p \qquad \alpha \not\preccurlyeq p \qquad \vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc \sqcap p}{\varnothing; pc; \mathcal{H} \vdash [\bot_p] : \tau_c \sqcap p, p}\ .$$

Case SOFT-SELECT ($\langle(\mathsf{soft}\ m^S).x_c, M\rangle \to_\alpha \langle v \blacktriangleright_\alpha (a \sqcap p), M\rangle$, where $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$ and $\langle m^S.x_c, M\rangle \xrightarrow{\mathsf{e}} \langle v, M\rangle$):

From the typing of e, we have $\varnothing; pc; \mathcal{H} \vdash (\mathsf{soft}\ m^S).x_c : \tau_c \sqcap p', p$ and $\vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc$, where $p' = a \sqcap p$. So $\tau = \tau_c \sqcap p'$ and $\mathcal{X} = p$. We need to show $\varnothing; pc; \mathcal{H} \vdash v \blacktriangleright_\alpha p' : \tau_c \sqcap p', p$.

From the typing of e, it follows that $\vdash_{wf} S : \mathsf{rectype}$, and therefore, we know $\vdash \mathsf{auth}^+(\tau_c) \preccurlyeq p'$. Since we also know $\vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc$, we therefore have $\vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc \sqcap p'$.

We proceed by cases according to the evaluation rules for $\langle m^S.x_c, M\rangle \xrightarrow{\mathsf{e}} \langle v, M\rangle$.

Sub-case SELECT ($\langle m^S.x_c, M\rangle \xrightarrow{\mathsf{e}} \langle v_c \blacktriangleright_\alpha p, M\rangle$, where $M(m^S) = \{\overrightarrow{x_i = v_i}\}$):

We have $v = v_c \blacktriangleright_\alpha p$.

By Lemma 7, we have $\vdash \mathcal{H} \preccurlyeq p$. Since M is well-formed, we also know that $\varnothing; \top; \top \vdash v_c : \tau_c, \top$, so by Lemma 8, we have

$$\varnothing; pc \sqcap p'; \mathcal{H} \vdash v_c : \tau_c, p. \tag{3.10}$$

Suppose $\vdash \alpha \preccurlyeq p'$. Then $\vdash \alpha \preccurlyeq p$, so $v \blacktriangleright_\alpha p' = v = v_c \blacktriangleright_\alpha p = v_c$. Similarly, if v_c is bracketed, then $v \blacktriangleright_\alpha p' = v_c$. In these cases, the result follows from (3.10) via Corollary 5 and T-SUBSUME.

Otherwise, $\alpha \not\preccurlyeq p'$ and v_c is unbracketed. So $v \blacktriangleright_\alpha p' = [v_c]$ and the result follows via T-BRACKET:

$$\frac{\varnothing; pc \sqcap p'; \mathcal{H} \vdash v_c : \tau_c, p \qquad \alpha \not\preccurlyeq p' \qquad \vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc \sqcap p'}{\varnothing; pc; \mathcal{H} \vdash [v_c] : \tau_c \sqcap p', p}\ .$$

Sub-case DANGLE-SELECT ($\langle m^S.x_c, M\rangle \xrightarrow{\mathsf{e}} \langle \bot_p \blacktriangleright_\alpha p, M\rangle$):

We have $v = \bot_p \blacktriangleright_\alpha p$.

By Lemma 7, we know $\vdash \mathcal{H} \preccurlyeq p$, and we have $\vdash p' \preccurlyeq p$ by definition, so by T-BOTTOM, we have $\varnothing; pc \sqcap p'; \mathcal{H} \vdash \bot_p : \tau_c \sqcap p', p$.

Suppose $\vdash \alpha \preccurlyeq p'$. Then $\vdash \alpha \preccurlyeq p$, so $v \blacktriangleright_\alpha p' = v = \bot_p \blacktriangleright_\alpha p = \bot_p$, and the result follows via Corollary 5.

Otherwise, $\alpha \not\preccurlyeq p'$, so $v \blacktriangleright_\alpha p' = [\bot_p]$, and the result follows via T-BRACKET:

$$\frac{\varnothing; pc \sqcap p'; \mathcal{H} \vdash \bot_p : \tau_c \sqcap p', p \qquad \alpha \not\preccurlyeq p' \qquad \vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc \sqcap p'}{\varnothing; pc; \mathcal{H} \vdash [\bot_p] : \tau_c \sqcap p', p}\ .$$

Case ASSIGN ($\langle m^S.x_c := v, M\rangle \rightarrow_\alpha \langle * \blacktriangleright_\alpha p, M[m^S.x_c \mapsto v']\rangle$, where $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$):

From the typing of e, we have $\varnothing; pc; \mathcal{H} \vdash m^S.x_c := v : \mathbf{1}, p$ and need to show $\varnothing; pc; \mathcal{H} \vdash * \blacktriangleright_\alpha p : \mathbf{1}, p$. By Lemma 7, we have $\vdash \mathcal{H} \preccurlyeq p$, so from T-UNIT and T-SUBSUME, we have $\varnothing; pc \sqcap p; \mathcal{H} \vdash * : \mathbf{1}, p$.

If $\vdash \alpha \preccurlyeq p$, then $* \blacktriangleright_\alpha p = *$, and the result follows by Corollary 5.

Otherwise, $\alpha \not\preccurlyeq p$ and $* \blacktriangleright_\alpha p = [*]$, and the result follows via T-BRACKET.

Case DANGLE-ASSIGN ($\langle m^S.x_c := v, M\rangle \rightarrow_\alpha \langle \bot_p \blacktriangleright_\alpha p, M\rangle$, where $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$):

From the typing of e, we have $\varnothing; pc; \mathcal{H} \vdash m^S.x_c := v : \mathbf{1}, p$ and we need to show $\varnothing; pc; \mathcal{H} \vdash \bot_p \blacktriangleright_\alpha p : \mathbf{1}, p$. By Lemma 7, we have $\vdash \mathcal{H} \preccurlyeq p$, so by T-BOTTOM, we have $\varnothing; pc \sqcap p; \mathcal{H} \vdash \bot_p : \mathbf{1}, p$.

If $\vdash \alpha \preccurlyeq p$, then $\bot_p \blacktriangleright_\alpha p = \bot_p$, and the result follows by Corollary 5.

Otherwise, $\alpha \not\preccurlyeq p$ and $\bot_p \blacktriangleright_\alpha p = [\bot_p]$, and the result follows via T-BRACKET.

Case SOFT-ASSIGN ($\langle(\mathsf{soft}\ m^S).x_c := v, M\rangle \rightarrow_\alpha \langle v' \blacktriangleright_\alpha (a \sqcap p), M'\rangle$, where $\langle m^S.x_c := v, M\rangle \xrightarrow{\mathsf{e}} \langle v', M'\rangle$ and $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$):

From the typing of e, we have $\varnothing; pc; \mathcal{H} \vdash (\mathsf{soft}\ m^S).x_c := v : \mathbf{1}, p$. By Lemma 7, we have $\vdash \mathcal{H} \preccurlyeq p$. Let $p' = a \sqcap p$. We need to show $\varnothing; pc; \mathcal{H} \vdash v' \blacktriangleright_\alpha p' : \mathbf{1}, p$. We proceed by cases according to the evaluation rules for $\langle m^S.x_c := v, M\rangle \xrightarrow{\mathsf{e}} \langle v', M'\rangle$.

Sub-case ASSIGN ($\langle m^S.x_c := v, M\rangle \xrightarrow{\mathsf{e}} \langle * \blacktriangleright_\alpha p, M'\rangle$):

We have $v' = * \blacktriangleright_\alpha p$.

If $\vdash \alpha \preccurlyeq p'$, then $\vdash \alpha \preccurlyeq p$, so $v' \blacktriangleright_\alpha p' = v' = * \blacktriangleright_\alpha p = *$. The result follows from T-UNIT and T-SUBSUME.

Otherwise, $\alpha \not\preccurlyeq p'$, so $v' \blacktriangleright_\alpha p' = [*]$. The result follows from T-UNIT, T-BRACKET, and T-SUBSUME.

Sub-case DANGLE-ASSIGN ($\langle m^S.x_c := v, M\rangle \xrightarrow{\mathsf{e}} \langle \bot_p \blacktriangleright_\alpha p, M\rangle$):

We have $v' = \bot_p \blacktriangleright_\alpha p$.

If $\vdash \alpha \preccurlyeq p'$, then $\vdash \alpha \preccurlyeq p$, so $v' \blacktriangleright_\alpha p' = v' = \bot_p \blacktriangleright_\alpha p = \bot_p$. The result follows from T-BOTTOM.

Otherwise, $\alpha \not\preccurlyeq p'$, so $v' \blacktriangleright_\alpha p' = [\bot_p]$. The result follows from T-BOTTOM and T-BRACKET.

Case EXISTS-TRUE ($\langle \mathsf{exists\ soft}\ m^S\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2, M\rangle \to_\alpha \langle (e_1\{m^S/x\}) \blacktriangleright_\alpha w, M\rangle$, where $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$ and $w = a \sqcap p$):

From the typing of e, we have

$$\frac{\begin{array}{c}\varnothing; pc; \mathcal{H} \vdash \mathsf{soft}\ m^S : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top \qquad \vdash a \preccurlyeq pc \qquad w = a \sqcap p \\ x : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top; pc \sqcap w; \mathcal{H} \vdash e_1 : \tau', \mathcal{X}_1 \\ \varnothing; pc \sqcap w; \mathcal{H} \vdash e_2 : \tau', \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w\end{array}}{\varnothing; pc; \mathcal{H} \vdash \mathsf{exists\ soft}\ m^S\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2 : \tau' \sqcap w, \mathcal{X}_1 \sqcap \mathcal{X}_2}$$

where $\tau = \tau' \sqcap w$ and $\mathcal{X} = \mathcal{X}_1 \sqcap \mathcal{X}_2$. We need to show

$$\varnothing; pc; \mathcal{H} \vdash (e_1\{m^S/x\}) \blacktriangleright_\alpha w : \tau, \mathcal{X}.$$

By T-LOC, we have $\varnothing; pc \sqcap w; \mathcal{H} \vdash m^S : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top$. Therefore, by Lemma 6, we know $\varnothing; pc \sqcap w; \mathcal{H} \vdash e_1\{m^S/x\} : \tau', \mathcal{X}_1$. Also, by Lemma 7, we know $\vdash \mathcal{H} \preccurlyeq \mathcal{X}_1 \sqcap \mathcal{X}_2$, so by T-SUBSUME, we know

$$\varnothing; pc \sqcap w; \mathcal{H} \vdash e_1\{m^S/x\} : \tau, \mathcal{X}.$$

Suppose $\vdash \alpha \preccurlyeq w$. Then $(e_1\{m^S/x\}) \blacktriangleright_\alpha w = e_1\{m^S/x\}$. The result therefore follows by Corollary 5.

Otherwise, we have $\alpha \not\preccurlyeq w$, so $(e_1\{m^S/x\}) \blacktriangleright_\alpha w = [e_1\{m^S/x\}]$. The result therefore follows via T-BRACKET:

$$\frac{\varnothing; pc \sqcap w; \mathcal{H} \vdash e_1\{m^S/x\} : \tau' \sqcap w, \mathcal{X} \qquad \alpha \not\preccurlyeq w \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w}{\varnothing; pc; \mathcal{H} \vdash [e_1\{m^S/x\}] : \tau' \sqcap w, \mathcal{X}}$$

Case EXISTS-FALSE ($\langle \mathsf{exists\ soft}\ m^S\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2, M\rangle \rightarrow_\alpha \langle e_2 \blacktriangleright_\alpha w, M\rangle$, where $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$ and $w = a \sqcap p$):

From the typing of e, we have

$$\frac{\begin{array}{c}\varnothing; pc; \mathcal{H} \vdash \mathsf{soft}\ m^S : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top \qquad \vdash a \preccurlyeq pc \qquad w = a \sqcap p \\ x : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top; pc \sqcap w; \mathcal{H} \vdash e_1 : \tau', \mathcal{X}_1 \\ \varnothing; pc \sqcap w; \mathcal{H} \vdash e_2 : \tau', \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w\end{array}}{\varnothing; pc; \mathcal{H} \vdash \mathsf{exists\ soft}\ m^S\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2 : \tau' \sqcap w, \mathcal{X}_1 \sqcap \mathcal{X}_2}$$

where $\tau = \tau' \sqcap w$ and $\mathcal{X} = \mathcal{X}_1 \sqcap \mathcal{X}_2$. We need to show $\varnothing; pc; \mathcal{H} \vdash e_2 \blacktriangleright_\alpha w : \tau, \mathcal{X}$.

Suppose $\vdash \alpha \preccurlyeq w$. Then $e_2 \blacktriangleright_\alpha w = e_2$. The result follows from Corollary 5 and T-SUBSUME.

Otherwise, $\alpha \not\preccurlyeq w$, so $e_2 \blacktriangleright_\alpha w = [e_2]$. By Lemma 7, we know $\vdash \mathcal{H} \preccurlyeq \mathcal{X}_1 \sqcap \mathcal{X}_2$.

The result follows via T-BRACKET and T-SUBSUME:

$$\frac{\varnothing; pc \sqcap w; \mathcal{H} \vdash e_2 : \tau', \mathcal{X}_2 \qquad \alpha \notin w \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w}{\begin{array}{c}\varnothing; pc; \mathcal{H} \vdash [e_2] : \tau' \sqcap w, \mathcal{X}_2 \\ \vdash \mathcal{H} \preccurlyeq \mathcal{X}_1 \sqcap \mathcal{X}_2 \qquad \vdash \mathcal{X}_1 \sqcap \mathcal{X}_2 \preccurlyeq \mathcal{X}_2\end{array}}{\varnothing; pc; \mathcal{H} \vdash [e_2] : \tau' \sqcap w, \mathcal{X}_1 \sqcap \mathcal{X}_2}$$

Case TRY-VAL ($\langle \mathsf{try}\ v\ \mathsf{catch}\ p\text{:}\ e_1, M\rangle \to_\alpha \langle v, M\rangle$, where $v \neq \bot_{p'}$ and $v \neq [\bot_{p'}]$ for all p'):

From the typing of e, we have

$$\varnothing; pc; \mathcal{H}, p \vdash v : \tau, \mathcal{X}_1,$$

for some $\mathcal{X}_1$. Since v is a non-bottom value, the result $\varnothing; pc; \mathcal{H} \vdash v : \tau, \mathcal{X}$ follows via Lemma 8.

Case TRY-CATCH ($\langle \mathsf{try}\ \bot_p\ \mathsf{catch}\ p'\text{:}\ e_2, M\rangle \to_\alpha \langle e_2, M\rangle$, where $\vdash p' \preccurlyeq p$):

We have $p' \preccurlyeq p$, so from the typing of e, we have

$$\varnothing; pc \sqcap p \sqcap \mathsf{integ}(\tau_1); \mathcal{H} \vdash e_2 : \tau_1, \mathcal{X},$$

where $\tau = \tau_1 \sqcap p$. We need to show $\varnothing; pc; \mathcal{H} \vdash e_2 : \tau, \mathcal{X}$. This follows from Corollary 5 and T-SUBSUME.

Case TRY-ESC ($\langle \mathsf{try}\ \bot_p\ \mathsf{catch}\ p'\text{:}\ e_2, M\rangle \to_\alpha \langle \bot_p, M\rangle$, where $p' \not\preccurlyeq p$):

We have $p' \not\preccurlyeq p$, so from the typing of e, it follows that $\mathcal{H} \preccurlyeq \mathcal{X} \preccurlyeq p$. We need to show $\varnothing; pc; \mathcal{H} \vdash \bot_p : \tau, \mathcal{X}$. This follows from T-BOTTOM and T-SUBSUME.

Case PARALLEL-RESULT ($\langle v_1 \parallel v_2, M\rangle \to_\alpha \langle *, M\rangle$):

From the typing of e, we have $\tau = \mathbf{1}$ and $\mathcal{X} = \top$. The result $\varnothing; pc; \mathcal{H} \vdash * : \mathbf{1}, \top$ follows trivially from T-UNIT.

Case IF-TRUE ($\langle \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2), M\rangle \to_\alpha \langle e_1, M\rangle$):

From the typing of e, we have $\varnothing; pc; \mathcal{H} \vdash e_1 : \tau, \mathcal{X}_1$, where $\mathcal{X} \preccurlyeq \mathcal{X}_1$. We need to show $\varnothing; pc; \mathcal{H} \vdash e_1 : \tau, \mathcal{X}$, which follows from Lemma 7 and T-SUBSUME.

Case IF-FALSE ($\langle \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2, M\rangle \to_\alpha \langle e_2, M\rangle$):

From the typing of e, we have $\varnothing; pc; \mathcal{H} \vdash e_2 : \tau, \mathcal{X}_2$, where $\mathcal{X} \preccurlyeq \mathcal{X}_2$. We need to show $\varnothing; pc; \mathcal{H} \vdash e_2 : \tau, \mathcal{X}$, which follows from Lemma 7 and T-SUBSUME.

Case LET ($\langle \mathsf{let}\ x = v\ \mathsf{in}\ e_1, M\rangle \to_\alpha \langle e_1\{v/x\}, M\rangle$, where $v \notin \{\bot_p, [\bot_p]\}$ for all p):

Since v is a non-bottom value, by Lemma 8, it can be typed with $\top$ effect. From the typing of e, then, we have

$$\varnothing; pc; \mathcal{H} \vdash v : \tau_1, \top$$

and

$$x{:}\tau_1; pc \sqcap \mathsf{integ}(\tau_1); \mathcal{H} \vdash e_1 : \tau, \mathcal{X}.$$

We need to show $\varnothing; pc; \mathcal{H} \vdash e_1\{v/x\} : \tau, \mathcal{X}$, which follows from Corollary 5 and Lemma 6.

Case EVAL-CONTEXT ($\langle E[e_1], M\rangle \to_\alpha \langle E[e_1'], M'\rangle$, where $\langle e_1, M\rangle \xrightarrow{\mathsf{e}} \langle e_1', M'\rangle$):

We need to show $\varnothing; pc; \mathcal{H} \vdash E[e_1'] : \tau, \mathcal{X}$. We proceed by cases according to the structure of $E[e_1]$.

Case soft e_1:

From the typing of e, we have $\varnothing; pc; \mathcal{H} \vdash e_1 : R_w, \mathcal{X}$, where $\tau = (\mathsf{soft}\ R)_w$. By the induction hypothesis, we have $\varnothing; pc; \mathcal{H} \vdash e_1' : R_w, \mathcal{X}$, so the result follows via T-SOFT.

Cases $e_1 \| e_2$ and $e_2 \| e_1$:

We show case $e_1 \| e_2$. The other case follows symmetrically. From the typing of e, we have $\varnothing; pc; \top \vdash e_1 : \tau_1, \top$. By the induction hypothesis, we have $\varnothing; pc; \top \vdash e_1' : \tau_1, \top$, so the result follows via T-PARALLEL.

Case try e_1 catch p: e_2:

From the typing of e, we have $\varnothing; pc; \mathcal{H}, p \vdash e_1 : \tau_1, \mathcal{X}_1$, where $\tau = \tau_1 \sqcap w$ with $w = \bigsqcap_{p' \in \mathcal{X}_1} (p \sqcup p')$ and $\mathcal{X} = (\mathcal{X}_1 / p) \sqcap \mathcal{X}_2$. By the induction hypothesis, we have $\varnothing; pc; \mathcal{H}, p \vdash e_1' : \tau_1, \mathcal{X}_1$, so the result follows via T-TRY.

Case let $x = e_1$ in e_2:

From the typing of e, we have $\varnothing; pc; \mathcal{H} \vdash e_1 : \tau_1, \mathcal{X}_1$, where $\vdash \mathsf{auth}^+(\tau_1) \preccurlyeq pc$ and $\vdash \mathcal{X}_1 \preccurlyeq \mathcal{X}$. By the induction hypothesis, we have $\varnothing; pc; \mathcal{H} \vdash e_1' : \tau_1, \mathcal{X}_1$, so the result follows via T-LET.

Case FAIL-PROP ($\langle F[\bot_p], M \rangle \to_\alpha \langle \bot_p, M \rangle$):

We have $\varnothing; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}$ and need to show $\varnothing; pc; \mathcal{H} \vdash \bot_p : \tau, \mathcal{X}$. From the typing of e, it follows that $\vdash \mathcal{H} \preccurlyeq \mathcal{X} \preccurlyeq p \neq \top$, so the result follows from T-BOTTOM and T-SUBSUME.

Case BRACKET-SELECT ($\langle [m^S].x_c, M \rangle \to_\alpha \langle [m^{S}.x_c], M \rangle$):

Without loss of generality, assume $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$. Then, from the typing of e, we have

$$\frac{\dfrac{\varnothing; pc \sqcap w; \mathcal{H} \vdash m^S : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top \qquad \alpha \notin w \qquad \vdash a \preccurlyeq pc \sqcap w}{\varnothing; pc; \mathcal{H} \vdash [m^S] : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_w, \top} \qquad \vdash a \preccurlyeq pc \qquad w' = w \sqcap p \qquad \vdash \mathcal{H} \preccurlyeq p}{\varnothing; pc; \mathcal{H} \vdash [m^S].x_c : \tau_c \sqcap w', p} \;,$$

where $\tau = \tau_c \sqcap w'$ and $\mathcal{X} = p$. We need to show $\varnothing; pc; \mathcal{H} \vdash [m^S.x_c] : \tau_c \sqcap w', p$.

By the typing of m^S, we know $\vdash_{wf} S : \mathsf{rectype}$, and so, $\vdash \mathsf{auth}^+(\tau_c) \preccurlyeq a \preccurlyeq pc \sqcap w$. Therefore, the result follows from an application of T-SELECT, followed by T-BRACKET:

$$\dfrac{\dfrac{\varnothing; pc \sqcap w; \mathcal{H} \vdash m^S : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top \qquad \vdash a \preccurlyeq pc \sqcap w \qquad \vdash \mathcal{H} \preccurlyeq p}{\varnothing; pc \sqcap w; \mathcal{H} \vdash m^S.x_c : \tau_c \sqcap p, p} \qquad \alpha \not\preccurlyeq w \qquad \vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc \sqcap w}{\varnothing; pc; \mathcal{H} \vdash [m^S.x_c] : \tau_c \sqcap w', p}\ .$$

Case BRACKET-SOFT-SELECT ($\langle [\mathsf{soft}\ m^S].x_c, M\rangle \rightarrow_\alpha \langle [(\mathsf{soft}\ m^S).x_c], M\rangle$):

Without loss of generality, assume $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$. Then, from the typing of e, we have

$$\dfrac{\dfrac{\varnothing; pc \sqcap w; \mathcal{H} \vdash \mathsf{soft}\ m^S : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top \qquad \alpha \not\preccurlyeq w \qquad \vdash a \preccurlyeq pc \sqcap w}{\varnothing; pc; \mathcal{H} \vdash [\mathsf{soft}\ m^S] : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_w, \top} \qquad \vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc \qquad w' = w \sqcap a \sqcap p \qquad \vdash \mathcal{H} \preccurlyeq p}{\varnothing; pc; \mathcal{H} \vdash [\mathsf{soft}\ m^S].x_c : \tau_c \sqcap w', p}\ ,$$

where $\tau = \tau_c \sqcap w'$ and $\mathcal{X} = p$. We need to show $\varnothing; pc; \mathcal{H} \vdash [(\mathsf{soft}\ m^S).x_c] : \tau_c \sqcap w', p$.

By the typing of m^S, we know $\vdash_{wf} S : \mathsf{rectype}$, and so, $\vdash \mathsf{auth}^+(\tau_c) \preccurlyeq a \preccurlyeq pc \sqcap w$. Therefore, the result follows from T-SOFT-SELECT and T-BRACKET:

$$\dfrac{\dfrac{\varnothing; pc \sqcap w; \mathcal{H} \vdash \mathsf{soft}\ m^S : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top \qquad \vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc \sqcap w \qquad w' = w \sqcap a \sqcap p \qquad \vdash \mathcal{H} \preccurlyeq p}{\varnothing; pc \sqcap w; \mathcal{H} \vdash (\mathsf{soft}\ m^S).x_c : \tau_c \sqcap w', p} \qquad \alpha \not\preccurlyeq w \qquad \vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc \sqcap w}{\varnothing; pc; \mathcal{H} \vdash [(\mathsf{soft}\ m^S).x_c] : \tau_c \sqcap w', p}\ .$$

Case BRACKET-ASSIGN ($\langle [m^S].x_c := v, M\rangle \rightarrow_\alpha \langle [m^S.x_c := v], M\rangle$):

Without loss of generality, assume $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$. Then, from the typing of e, we have

$$\dfrac{\dfrac{\varnothing; pc \sqcap w; \mathcal{H} \vdash m^S : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top \qquad \alpha \not\preccurlyeq w \qquad \vdash a \preccurlyeq pc \sqcap w}{\varnothing; pc; \mathcal{H} \vdash [m^S] : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_w, \top} \qquad \vdash a \preccurlyeq pc \qquad \varnothing; pc; \mathcal{H} \vdash v : \tau', \top \qquad \vdash \tau' \sqcap pc \sqcap w \leq \tau_c \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w \qquad \vdash \mathcal{H} \preccurlyeq p}{\varnothing; pc; \mathcal{H} \vdash [m^S].x_c := v : \mathbf{1}, p},$$

where $\tau = \mathbf{1}$ and $\mathcal{X} = p$. We need to show $\varnothing; pc; \mathcal{H} \vdash [m^S.x_c := v] : \mathbf{1}, p$.

From $\vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w$ and Lemma 8, we know $\varnothing; pc \sqcap w; \mathcal{H} \vdash v : \tau', \top$. The result then follows from an application of T-ASSIGN followed by T-BRACKET:

$$\dfrac{\dfrac{\varnothing; pc \sqcap w; \mathcal{H} \vdash m^S : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top \qquad \vdash a \preccurlyeq pc \qquad \varnothing; pc \sqcap w; \mathcal{H} \vdash v : \tau', \top \qquad \vdash \tau' \sqcap pc \sqcap w \leq \tau_c \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w \qquad \vdash \mathcal{H} \preccurlyeq p}{\varnothing; pc \sqcap w; \mathcal{H} \vdash m^S.x_c := v : \mathbf{1}, p} \qquad \alpha \not\preccurlyeq w}{\varnothing; pc; \mathcal{H} \vdash [m^S.x_c := v] : \mathbf{1}, p}.$$

Case BRACKET-SOFT-ASSIGN

$(\langle[\mathsf{soft}\ m^S].x_c := v, M\rangle \to_\alpha \langle[(\mathsf{soft}\ m^S).x_c := v], M\rangle)$:

Without loss of generality, assume $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$. Then, from the typing of e, we have

$$\frac{\dfrac{\dfrac{\varnothing; pc \sqcap w; \mathcal{H} \vdash \mathsf{soft}\ m^S : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top \qquad \alpha \notin w \qquad \vdash a \preccurlyeq pc \sqcap w}{\varnothing; pc; \mathcal{H} \vdash [\mathsf{soft}\ m^S] : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_w, \top} \qquad \varnothing; pc; \mathcal{H} \vdash v : \tau', \top \qquad \vdash \tau' \sqcap pc \sqcap w \le \tau_c \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w \qquad \vdash \mathcal{H} \preccurlyeq p}{\varnothing; pc; \mathcal{H} \vdash [\mathsf{soft}\ m^S].x_c := v : *, p},$$

where $\tau = \mathbf{1}$ and $\mathcal{X} = p$. We need to show $\varnothing; pc; \mathcal{H} \vdash [(\mathsf{soft}\ m^S).x_c := v] : \mathbf{1}, p$. From $\vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w$ and Lemma 8, we know $\varnothing; pc \sqcap w; \mathcal{H} \vdash v : \tau', \top$. The result then follows from T-SOFT-ASSIGN and T-BRACKET:

$$\frac{\dfrac{\varnothing; pc \sqcap w; \mathcal{H} \vdash \mathsf{soft}\ m^S : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top \qquad \varnothing; pc \sqcap w; \mathcal{H} \vdash v : \tau, \top \qquad \vdash \tau' \sqcap pc \sqcap w \le \tau_c \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w \qquad \vdash \mathcal{H} \preccurlyeq p}{\varnothing; pc \sqcap w; \mathcal{H} \vdash (\mathsf{soft}\ m^S).x_c := v : \mathbf{1}, p \qquad \alpha \notin w}}{\varnothing; pc; \mathcal{H} \vdash [(\mathsf{soft}\ m^S).x_c := v] : \mathbf{1}, p}.$$

Case BRACKET-SOFT ($\langle \mathsf{soft}\ [m^S], M\rangle \to_\alpha \langle[\mathsf{soft}\ m^S], M\rangle$):

Without loss of generality, assume $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$. Then, from the typing of e, we have

$$\frac{\dfrac{\dfrac{\varnothing; pc \sqcap w; \mathcal{H} \vdash m^S : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top \qquad \alpha \notin w \qquad \vdash a \preccurlyeq pc \sqcap w}{\varnothing; pc; \mathcal{H} \vdash [m^S] : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_w, \top}}{\varnothing; pc; \mathcal{H} \vdash \mathsf{soft}\ [m^S] : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_w, \top},$$

where $\tau = (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_w$ and $\mathcal{X} = \top$. We need to show $\varnothing; pc; \mathcal{H} \vdash [\mathsf{soft}\ m^S] : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_w, \top$. This follows via an application of T-SOFT followed by T-BRACKET:

$$\dfrac{\dfrac{\varnothing; pc \sqcap w; \mathcal{H} \vdash m^S : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top}{\varnothing; pc \sqcap w; \mathcal{H} \vdash \mathsf{soft}\ m^S : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top} \quad \alpha \not\preccurlyeq w}{\varnothing; pc; \mathcal{H} \vdash [\mathsf{soft}\ m^S] : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_w, \top}\ .$$

Case BRACKET-EXISTS

$(\langle \mathsf{exists}\ [v]\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2, M\rangle \rightarrow_\alpha \langle [\mathsf{exists}\ v\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2], M\rangle)$:

From the typing of e, we have

$$\dfrac{\begin{array}{c}\varnothing; pc \sqcap \ell; \mathcal{H} \vdash v : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_r)_w, \top \\ \alpha \not\preccurlyeq \ell \qquad \vdash \mathsf{auth}^+(r) \preccurlyeq pc \sqcap \ell \\ \hline \varnothing; pc; \mathcal{H} \vdash [v] : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_r)_{w \sqcap \ell}, \top \\ \vdash \mathsf{auth}^+(r) \preccurlyeq pc \sqcap w \sqcap \ell \qquad w' = \mathsf{auth}^-(r) \sqcap \mathsf{persist}(r) \sqcap w \sqcap \ell \\ x : (\{\overrightarrow{x_i : \tau_i}\}_r)_{w \sqcap \ell}; pc \sqcap w'; \mathcal{H} \vdash e_1 : \tau', \mathcal{X}_1 \\ \varnothing; pc \sqcap w'; \mathcal{H} \vdash e_2 : \tau', \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w'\end{array}}{\varnothing; pc; \mathcal{H} \vdash \mathsf{exists}\ [v]\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2 : \tau' \sqcap w', \mathcal{X}_1 \sqcap \mathcal{X}_2}$$

where $\tau = \tau' \sqcap w'$ and $\mathcal{X} = \mathcal{X}_1 \sqcap \mathcal{X}_2$. We need to show $\varnothing; pc; \mathcal{H} \vdash [\mathsf{exists}\ v\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2] : \tau' \sqcap w', \mathcal{X}_1 \sqcap \mathcal{X}_2$. To do this, we need to know that $x : (\{\overrightarrow{x_i : \tau_i}\}_r)_w; pc \sqcap w'; \mathcal{H} \vdash e_1 : \tau', \mathcal{X}_1$, which can be demonstrated by an easy induction on the derivation of $x : (\{\overrightarrow{x_i : \tau_i}\}_r)_{w \sqcap \ell}; pc \sqcap w'; \mathcal{H} \vdash e_1 : \tau', \mathcal{X}_1$. The result therefore follows via an application of T-EXISTS and T-BRACKET.

$$\frac{\begin{array}{c}\varnothing; pc \sqcap \ell; \mathcal{H} \vdash v : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_r)_w, \top \\ \vdash \mathsf{auth}^+(r) \preccurlyeq pc \sqcap w \sqcap \ell \qquad w'' = \mathsf{auth}^-(r) \sqcap \mathsf{persist}(r) \sqcap w \\ x : (\{\overrightarrow{x_i : \tau_i}\}_r)_w; pc \sqcap \ell \sqcap w''; \mathcal{H} \vdash e_1 : \tau', \mathcal{X}_1 \\ \varnothing; pc \sqcap \ell \sqcap w''; \mathcal{H} \vdash e_2 : \tau', \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap \ell \sqcap w''\end{array}}{\begin{array}{c}\varnothing; pc \sqcap \ell; \mathcal{H} \vdash \mathsf{exists}\ v\ \mathsf{as}\ x :\ e_1\ \mathsf{else}\ e_2 : \tau' \sqcap w'', \mathcal{X}_1 \sqcap \mathcal{X}_2 \\ \alpha \not\preccurlyeq \ell \qquad \vdash \mathsf{auth}^+(\tau' \sqcap w'') \preccurlyeq pc \sqcap \ell \sqcap w'' \preccurlyeq pc \sqcap \ell\end{array}}$$

$$\varnothing; pc; \mathcal{H} \vdash [\mathsf{exists}\ v\ \mathsf{as}\ x :\ e_1\ \mathsf{else}\ e_2] : \tau' \sqcap w', \mathcal{X}_1 \sqcap \mathcal{X}_2$$

(Note that $w' = \ell \sqcap w''$.)

Case BRACKET-APPLY

$(\langle [\lambda(x{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1]\ v_2, M\rangle \rightarrow_\alpha \langle [(\lambda(x{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1)\ v_2], M\rangle)$:

From the typing of e, we have

$$\frac{\begin{array}{c}x{:}\tau_1; pc_1; \mathcal{H}_1 \vdash e_1 : \tau', \mathcal{H}_1 \\ \vdash_{wf} (\tau_1 \xrightarrow{pc_1, \mathcal{H}_1} \tau')_\top : \mathsf{type} \qquad \vdash pc_1 \preccurlyeq pc \sqcap w\end{array}}{\begin{array}{c}\varnothing; pc \sqcap w; \mathcal{H} \vdash \lambda(x{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1 : (\tau_1 \xrightarrow{pc_1, \mathcal{H}_1} \tau')_\top, \top \\ \alpha \not\preccurlyeq w \qquad \vdash pc_1 \preccurlyeq pc \sqcap w\end{array}}$$

$$\frac{\begin{array}{c}\varnothing; pc; \mathcal{H} \vdash [\lambda(x{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1] : (\tau_1 \xrightarrow{pc_1, \mathcal{H}_1} \tau')_w, \top \\ \varnothing; pc; \mathcal{H} \vdash v_2 : \tau_1, \top \qquad \vdash pc_1 \preccurlyeq pc \sqcap w \qquad \vdash \mathcal{H} \preccurlyeq \mathcal{H}_1\end{array}}{\varnothing; pc; \mathcal{H} \vdash [\lambda(x{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1]\ v_2 : \tau' \sqcap w, \mathcal{H}_1} \quad ,$$

where $\tau = \tau' \sqcap w$ and $\mathcal{X} = \mathcal{H}_1$. We need to show $\varnothing; pc; \mathcal{H} \vdash [(\lambda(x : \tau_1)[pc_1; \mathcal{H}_1].\, e_1)\ v_2] : \tau' \sqcap w, \mathcal{H}_1$.

By WT4, from $\vdash_{wf} (\tau_1 \xrightarrow{pc_1, \mathcal{H}_1} \tau')_\top : \mathsf{type}$, we know $\vdash \mathsf{auth}^+(\tau_1) \sqcup \mathsf{auth}^+(\tau') \preccurlyeq pc_1$. Since we also know from the above derivation that $\vdash pc_1 \preccurlyeq pc \sqcap w$, it therefore follows that

$$\vdash \mathsf{auth}^+(\tau_1) \preccurlyeq pc \sqcap w \tag{3.11}$$

and $\vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w$. From the above derivation, we also have $\varnothing; pc; \mathcal{H} \vdash v_2 : \tau_1, \top$. Applying Lemma 8 to this and (3.11), we have $\varnothing; pc \sqcap w; \mathcal{H} \vdash v_2 : \tau_1, \top$. The result then follows from an application of T-APP followed by T-BRACKET:

$$\dfrac{\dfrac{\varnothing; pc \sqcap w; \mathcal{H} \vdash \lambda(x{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1 : (\tau_1 \xrightarrow{pc_1, \mathcal{H}_1} \tau')_\top, \top \qquad \varnothing; pc \sqcap w; \mathcal{H} \vdash v_2 : \tau_1, \top \qquad \vdash pc_1 \preccurlyeq pc \sqcap w \qquad \vdash \mathcal{H} \preccurlyeq \mathcal{H}_1}{\varnothing; pc \sqcap w; \mathcal{H} \vdash (\lambda(x{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1)\ v_2 : \tau', \mathcal{H}_1} \qquad \alpha \notin w \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w}{\varnothing; pc; \mathcal{H} \vdash [(\lambda(x{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1)\ v_2] : \tau' \sqcap w, \mathcal{H}_1} \quad .$$

Case BRACKET-TRY ($\langle \mathsf{try}\ [v]\ \mathsf{catch}\ p{:}\ e_2, M\rangle \rightarrow_\alpha \langle [\mathsf{try}\ v\ \mathsf{catch}\ p{:}\ e_2], M\rangle$):

From the typing of e, we have

$$\dfrac{\dfrac{\varnothing; pc \sqcap \ell; \mathcal{H}, p \vdash v : \tau', \mathcal{X}_1 \qquad \alpha \notin \ell \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap \ell}{\varnothing; pc; \mathcal{H}, p \vdash [v] : \tau' \sqcap \ell, \mathcal{X}_1} \qquad w = \prod_{p' \in \mathcal{X}_1} (p \sqcup p') \qquad \varnothing; pc \sqcap w \sqcap \mathsf{integ}(\tau' \sqcap \ell); \mathcal{H} \vdash e_2 : \tau' \sqcap \ell, \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau' \sqcap \ell) \preccurlyeq pc}{\varnothing; pc; \mathcal{H} \vdash \mathsf{try}\ [v]\ \mathsf{catch}\ p{:}\ e_2 : \tau' \sqcap \ell \sqcap w, (\mathcal{X}_1/p) \sqcap \mathcal{X}_2} \quad ,$$

where $\tau = \tau' \sqcap \ell \sqcap w$ and $\mathcal{X} = (\mathcal{X}_1/p) \sqcap \mathcal{X}_2$.

We need to show

$$\varnothing; pc; \mathcal{H} \vdash [\mathsf{try}\ v\ \mathsf{catch}\ p{:}\ e_2] : \tau' \sqcap \ell \sqcap w, (\mathcal{X}_1/p) \sqcap \mathcal{X}_2.$$

Note that $\mathsf{integ}(\tau' \sqcap \ell) = \mathsf{integ}(\tau') \sqcap \ell$ and $\mathsf{auth}^+(\tau') = \mathsf{auth}^+(\tau' \sqcap \ell) = \mathsf{auth}^+(\tau' \sqcap \ell \sqcap w)$. The result then follows from S3, T-SUBSUME, T-TRY, and T-BRACKET:

$$\dfrac{\dfrac{}{\varnothing; pc \sqcap \ell; \mathcal{H}, p \vdash v : \tau', \mathcal{X}_1} \qquad \dfrac{\vdash \mathsf{integ}(\tau') \sqcap \ell \preccurlyeq \mathsf{integ}(\tau')}{\vdash \tau' \leq \tau' \sqcap \ell}}{\varnothing; pc \sqcap \ell; \mathcal{H}, p \vdash v : \tau' \sqcap \ell, \mathcal{X}_1} \qquad w = \bigsqcap_{p' \in \mathcal{X}_1} (p \sqcup p')$$

$$\dfrac{\varnothing; pc \sqcap \ell \sqcap w \sqcap \mathsf{integ}(\tau' \sqcap \ell); \mathcal{H} \vdash e_2 : \tau' \sqcap \ell, \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau' \sqcap \ell) \preccurlyeq pc \sqcap \ell}{\varnothing; pc \sqcap \ell; \mathcal{H} \vdash \mathsf{try}\ v\ \mathsf{catch}\ p\text{:}\ e_2 : \tau' \sqcap \ell \sqcap w, (\mathcal{X}_1/p) \sqcap \mathcal{X}_2} \qquad \alpha \not\leq \ell \qquad \vdash \mathsf{auth}^+(\tau' \sqcap \ell \sqcap w) \preccurlyeq pc \sqcap \ell$$

$$\varnothing; pc; \mathcal{H} \vdash [\mathsf{try}\ v\ \mathsf{catch}\ p\text{:}\ e_2] : \tau' \sqcap \ell \sqcap w, (\mathcal{X}_1/p) \sqcap \mathcal{X}_2 \quad .$$

Case BRACKET-IF ($\langle \mathsf{if}\ [v]\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, M\rangle \rightarrow_\alpha \langle [\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2], M\rangle$):

From the typing of e, we have

$$\dfrac{\dfrac{\varnothing; pc \sqcap \ell; \mathcal{H} \vdash v : \mathsf{bool}_w, \top \qquad \alpha \not\leq \ell}{\varnothing; pc; \mathcal{H} \vdash [v] : \mathsf{bool}_{w \sqcap \ell}, \top} \qquad \varnothing; pc \sqcap w \sqcap \ell; \mathcal{H} \vdash e_i : \tau', \mathcal{X}_i \ {}^{(\forall i)} \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w \sqcap \ell}{\varnothing; pc; \mathcal{H} \vdash \mathsf{if}\ [v]\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau' \sqcap w \sqcap \ell, \mathcal{X}_1 \sqcap \mathcal{X}_2} \quad ,$$

where $\tau = \tau' \sqcap w \sqcap \ell$ and $\mathcal{X} = \mathcal{X}_1 \sqcap \mathcal{X}_2$. We need to show $\varnothing; pc; \mathcal{H} \vdash [\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2] : \tau' \sqcap w \sqcap \ell, \mathcal{X}_1 \sqcap \mathcal{X}_2$.

Since $\mathsf{auth}^+(\tau' \sqcap w) = \mathsf{auth}^+(\tau')$ and $\vdash pc \sqcap w \sqcap \ell \preccurlyeq pc \sqcap \ell$, we therefore have $\vdash \mathsf{auth}^+(\tau' \sqcap w) \preccurlyeq pc \sqcap \ell$. The result then follows from T-IF and T-BRACKET:

$$\dfrac{\varnothing; pc \sqcap \ell; \mathcal{H} \vdash v : \mathsf{bool}_w, \top \qquad \varnothing; pc \sqcap \ell \sqcap w; \mathcal{H} \vdash e_i : \tau', \mathcal{X}_i \ {}^{(\forall i)} \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap \ell \sqcap w}{\varnothing; pc \sqcap \ell; \mathcal{H} \vdash \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau' \sqcap w, \mathcal{X}_1 \sqcap \mathcal{X}_2} \qquad \alpha \not\leq \ell \qquad \vdash \mathsf{auth}^+(\tau' \sqcap w) \preccurlyeq pc \sqcap \ell$$

$$\varnothing; pc; \mathcal{H} \vdash [\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2] : \tau' \sqcap w \sqcap \ell, \mathcal{X}_1 \sqcap \mathcal{X}_2 \quad .$$

Case BRACKET-LET ($\langle \mathsf{let}\ x = [v]\ \mathsf{in}\ e_1, M\rangle \to_\alpha \langle [e_1\{[v]/x\}], M\rangle$, where $v \neq \bot_p$ for all p):

By Lemma 8, we know $[v]$ can be typed with $\top$ effect. Therefore, from the typing of e, we have

$$\dfrac{w = \mathsf{integ}(\tau'' \sqcap \ell) \qquad \dfrac{\dfrac{\varnothing; pc \sqcap \ell; \mathcal{H} \vdash v : \tau'', \top \qquad \alpha \notin \ell \qquad \vdash \mathsf{auth}^+(\tau'') \preccurlyeq pc \sqcap \ell}{\varnothing; pc; \mathcal{H} \vdash [v] : \tau'' \sqcap \ell, \top} \qquad \vdash \mathsf{auth}^+(\tau'' \sqcap \ell) \preccurlyeq pc}{x{:}\tau'' \sqcap \ell; pc \sqcap w; \mathcal{H} \vdash e_1 : \tau', \mathcal{X}} \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w}{\varnothing; pc; \mathcal{H} \vdash \mathsf{let}\ x = [v]\ \mathsf{in}\ e_1 : \tau' \sqcap w, \mathcal{X}}\ ,$$

where $\tau = \tau' \sqcap w$. We need to show $\varnothing; pc; \mathcal{H} \vdash [e_1\{[v]/x\}] : \tau' \sqcap w, \mathcal{X}$.

From the derivation above, we know $\varnothing; pc; \mathcal{H} \vdash [v] : \tau'' \sqcap \ell, \top$. Since $\mathsf{auth}^+(\tau'' \sqcap \ell) = \mathsf{auth}^+(\tau'')$, from $\vdash \mathsf{auth}^+(\tau'') \preccurlyeq pc \sqcap \ell$ above, we have $\vdash \mathsf{auth}^+(\tau'' \sqcap \ell) \preccurlyeq pc \sqcap \ell$. Therefore, by Lemma 8, we know

$$\varnothing; pc \sqcap \ell; \mathcal{H} \vdash [v] : \tau'' \sqcap \ell, \top. \tag{3.12}$$

From the derivation above, we also know $x{:}\tau'' \sqcap \ell; pc \sqcap w; \mathcal{H} \vdash e_1 : \tau', \mathcal{X}$. By definition, $\vdash w \preccurlyeq \ell$, so by Corollary 5, we know $x{:}\tau'' \sqcap \ell; pc \sqcap \ell; \mathcal{H} \vdash e_1 : \tau', \mathcal{X}$. Applying Lemma 6 to this and (3.12), we have

$$\varnothing; pc \sqcap \ell; \mathcal{H} \vdash e_1\{[v]/x\} : \tau' \sqcap w, \mathcal{X}.$$

Since $\mathsf{auth}^+(\tau' \sqcap w) = \mathsf{auth}^+(\tau')$ and $\vdash w \preccurlyeq \ell$, from $\vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w$ above, we have

$$\vdash \mathsf{auth}^+(\tau' \sqcap w) \preccurlyeq pc \sqcap \ell.$$

Finally, by Lemma 7, we have $\vdash \mathcal{H} \preccurlyeq \mathcal{X}$.

The result, then, follows from T-SUBSUME and T-BRACKET:

$$\dfrac{\dfrac{\varnothing; pc \sqcap \ell; \mathcal{H} \vdash e_1\{[v]/x\} : \tau' \sqcap w, \mathcal{X} \qquad \vdash \mathcal{H} \preccurlyeq \mathcal{X}}{\varnothing; pc \sqcap \ell; \mathcal{H} \vdash e_1\{[v]/x\} : \tau' \sqcap w, \mathcal{X}} \qquad \alpha \notin \ell \qquad \vdash \mathsf{auth}^+(\tau' \sqcap w) \preccurlyeq pc \sqcap \ell}{\varnothing; pc; \mathcal{H} \vdash [e_1\{[v]/x\}] : \tau' \sqcap w \sqcap \ell, \mathcal{X}} .$$

Case DOUBLE-BRACKET ($\langle [[v]], M\rangle \rightarrow_\alpha \langle [v], M\rangle$):

From the typing of e, we have

$$\dfrac{\varnothing; pc \sqcap \ell; \mathcal{H} \vdash [v] : \tau', \mathcal{X} \qquad \alpha \notin \ell \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap \ell}{\varnothing; pc; \mathcal{H} \vdash [[v]] : \tau' \sqcap \ell, \mathcal{X}} ,$$

where $\tau = \tau' \sqcap \ell$. We need to show $\varnothing; pc; \mathcal{H} \vdash [v] : \tau' \sqcap \ell, \mathcal{X}$. By Corollary 5, we know $\varnothing; pc; \mathcal{H} \vdash [v] : \tau', \mathcal{X}$. The result then follows via S3 and T-SUBSUME:

$$\dfrac{\varnothing; pc; \mathcal{H} \vdash [v] : \tau', \mathcal{X} \qquad \dfrac{\vdash \mathsf{integ}(\tau') \sqcap \ell \preccurlyeq \mathsf{integ}(\tau')}{\vdash \tau' \leq \tau' \sqcap \ell}}{\varnothing; pc; \mathcal{H} \vdash [v] : \tau' \sqcap \ell, \mathcal{X}} .$$

Case BRACKET-CONTEXT ($\langle [e_1], M\rangle \rightarrow_\alpha \langle [e_1'], M'\rangle$, where $\langle e_1, M\rangle \rightarrow_\alpha \langle e_1', M'\rangle$):

From the typing of e, we have

$$\dfrac{\varnothing; pc \sqcap \ell; \mathcal{H} \vdash e_1 : \tau', \mathcal{X} \qquad \alpha \notin \ell \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap \ell}{\varnothing; pc; \mathcal{H} \vdash [e_1] : \tau' \sqcap \ell, \mathcal{X}} ,$$

where $\tau = \tau' \sqcap \ell$. We need to show $\varnothing; pc; \mathcal{H} \vdash [e_1'] : \tau, \mathcal{X}$.

By the induction hypothesis, we have $\varnothing; pc \sqcap \ell; \mathcal{H} \vdash e_1' : \tau', \mathcal{X}$. The result follows by T-BRACKET:

$$\dfrac{\varnothing; pc \sqcap \ell; \mathcal{H} \vdash e_1' : \tau', \mathcal{X} \qquad \alpha \notin \ell \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap \ell}{\varnothing; pc; \mathcal{H} \vdash [e_1'] : \tau' \sqcap \ell, \mathcal{X}} .$$

Case BRACKET-FAIL ($\langle F[[\bot_p]], M\rangle \to_\alpha \langle [\bot_p], M\rangle$):

We need to show $\varnothing; pc; \mathcal{H} \vdash [\bot_p] : \tau, \mathcal{X}$. We proceed by cases according to the structure of $F[[\bot_p]]$.

Case soft $[\bot_p]$:

From the typing of e, we have

$$\dfrac{\dfrac{\dfrac{p \neq \top \qquad \vdash \mathcal{H} \preccurlyeq p}{\varnothing; pc \sqcap w; \mathcal{H} \vdash \bot_p : R_\top, p} \qquad \alpha \notin w \qquad \vdash \mathsf{auth}^+(R_\top) \preccurlyeq pc \sqcap w}{\varnothing; pc; \mathcal{H} \vdash [\bot_p] : R_w, p}}{\varnothing; pc; \mathcal{H} \vdash \mathsf{soft}\ [\bot_p] : (\mathsf{soft}\ R)_w, p}\ ,$$

where $\tau = (\mathsf{soft}\ R)_w$ and $\mathcal{X} = p$. The result follows via T-BOTTOM and T-BRACKET:

$$\dfrac{\dfrac{\dfrac{p \neq \top \qquad \vdash \mathcal{H} \preccurlyeq p}{\varnothing; pc \sqcap w; \mathcal{H} \vdash \bot_p : (\mathsf{soft}\ R)_\top, p} \qquad \alpha \notin w \qquad \vdash \mathsf{auth}^+((\mathsf{soft}\ R)_\top) \preccurlyeq pc \sqcap w}{\varnothing; pc; \mathcal{H} \vdash [\bot_p] : (\mathsf{soft}\ R)_w, p}\ .$$

Case let $x = [\bot_p]$ in e_2:

From the typing of e, we have

$$\dfrac{\dfrac{\dfrac{p \neq \top \qquad \vdash \mathcal{H} \preccurlyeq p}{\varnothing; pc \sqcap \ell; \mathcal{H} \vdash \bot_p : \tau_1, p} \qquad \alpha \notin \ell \qquad \vdash \mathsf{auth}^+(\tau_1) \preccurlyeq pc \sqcap \ell}{\varnothing; pc; \mathcal{H} \vdash [\bot_p] : \tau_1 \sqcap \ell, p} \qquad \vdash \mathsf{auth}^+(\tau_1 \sqcap \ell) \preccurlyeq pc \qquad w = p \sqcap \mathsf{integ}(\tau_1 \sqcap \ell)}{\varnothing; pc; \mathcal{H} \vdash \mathsf{let}\ x = [\bot_p]\ \mathsf{in}\ e_2 : \tau_2 \sqcap w, \mathcal{X}_2 \sqcap p}\ ,$$

where $\tau = \tau_2 \sqcap w$ and $\mathcal{X} = \mathcal{X}_2 \sqcap p$. By construction, $\vdash w \preccurlyeq \ell$, so we know $\alpha \notin w$. By Lemma 7, we also know $\vdash \mathcal{H} \preccurlyeq \mathcal{X}_2 \sqcap p$. The result follows via T-BOTTOM, T-BRACKET and T-SUBSUME:

$$\dfrac{\dfrac{\dfrac{p \neq \top \qquad \vdash \mathcal{H} \preccurlyeq p}{\varnothing; pc \sqcap w; \mathcal{H} \vdash \bot_p : \tau_2, p} \qquad \dfrac{\alpha \notin w \qquad \vdash \mathsf{auth}^+(\tau_2) \preccurlyeq pc \sqcap w}{\varnothing; pc; \mathcal{H} \vdash [\bot_p] : \tau_2 \sqcap w, p} \qquad \vdash \mathcal{H} \preccurlyeq \mathcal{X}_2 \sqcap p \qquad \vdash \mathcal{X}_2 \sqcap p \preccurlyeq \mathcal{X}_2}{\varnothing; pc; \mathcal{H} \vdash [\bot_p] : \tau_2 \sqcap w, \mathcal{X}_2 \sqcap p}\,.$$

□

We now show that well-formedness is also preserved during execution.

Lemma 10 (Well-formedness preservation)**.** *If $\langle e, M\rangle$ is a well-formed configuration wherein e is well-typed, and $\langle e, M\rangle$ takes an $\rightarrow_\alpha$ transition, then the new configuration $\langle e', M'\rangle$ will also be well-formed:*

$$\vdash^{\alpha}_{[wf]} \langle e, M\rangle \wedge \varnothing; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \wedge \langle e, M\rangle \rightarrow_\alpha \langle e', M'\rangle \Rightarrow \vdash^{\alpha}_{[wf]} \langle e', M'\rangle\,.$$

Proof. First, note that $\mathsf{dom}(M) \subseteq \mathsf{dom}(M')$. This can be shown by an easy induction on the derivation of $\langle e, M\rangle \rightarrow_\alpha \langle e', M'\rangle$.

We prove the lemma by induction on the derivation of $\langle e, M\rangle \rightarrow_\alpha \langle e', M'\rangle$. Given $\langle e, M\rangle \rightarrow_\alpha \langle e', M'\rangle$, the proof proceeds by cases according to the evaluation rules.

Rules DANGLE-SELECT, DANGLE-ASSIGN, PARALLEL-RESULT, TRY-ESC, FAIL-PROP, and BRACKET-FAIL, are trivial base cases in which $\mathsf{locs}(e') = \varnothing$ and $M' = M$. Rules SELECT, and SOFT-SELECT follow from the definition of $\vdash^{\alpha}_{[wf]} M$.

Rules APPLY, EXISTS-TRUE, EXISTS-FALSE, TRY-VAL, TRY-CATCH, IF-TRUE, IF-FALSE, LET, BRACKET-SELECT, BRACKET-SOFT-SELECT, BRACKET-ASSIGN,

BRACKET-SOFT-ASSIGN, BRACKET-SOFT, BRACKET-EXISTS, BRACKET-APPLY, BRACKET-TRY, BRACKET-IF, BRACKET-LET, and DOUBLE-BRACKET follow because in these cases, $\mathsf{locs}(e') \subseteq \mathsf{locs}(e)$, $M' = M$, and $\mathsf{root}(m^S, e') \Rightarrow \mathsf{root}(m^S, e)$.

Rule GC follows from the fact that $\vdash^{\alpha}_{[wf]} M$ and that minimal collectible groups must be disjoint.

Rules α-CREATE and α-ASSIGN follow trivially from the transition rules.

Rule α-FORGET follows from the fact that $\vdash^{\alpha}_{[wf]} M$.

Case CREATE ($\left\langle \{\overrightarrow{x_i = v_i}\}^S, M \right\rangle \rightarrow_\alpha \left\langle m^S, M[m^S \mapsto \{\overrightarrow{x_i = v_i \blacktriangleright_\alpha \tau_i}\}] \right\rangle$, where m^S is fresh and $S = \{\overrightarrow{x_i : \tau_i}\}_s$):

We show $\varnothing; \top; \top \vdash v_i : \tau_i, \top$ for arbitrary i; the rest of this case follows from $\vdash^{\alpha}_{[wf]} \langle e, M \rangle$.

From the typing of e, we have $\varnothing; pc; \mathcal{H} \vdash v_i : \tau_i, \top$. From this, the result follows via Lemma 8 and T-BRACKET.

Case ASSIGN ($\langle m^S.x_c := v, M \rangle \rightarrow_\alpha \langle * \blacktriangleright_\alpha p, M[m^S \mapsto v \blacktriangleright_\alpha \tau_c] \rangle$, where $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$):

From the typing of e and by T-SUBSUME, we have $\varnothing; pc; \mathcal{H} \vdash v : \tau_c, \top$. By Lemma 8, we have $\varnothing; \top; \top \vdash v : \tau_c, \top$, and the rest of this case follows from $\vdash^{\alpha}_{[wf]} \langle e, M \rangle$.

Case SOFT-ASSIGN ($\langle (\mathsf{soft}\ m^S).x_c := v, M \rangle \rightarrow_\alpha \langle v' \blacktriangleright_\alpha (a \sqcap p), M' \rangle$, where $\langle m^S.x_c := v, M \rangle \xrightarrow{\mathsf{e}} \langle v', M' \rangle$ and $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$):

If $M(m^S) = \bot$, then we have $\mathsf{locs}(e') = \varnothing$ and $M' = M$, and the result follows.

Otherwise, $M(m^S) \neq \bot$, so from the definition of $\vdash^{\alpha}_{[wf]} \langle (\mathsf{soft}\ m^S).x_c := v, M \rangle$, we can obtain $\vdash^{\alpha}_{[wf]} \langle m^S.x_c := v, M \rangle$, and the rest follows similarly to the previous case.

Case EVAL-CONTEXT ($\langle E[e_1], M\rangle \rightarrow_\alpha \langle E[e_1'], M'\rangle$, where $\langle e_1, M\rangle \xrightarrow{\mathsf{e}} \langle e_1', M'\rangle$):

Since $\vdash^{\alpha}_{[wf]} \langle e, M\rangle$, we know $\vdash^{\alpha}_{[wf]} \langle e_1, M\rangle$. Proceeding by cases according to the structure of $E[e_1]$, in each case we have $\varnothing; pc_1; \mathcal{H}_1 \vdash e_1 : \tau_1, \mathcal{X}_1$ (for some pc_1, $\mathcal{H}_1$, τ_1, and $\mathcal{X}_1$). So, by the induction hypothesis, we have $\vdash^{\alpha}_{[wf]} \langle e_1', M'\rangle$. From this and the fact that $\vdash^{\alpha}_{[wf]} \langle e, M\rangle$, the result follows.

Case BRACKET-CONTEXT ($\langle [e_1], M\rangle \rightarrow_\alpha \langle [e_1'], M'\rangle$, where $\langle e_1, M\rangle \xrightarrow{\mathsf{e}} \langle e_1', M'\rangle$):

Since $\vdash^{\alpha}_{[wf]} \langle e, M\rangle$, we know $\vdash^{\alpha}_{[wf]} \langle e_1, M\rangle$. From the typing of e, we have $\varnothing; pc'; \mathcal{H} \vdash e_1 : \tau_1, \mathcal{X}$, for some appropriate pc' and τ_1. So, by the induction hypothesis, we have $\vdash^{\alpha}_{[wf]} \langle e_1', M'\rangle$, and therefore $\vdash^{\alpha}_{[wf]} \langle e', M'\rangle$.

□

Corollary 11 (Preservation)**.**

$$\vdash^{\alpha}_{[wf]} \langle e, M\rangle \wedge \varnothing; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \wedge \langle e, M\rangle \rightarrow^*_\alpha \langle e', M'\rangle$$
$$\Rightarrow \vdash^{\alpha}_{[wf]} \langle e', M'\rangle \wedge \varnothing; pc; \mathcal{H} \vdash e' : \tau, \mathcal{X}$$

Proof. This follows from Lemmas 9 and 10 by induction on the number of $\rightarrow_\alpha$ transitions taken. □

Corollary 12 (Preservation (non-adversarial execution))**.**

$$\vdash_{[wf]} \langle e, M\rangle \wedge \varnothing; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \wedge \langle e, M\rangle \rightarrow^* \langle e', M'\rangle$$
$$\Rightarrow \vdash_{[wf]} \langle e', M'\rangle \wedge \varnothing; pc; \mathcal{H} \vdash e' : \tau, \mathcal{X}$$

Proof. This follows by Corollary 11 and the definition of $\vdash_{[wf]} \langle e, M\rangle$. □

Lemma 13 (Progress)**.** *Let $\langle e, M\rangle$ be a configuration wherein e has type τ and effect $\mathcal{X}$, and the locations appearing in e are mapped by M. Then either e is a value, or $\langle e, M\rangle$ can take an $\xrightarrow{\mathsf{e}}$ transition:*

$$\varnothing; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \wedge \mathsf{locs}(e) \subseteq \mathsf{dom}(M)$$
$$\Rightarrow e \textit{ is a value} \vee \exists e', M'.\ \langle e, M\rangle \xrightarrow{\mathsf{e}} \langle e', M'\rangle$$

(Doubly bracketed values are considered expressions and not values.)

Proof. By induction on the derivation of $\varnothing; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}$. We proceed by cases according to the syntax of e. Note that since e is typed in an empty type context ($\Gamma = \varnothing$), we must have $\mathsf{FV}(e) = \varnothing$.

Case $e = v$:

Trivial since e is a value.

Case $e = v_1\ v_2$:

From the typing of e, we know that v_1 is a value with an arrow type and $\top$ effect, so it is either an abstraction $\lambda(x{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1$ or a bracketed value $[v_1']$. In the former case, by APPLY, we have

$$\langle e, M\rangle = \langle (\lambda(x{:}\tau_1)[pc_1; \mathcal{H}_1].\, e_1)\ v_2, M\rangle \xrightarrow{\mathsf{e}} \langle e_1\{v_2/x\}, M\rangle\,.$$

In the latter case, by BRACKET-APPLY, we have $\langle e, M\rangle = \langle [v_1']\ v_2, M\rangle \xrightarrow{\mathsf{e}} \langle [v_1'\ v_2], M\rangle$.

Case $e = \mathsf{if}\ v_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$:

From the typing of e, we know that v_1 is a value with bool type and $\top$ effect, so either $v_1 = \mathsf{true}$, $v_1 = \mathsf{false}$, or v_1 is a bracketed value $[v_1']$. If $v_1 = \mathsf{true}$, then by IF-TRUE, we have $\langle e, M\rangle = \langle \mathsf{if\ true\ then}\ e_2\ \mathsf{else}\ e_3, M\rangle \xrightarrow{\mathsf{e}} \langle e_2, M\rangle$. If $v_1 = \mathsf{false}$, then by IF-FALSE, we have $\langle e, M\rangle = \langle \mathsf{if\ false\ then}\ e_2\ \mathsf{else}\ e_3, M\rangle \xrightarrow{\mathsf{e}} \langle e_3, M\rangle$. Otherwise, $v_1 = [v_1']$ and by BRACKET-IF, we have $\langle e, M\rangle = \langle \mathsf{if}\ [v_1']\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3, M\rangle \xrightarrow{\mathsf{e}} \langle [\mathsf{if}\ v_1'\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3], M\rangle$.

Case $e = \{\overrightarrow{x_i = v_i}\}^S$:

Trivial by CREATE.

Case $e = v.x_c$:

From the typing of e, we know v is a value with record type and $\top$ effect, so it is either a bracketed hard reference ($v = [m^S]$), a bracketed soft reference ($v = [\mathsf{soft}\ m^S]$), a hard reference ($v = m^S$), or a soft reference ($v = \mathsf{soft}\ m^S$).

In the first case ($v = [m^S]$), by BRACKET-SELECT, we have $\langle e, M\rangle = \langle [m^S].x_c, M\rangle \xrightarrow{\mathsf{e}} \langle [m^S.x_c], M\rangle$. In the second case ($v = [\mathsf{soft}\ m^S]$), by BRACKET-SOFT-SELECT, we have $\langle e, M\rangle = \langle [\mathsf{soft}\ m^S].x_c, M\rangle \xrightarrow{\mathsf{e}} \langle [(\mathsf{soft}\ m^S).x_c], M\rangle$.

In the third ($v = m^S$) and fourth ($v = \mathsf{soft}\ m^S$) cases, assume without loss of generality $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$. Since $\mathsf{locs}(e) \subseteq \mathsf{dom}(M)$, we must have $m^S \in \mathsf{dom}(M)$. There are two sub-cases to consider. In each sub-case, we will show $\langle m^S.x_c, M\rangle \xrightarrow{\mathsf{e}} \langle v'', M\rangle$, for some v'', thereby proving the case of a hard reference ($v = m^S$). The case of a soft reference ($v = \mathsf{soft}\ m^S$) follows by SOFT-SELECT: $\langle (\mathsf{soft}\ m^S).x_c, M\rangle \xrightarrow{\mathsf{e}} \langle v'' \blacktriangleright_\alpha (a \sqcap p), M\rangle$.

1. Case $M(m^S) \neq \bot$:

   Without loss of generality, assume $M(m^S) = \{\overrightarrow{x_i = v_i}\}$. Then, by SELECT,

   $$\left\langle m^S.x_c, M\right\rangle \xrightarrow{\mathsf{e}} \langle v_c \blacktriangleright_\alpha p, M\rangle\,.$$

2. Case $M(m^S) = \bot$:

   By DANGLE-SELECT, $\langle m^S.x_c, M\rangle \xrightarrow{\mathsf{e}} \langle \bot_p \blacktriangleright_\alpha p, M\rangle$.

Case $e = v.x_c := v'$:

From the typing of e, we know that v is a value with record type and $\top$ effect, so it must be either a bracketed hard reference ($v = [m^S]$), a bracketed soft reference ($v = [\mathsf{soft}\ m^S]$), a hard reference ($v = m^S$), or a soft reference ($v = \mathsf{soft}\ m^S$).

In the first case ($v = [m^S]$), by BRACKET-ASSIGN, we have $\langle e, M\rangle = \langle [m^S].x_c := v', M\rangle \xrightarrow{\mathsf{e}} \langle [m^S.x_c := v'], M\rangle$.

In the second case ($v = [\mathsf{soft}\ m^S]$), by BRACKET-SOFT-ASSIGN, we have $\langle e, M\rangle = \langle [\mathsf{soft}\ m^S].x_c := v', M\rangle \xrightarrow{\mathsf{e}} \langle [(\mathsf{soft}\ m^S).x_c := v'], M\rangle$.

For the remaining cases, suppose $v = m^S$ or $v = \mathsf{soft}\, m^S$. Then, since $\mathsf{locs}(e) \subseteq \mathsf{dom}(M)$, we must have $m^S \in \mathsf{dom}(M)$. Without loss of generality, assume $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$. From the typing of e, we also know that v' is a value with $\top$ effect, so $v' \neq \bot_{p'}$ and $v' \neq [\bot_{p'}]$ for all p'. There are two sub-cases to consider. In each sub-case, we will show $\langle m^S.x_c := v', M\rangle \xrightarrow{\mathsf{e}} \langle v''', M'\rangle$, for some v''' and some M', thereby proving the case of a hard reference ($v = m^S$). The case of a soft reference ($v = \mathsf{soft}\, m^S$) follows by SOFT-ASSIGN: $\langle (\mathsf{soft}\, m^S).x_c := v', M\rangle \xrightarrow{\mathsf{e}} \langle v''' \blacktriangleright_\alpha (a \sqcap p), M'\rangle$.

1. Case $M(m^S) \neq \bot$:

   Without loss of generality, assume $M(m^S) = \{\overrightarrow{x_i = v_i}\}$. Then, by ASSIGN,

   $$\left\langle m^S.x_c := v', M\right\rangle \xrightarrow{\mathsf{e}} \left\langle * \blacktriangleright_\alpha p, M[m^S.x_c \mapsto v' \blacktriangleright_\alpha \tau_c]\right\rangle.$$

2. Case $M(m^S) = \bot$:

   By DANGLE-ASSIGN, $\langle m^S.x_c := v', M\rangle \xrightarrow{\mathsf{e}} \langle \bot_p \blacktriangleright_\alpha p, M\rangle$.

Case $e = \mathsf{exists}\, v\, \mathsf{as}\, x : e_1\, \mathsf{else}\, e_2$:

From the typing of e, we know that v is a value with soft reference type, so it is either a bracketed value ($v = [v']$) or a soft reference ($v = \mathsf{soft}\, m^S$).

In the first case ($v = [v']$), by BRACKET-EXISTS, we have

$$\langle e, M\rangle = \langle \mathsf{exists}\, [v']\, \mathsf{as}\, x : e_1\, \mathsf{else}\, e_2, M\rangle \xrightarrow{\mathsf{e}} \langle [\mathsf{exists}\, v'\, \mathsf{as}\, x : e_1\, \mathsf{else}\, e_2], M\rangle$$

The second case ($v = \mathsf{soft}\, m^S$) splits into two sub-cases. Assume $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$ without loss of generality. If $M(m^S) \neq \bot$, then by EXISTS-TRUE, we have

$$\langle e, M\rangle = \left\langle \mathsf{exists\ soft}\, m^S\, \mathsf{as}\, x : e_1\, \mathsf{else}\, e_2, M\right\rangle \xrightarrow{\mathsf{e}} \left\langle (e_1\{m^S/x\}) \blacktriangleright_\alpha (a \sqcap p), M\right\rangle$$

Otherwise, $M(m^S) = \bot$, and by EXISTS-FALSE, we have

$$\langle e, M\rangle = \left\langle \mathsf{exists\ soft}\ m^S\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2, M\right\rangle \xrightarrow{\mathsf{e}} \langle e_2 \blacktriangleright_\alpha (a \sqcap p), M\rangle$$

Case $e = \mathsf{soft}\ e_1$:

From the typing of e, we know that $\varnothing; pc; \mathcal{H} \vdash e_1 : R_w, \mathcal{X}$, where $\tau = (\mathsf{soft}\ R)_w$. Since $\mathsf{locs}(e) \subseteq \mathsf{dom}(M)$, we must also have $\mathsf{locs}(e_1) \subseteq \mathsf{dom}(M)$, so by the induction hypothesis, either e_1 is a value or $\langle e_1, M\rangle \xrightarrow{\mathsf{e}} \langle e_1', M'\rangle$, for some e_1' and M'. There are four cases to consider:

1. e_1 is a value $\bot_p$,
2. e_1 is a bracketed value $[v]$,
3. e_1 is some other value, and
4. $\langle e_1, M\rangle \xrightarrow{\mathsf{e}} \langle e_1', M'\rangle$.

In case 1, by FAIL-PROP, we have $\langle e, M\rangle = \langle \mathsf{soft}\ \bot_p, M\rangle \xrightarrow{\mathsf{e}} \langle \bot_p, M\rangle$.

In case 2, by BRACKET-SOFT, we have $\langle e, M\rangle = \langle \mathsf{soft}\ [v], M\rangle \xrightarrow{\mathsf{e}} \langle [\mathsf{soft}\ v], M\rangle$.

In case 3, e_1 must be a hard reference m^S, so $e = \mathsf{soft}\ m^S$ is also a value.

In case 4, by EVAL-CONTEXT, we have $\langle e, M\rangle = \langle \mathsf{soft}\ e_1, M\rangle \xrightarrow{\mathsf{e}} \langle \mathsf{soft}\ e_1', M'\rangle$.

Case $e = e_1 \| e_2$:

From the typing of e, we have $\varnothing; pc; \top \vdash e_i : \tau_i, \top$ for $i \in \{1, 2\}$. Since $\mathsf{locs}(e) \subseteq \mathsf{dom}(M)$, we must also have $\mathsf{locs}(e_i) \subseteq \mathsf{dom}(M)$, so by the induction hypothesis, either e_1 and e_2 are both values, or (without loss of generality) $\langle e_1, M\rangle \xrightarrow{\mathsf{e}} \langle e_1', M'\rangle$, for some e_1' and M'.

If e_1 and e_2 are both values, then by PARALLEL-RESULT, we have $\langle e, M\rangle = \langle e_1 \| e_2, M\rangle \xrightarrow{\mathsf{e}} \langle *, M\rangle$.

Otherwise, by EVAL-CONTEXT, we have $\langle e, M\rangle = \langle e_1 \| e_2, M\rangle \xrightarrow{\mathsf{e}} \langle e_1' \| e_2, M'\rangle$.

Case $e = \mathsf{try}\ e_1\ \mathsf{catch}\ p\colon e_2$:

From the typing of e, we have $\varnothing; pc; \mathcal{H}, p \vdash e_1 : \tau_1, \mathcal{X}_1$, where $\tau = \tau_1 \sqcap w$ with $w = \bigsqcap_{p' \in \mathcal{X}_1} (p \sqcup p')$. Since $\mathsf{locs}(e) \subseteq \mathsf{dom}(M)$, we must also have $\mathsf{locs}(e_1) \subseteq \mathsf{dom}(M)$, so by the induction hypothesis, either e_1 is a value or $\langle e_1, M\rangle \xrightarrow{\mathsf{e}} \langle e_1', M'\rangle$, for some e_1' and M'. There are five cases to consider:

1. e_1 is a value $\bot_{p'}$ and $\vdash p \preccurlyeq p'$,
2. e_1 is a value $\bot_{p'}$ and $p \not\preccurlyeq p'$,
3. e_1 is a bracketed value $[v]$,
4. e_1 is some other value v, and
5. $\langle e_1, M\rangle \xrightarrow{\mathsf{e}} \langle e_1', M'\rangle$.

In case 1, by TRY-CATCH, we have $\langle e, M\rangle = \langle \mathsf{try}\ \bot_{p'}\ \mathsf{catch}\ p\colon e_2, M\rangle \xrightarrow{\mathsf{e}} \langle e_2, M\rangle$.

In case 2, by TRY-ESC, we have $\langle e, M\rangle = \langle \mathsf{try}\ \bot_{p'}\ \mathsf{catch}\ p\colon e_2, M\rangle \xrightarrow{\mathsf{e}} \langle \bot_{p'}, M\rangle$.

In case 3, by BRACKET-TRY, we have $\langle e, M\rangle = \langle \mathsf{try}\ [v]\ \mathsf{catch}\ p\colon e_2, M\rangle \xrightarrow{\mathsf{e}} \langle [\mathsf{try}\ v\ \mathsf{catch}\ p\colon e_2], M\rangle$.

In case 4, by TRY-VAL, we have $\langle e, M\rangle = \langle \mathsf{try}\ v\ \mathsf{catch}\ p\colon e_2, M\rangle \xrightarrow{\mathsf{e}} \langle v, M\rangle$.

Finally, in case 5, by EVAL-CONTEXT, we have

$$\langle e, M\rangle = \langle \mathsf{try}\ e_1\ \mathsf{catch}\ p\colon e_2, M\rangle \xrightarrow{\mathsf{e}} \langle \mathsf{try}\ e_1'\ \mathsf{catch}\ p\colon e_2, M'\rangle$$

Case $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$:

From the typing of e, we have $\varnothing; pc; \mathcal{H} \vdash e_1 : \tau_1, \mathcal{X}_1$. Since $\mathsf{locs}(e) \subseteq \mathsf{dom}(M)$, we must also have $\mathsf{locs}(e_1) \subseteq \mathsf{dom}(M)$, so by the induction hypothesis, either e_1 is a value or $\langle e_1, M\rangle \xrightarrow{\mathsf{e}} \langle e_1', M'\rangle$, for some e_1' and M'. There are five cases to consider:

1. e_1 is a bottom value $\bot_p$,
2. e_1 is a bracketed bottom value $[\bot_p]$,
3. e_1 is some other bracketed value $[v]$,
4. e_1 is some other value v, and
5. $\langle e_1, M\rangle \xrightarrow{\mathsf{e}} \langle e_1', M'\rangle$.

In case 1, by FAIL-PROP, we have $\langle e, M\rangle = \langle \mathsf{let}\ x = \bot_p\ \mathsf{in}\ e_2, M\rangle \xrightarrow{\mathsf{e}} \langle \bot_p, M\rangle$.

In case 2, by BRACKET-FAIL, we have $\langle e, M\rangle = \langle \mathsf{let}\ x = [\bot_p]\ \mathsf{in}\ e_2, M\rangle \xrightarrow{\mathsf{e}} \langle [\bot_p], M\rangle$.

In case 3, by BRACKET-LET, we have $\langle e, M\rangle = \langle \mathsf{let}\ x = [v]\ \mathsf{in}\ e_2, M\rangle \xrightarrow{\mathsf{e}} \langle [e_2\{[v]/x\}], M\rangle$.

In case 4, by LET, we have $\langle e, M\rangle = \langle \mathsf{let}\ x = v\ \mathsf{in}\ e_2, M\rangle \xrightarrow{\mathsf{e}} \langle e_2\{v/x\}, M\rangle$.

In case 5, by EVAL-CONTEXT, we have $\langle e, M\rangle = \langle \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, M\rangle \xrightarrow{\mathsf{e}} \langle \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2, M'\rangle$.

Case $e = [e']$:

From the typing of e, we have $\varnothing; pc'; \mathcal{H} \vdash e' : \tau', \mathcal{X}$. Since $\mathsf{locs}(e) \subseteq \mathsf{dom}(M)$, we must also have $\mathsf{locs}(e') \subseteq \mathsf{dom}(M)$, so by the induction hypothesis, either e' is a value or $\langle e', M\rangle \xrightarrow{\mathsf{e}} \langle e'', M'\rangle$.

If e' is a bracketed value $[v]$, then by DOUBLE-BRACKET, we have $\langle e, M\rangle = \langle [[v]], M\rangle \xrightarrow{\mathsf{e}} \langle [v], M\rangle$.

If e' is some other value v, then $e = [v]$ is a value.

Otherwise, by BRACKET-CONTEXT, we have $\langle e, M\rangle = \langle [e'], M\rangle \xrightarrow{\mathsf{e}} \langle [e''], M'\rangle$.

□

Corollary 14 (Soundness of $[\lambda_{persist}]$)**.**

$$\vdash^{\alpha}_{[wf]} \langle e, M\rangle \wedge \varnothing; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}$$
$$\Rightarrow \langle e, M\rangle \Uparrow \vee \exists v \in \mathsf{Val}, M'.\ \langle e, M\rangle \xrightarrow{\mathsf{e}}{}^{*} \langle v, M'\rangle$$

Proof. This follows from Corollary 11 and Lemma 13 by induction on the number of $\xrightarrow{\mathsf{e}}$ transitions taken. □

3.7.4 Limited adversary influence

The key to proving both referential integrity and immunity to storage attacks is to show that the adversary cannot meaningfully influence the high-integrity parts of the program and memory. This property is similar to noninterference [29], and similarly can be expressed using an equivalence relation on configurations. Two configurations are equivalent if they agree on all high-integrity parts of the program and of the memory.

The property states that for any execution influenced by the adversary, there is a corresponding, equivalent execution in which the adversary is not present. Hence, the adversary's influence is not significant. More precisely, each configuration $\langle e_1, M_1\rangle$ reached via the language augmented by adversarial transitions must be equivalent to some configuration $\langle e_2, M_2\rangle$ reachable by purely non-adversarial execution. This is a possibilistic security property, which is problematic for confidentiality properties [70], but is acceptable for integrity.

Because the two executions being compared operate on different heaps, with the adversary behaving differently in the two executions, the addresses chosen when records are allocated may differ. However, the structure of the high-integrity part of the heap should still correspond. A homomorphism ϕ is used to relate corresponding locations in the two heaps that are high-integrity or high-persistence.

Definition 11 (High-integrity homomorphism)**.** *An injective partial function* ϕ : $\mathsf{dom}(M_1) \rightharpoonup \mathsf{dom}(M_2)$ *is a* high-integrity homomorphism from M_1 to M_2 *if it satisfies the following:*

- *Injective:* $m_1^{S_1} \neq m_2^{S_2} \wedge \{m_1^{S_1}, m_2^{S_2}\} \subseteq \mathsf{dom}(\phi) \Rightarrow \phi(m_1^{S_1}) \neq \phi(m_2^{S_2})$*;*
- *Type-preserving:* $m_2^{S_2} = \phi(m_1^{S_1}) \Rightarrow S_1 = S_2$*; and*
- *Isomorphous when the domain and range are restricted to the high-integrity and high-persistence locations in* M_1 *and* M_2*:*

$$\phi|_D : D \twoheadrightarrow R,$$

where $D = \{m^S \in \mathsf{dom}(M_1) \mid \vdash \alpha \preccurlyeq \mathsf{integ}(S) \vee \vdash \alpha \preccurlyeq \mathsf{persist}(S)\}$ *and* $R = \{m^S \in \mathsf{dom}(M_2) \mid \vdash \alpha \preccurlyeq \mathsf{integ}(S) \vee \vdash \alpha \preccurlyeq \mathsf{persist}(S)\}$*. The notation* $\mathsf{integ}(S)$ *denotes the integrity of a record with type* S*: the least upper bound of its fields' integrity.*

$$\mathsf{integ}(\{\overrightarrow{x_i : \tau_i}\}_s) = \bigsqcup_i \mathsf{integ}(\tau_i)$$

We are now ready to define our notion of expression equivalence. The expression e_1 is considered to be equivalent to e_2 via a high-integrity homomorphism ϕ, written $\phi(e_1) \approx_\alpha e_2$, if e_1 is equal to e_2 (modulo bracketed expressions) when the memory locations in e_1 are transformed via ϕ. This is defined formally in Figure 3.12.

To ensure that high-integrity dereferences yield equivalent results, we also define an equivalence relation on memories: M_1 and M_2 are equivalent via ϕ, written $\phi(M_1) \approx_\alpha M_2$, if whenever $m^S \in \mathsf{dom}(\phi)$ is not deleted, then $\phi(m^S)$ maps to an equivalent record. We also require that if $m^S \in \mathsf{dom}(\phi)$ is a deleted high-

$$\frac{}{\phi(x) \approx_\alpha x} \qquad \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\phi(b) \approx_\alpha b} \qquad \frac{}{\phi(*) \approx_\alpha *} \qquad \frac{\phi(m_1^S) = m_2^S}{\phi(m_1^S) \approx_\alpha m_2^S}$$

$$\frac{\phi(e) \approx_\alpha e'}{\phi(\lambda(x{:}\tau)[pc; \mathcal{H}].\, e) \approx_\alpha \lambda(x{:}\tau)[pc; \mathcal{H}].\, e'} \qquad \frac{}{\phi(\bot_p) \approx_\alpha \bot_p}$$

$$\frac{\phi(v_i) \approx_\alpha v_i'^{\ (\forall i)}}{\phi(v_1\ v_2) \approx_\alpha v_1'\ v_2'} \qquad \frac{\phi(e_i) \approx_\alpha e_i'^{\ (\forall i)}}{\phi(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) \approx_\alpha \mathsf{if}\ e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'}$$

$$\frac{\phi(v_i) \approx_\alpha v_i'^{\ (\forall i)}}{\phi(\{\overrightarrow{x_i = v_i}\}^S) \approx_\alpha \{\overrightarrow{x_i = v_i'}\}^S} \qquad \frac{\phi(v) \approx_\alpha v'}{\phi(v.x) \approx_\alpha v'.x} \qquad \frac{\phi(v_i) \approx_\alpha v_i'^{\ (\forall i)}}{\phi(v_1.x := v_2) \approx_\alpha v_1'.x := v_2'}$$

$$\frac{\phi(e) \approx_\alpha e'}{\phi(\mathsf{soft}\ e) \approx_\alpha \mathsf{soft}\ e'} \qquad \frac{}{\phi([e]) \approx_\alpha [e']} \qquad \frac{\phi(e_i) \approx_\alpha e_i'^{\ (\forall i)}}{\phi(e_1 \| e_2) \approx_\alpha e_1' \| e_2'}$$

$$\frac{\phi(e_i) \approx_\alpha e_i'^{\ (\forall i)}}{\phi(\mathsf{exists}\ e_1\ \mathsf{as}\ x :\ e_2\ \mathsf{else}\ e_3) \approx_\alpha \mathsf{exists}\ e_1'\ \mathsf{as}\ x :\ e_2'\ \mathsf{else}\ e_3'}$$

$$\frac{\phi(e_i) \approx_\alpha e_i'^{\ (\forall i)}}{\phi(\mathsf{try}\ e_1\ \mathsf{catch}\ p{:}\ e_2) \approx_\alpha \mathsf{try}\ e_1'\ \mathsf{catch}\ p{:}\ e_2'} \qquad \frac{\phi(e_i) \approx_\alpha e_i'^{\ (\forall i)}}{\phi(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \approx_\alpha \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2'}$$

Figure 3.12: Equivalence of expressions in $[\lambda_{persist}]$

authority, high-persistence location, then so is $\phi(m^S)$. Formally,

$$\begin{aligned}
\phi(M_1) \approx_\alpha M_2 \overset{def.}{\iff} \forall m^S \in \mathsf{dom}(\phi).\ & \\
(M_1(m^S) \neq \bot & \\
\Rightarrow M_2(\phi(m^S)) \neq \bot \wedge \phi(M_1(m^S)) \approx_\alpha M_2(\phi(m^S))) & \\
\wedge (\vdash \alpha \preccurlyeq \mathsf{auth}^+(S) \sqcap \mathsf{persist}(S) \wedge M_1(m^S) = \bot & \\
\Rightarrow M_2(\phi(m^S)) = \bot) &
\end{aligned}$$

Together, these two equivalence definitions induce a natural equivalence relation on configurations:

$$\phi\langle e_1, M_1\rangle \approx_\alpha \langle e_2, M_2\rangle \overset{def.}{\iff} \phi(e_1) \approx_\alpha e_2 \wedge \phi(M_1) \approx_\alpha M_2.$$

A $[\lambda_{persist}]$ program has limited adversary influence if equivalent initial configurations produce equivalent final configurations.

We use this equivalence relation to formalize referential integrity: the adversary must be unable to cause a persistence failure in the high-integrity parts

of the program (expression equivalence), and unable to cause the deletion of high-authority, high-persistence objects (memory equivalence).

3.7.5 Storage attacks

To formalize immunity to storage attacks, we first show that the adversary is unable to cause more high-persistence locations to be allocated. This is captured by the equivalence relation, since all high-persistence locations are mapped by the homomorphism.

We also need to show that the adversary is unable to cause more high-authority locations to become reachable through hard references. Lemma 20 shows that this is implied by limited adversary influence.

Lemma 15. *Let e be well-typed in a low-integrity context. Assume that if e is a memory location, then it is low-authority. If m^S is a GC root in e, then m^S must be low-authority:*

$$\begin{aligned}&\Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \wedge \alpha \not\in pc \\ &\wedge (e \in \mathsf{dom}(M) \Rightarrow \alpha \not\in \mathsf{auth}^+(\tau)) \wedge \mathsf{root}(m^S, e) \\ &\qquad \Rightarrow \alpha \not\in \mathsf{auth}^+(S)\end{aligned}$$

Proof. By induction on the derivation of $\mathsf{root}(m^S, e)$.

Case R1 ($e = m^S$):

Without loss of generality, assume $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$, where $\alpha \not\in a$. We have $\tau = (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top$, so $\mathsf{auth}^+(\tau) = \mathsf{auth}^+(S)$ and the result follows trivially.

Case R2 ($e = \mathsf{soft}\ e'$, where $\forall m_1^{S'}.\ e' \neq m_1^{S'}$):

From the derivation of $\mathsf{root}(m^S, e)$, we have $\mathsf{root}(m^S, e')$. From the typing of e, we have $\Gamma; pc; \mathcal{H} \vdash e' : R_w, \mathcal{X}$. Therefore, the induction hypothesis applies, and the result follows.

Case R3 ($e = \{\overrightarrow{x_i = v_i}\}^{S'}$):

From the derivation of $\mathsf{root}(m^S, e)$, we have $\mathsf{root}(m^S, v_c)$ for some c.

From the typing of e, we know $\Gamma; pc; \mathcal{H} \vdash v_c : \tau', \top$, where $\vdash \mathsf{auth}^+(\tau') \preccurlyeq pc$. From this, we also know $\alpha \notin \mathsf{auth}^+(\tau')$.

Therefore, the induction hypothesis applies, and the result follows.

Case R4 ($e = v.x$):

From the derivation of $\mathsf{root}(m^S, e)$, we have $\mathsf{root}(m^S, v)$.

From the typing of e, we have $\Gamma; pc; \mathcal{H} \vdash v : \tau', \top$, where either $\tau' = (\{\overrightarrow{x_i : \tau_i}\}_{(a^+, a^-, p)})_w$ with $\vdash a^+ \preccurlyeq pc$, or $\tau' = (\mathsf{soft}\ R)_w$ for some R. We therefore know $\alpha \notin \mathsf{auth}^+(\tau')$.

Therefore, the induction hypothesis applies, and the result follows.

Case R5 ($e = v_1.x := v_2$):

From the derivation of $\mathsf{root}(m^S, e)$, we have $\mathsf{root}(m^S, v_i)$, for some $i \in \{1, 2\}$.

From the typing of e, we know $\Gamma; pc; \mathcal{H} \vdash v_1 : \tau', \top$ (where either $\tau' = (\{\overrightarrow{x_i : \tau_i}\}_{(a^+, a^-, p)})_w$ with $\vdash a^+ \preccurlyeq pc$, or $\tau' = (\mathsf{soft}\ R)_w$ for some R) and $\Gamma; pc; \mathcal{H} \vdash v_2 : \tau'', \top$, where $\vdash \mathsf{auth}^+(\tau'') \preccurlyeq pc$. Therefore, $\alpha \notin \mathsf{auth}^+(\tau')$ and $\alpha \notin \mathsf{auth}^+(\tau'')$. So, the induction hypothesis applies for $i \in \{1, 2\}$, and the result follows.

Case R6 ($e = \lambda(x{:}\tau')[pc'; \mathcal{H}'].\, e'$):

From the typing of e, we know $\vdash_{wf} \tau : \mathsf{type}$, $\vdash pc' \preccurlyeq pc$, and $\Gamma, x{:}\tau'; pc'; \mathcal{H}' \vdash e' : \tau'', \mathcal{H}'$, where $\tau = \tau' \xrightarrow{pc', \mathcal{H}'} \tau''$. Since $\alpha \notin pc$, we have $\alpha \notin pc'$. From $\vdash_{wf} \tau : \mathsf{type}$, we know $\vdash \mathsf{auth}^+(\tau'') \preccurlyeq pc'$, so we therefore know $\alpha \notin \mathsf{auth}^+(\tau'')$. From the derivation of $\mathsf{root}(m^S, e)$, we know $\mathsf{root}(m^S, e')$. Therefore, the induction hypothesis applies, and the result follows.

Case R7 ($e = v_1\ v_2$):

From the derivation of $\mathsf{root}(m^S, e)$, we have $\mathsf{root}(m^S, v_i)$ for some $i \in \{1, 2\}$.

From the typing of e, we know $\Gamma; pc; \mathcal{H} \vdash v_1 : (\tau' \xrightarrow{pc', \mathcal{H}'} \tau)_w, \top$, $\Gamma; pc; \mathcal{H} \vdash v_2 : \tau', \top$, and $\vdash_{wf} (\tau' \xrightarrow{pc', \mathcal{H}'} \tau)_w : \mathsf{type}$, with $\vdash pc' \preccurlyeq pc$. Therefore, $\alpha \notin pc'$.

From the derivation of $\vdash_{wf} (\tau' \xrightarrow{pc', \mathcal{H}'} \tau)_w : \mathsf{type}$, we have $\vdash \mathsf{auth}^+(\tau') \preccurlyeq pc$ and $\vdash \mathsf{auth}^+(\tau) \preccurlyeq pc$. We therefore know $\alpha \notin \mathsf{auth}^+(\tau')$.

So, the induction hypothesis applies for $i \in \{1, 2\}$, and the result follows.

Case R8 ($e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$):

From the derivation of $\mathsf{root}(m^S, e)$, we have $\mathsf{root}(m^S, e_i)$, for some $i \in \{1, 2\}$.

From the typing of e, we know $\Gamma; pc; \mathcal{H} \vdash e_1 : \tau', \mathcal{X}_1$ and $\Gamma, x : \tau'; pc'; \mathcal{H} \vdash e_2 : \tau'', \mathcal{X}_2$, where $\vdash \mathsf{auth}^+(\tau') \preccurlyeq pc$, $pc' = pc \sqcap w$, and $\vdash \mathsf{auth}^+(\tau'') \preccurlyeq pc'$, for some w. Therefore, it follows that $\alpha \notin pc'$, $\alpha \notin \mathsf{auth}^+(\tau')$, and $\alpha \notin \mathsf{auth}^+(\tau'')$.

So, the induction hypothesis applies for $i \in \{1, 2\}$, and the result follows.

Case R9 ($e = \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$):

From the typing of e, we know $\Gamma; pc; \mathcal{H} \vdash v : \mathsf{bool}_w, \top$ and $\Gamma; pc \sqcap w; \mathcal{H} \vdash e_i : \tau', \mathcal{X}_i$ (for $i \in \{1, 2\}$), where $\vdash \mathsf{auth}^+(\tau') \preccurlyeq pc$. From this, we therefore know $\alpha \notin \mathsf{auth}^+(\tau')$.

From the derivation of $\mathsf{root}(m^S, e)$, we have either $\mathsf{root}(m^S, v)$ or $\mathsf{root}(m^S, e_j)$, for some $j \in \{1, 2\}$. In all cases, the induction hypothesis applies, and the result follows.

Case R10 ($e = \mathsf{exists}\ v\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2$):

From the typing of e, we know $\Gamma; pc; \mathcal{H} \vdash v : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_r)_w, \top$ and $\Gamma_i; pc \sqcap w'; \mathcal{H} \vdash e_i : \tau', \mathcal{X}_i$ (for $i \in \{1, 2\}$), where $\vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap w'$. From this, we therefore know $\alpha \notin \mathsf{auth}^+(\tau')$.

From the derivation of $\mathsf{root}(m^S, e)$, we have either $\mathsf{root}(m^S, v)$ or $\mathsf{root}(m^S, e_j)$, for some $j \in \{1, 2\}$. In all cases, the induction hypothesis applies, and the result follows.

Case R11 ($e = e_1 \| e_2$):

From the derivation of $\mathsf{root}(m^S, e)$, we have $\mathsf{root}(m^S, e_i)$ for some $i \in \{1, 2\}$. Without loss of generality, assume $i = 1$.

From the typing of e, we know $\Gamma; pc; \top \vdash e_1 : \tau_1, \top$ and $\vdash \mathsf{auth}^+(\tau_1) \preccurlyeq pc$. From this, we also know $\alpha \notin \mathsf{auth}^+(\tau_1)$.

Therefore, the induction hypothesis applies, and the result follows.

Case R12 ($e = \mathsf{try}\ e_1\ \mathsf{catch}\ p\text{:}\ e_2$):

From the derivation of $\mathsf{root}(m^S, e)$, we have $\mathsf{root}(m^S, e_i)$, for some $i \in \{1, 2\}$.

From the typing of e, we know $\Gamma; pc; \mathcal{H}, p \vdash e_1 : \tau', \mathcal{X}_1$ and $\Gamma; pc'; \mathcal{H} \vdash e_2 : \tau', \mathcal{X}_2$, where $\vdash pc' \preccurlyeq pc$ and $\vdash \mathsf{auth}^+(\tau') \preccurlyeq pc$. From this, we also know $\alpha \notin pc'$ and $\alpha \notin \mathsf{auth}^+(\tau')$.

So, the induction hypothesis applies for $i \in \{1, 2\}$, and the result follows.

Case R13 ($e = [e']$):

From the derivation of $\mathsf{root}(m^S, e)$, we have $\mathsf{root}(m^S, e')$.

From the typing of e, we know $\Gamma; pc'; \mathcal{H} \vdash e' : \tau', \mathcal{X}$, where $\vdash pc' \preccurlyeq pc$ and $\vdash \mathsf{auth}^+(\tau') \preccurlyeq pc$. From this, it follows that $\alpha \notin pc'$ and $\alpha \notin \mathsf{auth}^+(\tau')$.

Therefore, the induction hypothesis applies, and the result follows.

□

Lemma 16. *Let e_1 and e_2 be well-typed expressions, both of type τ, that are equivalent via a high-integrity homomorphism ϕ. If m^S is a high-authority GC root in e_1, then $\phi(m^S)$ is also a GC root in $\langle e_2, M_2 \rangle$.*

$$\begin{aligned} &\Gamma; pc; \mathcal{H} \vdash e_1 : \tau, \mathcal{X} \wedge \Gamma; pc; \mathcal{H} \vdash e_2 : \tau, \mathcal{X} \\ &\wedge\, \phi(e_1) \approx_\alpha e_2 \wedge \vdash \alpha \preccurlyeq \mathsf{auth}^+(S) \wedge \mathsf{root}(m^S, e_1) \\ &\quad \Rightarrow \mathsf{root}(\phi(m^S), e_2) \end{aligned}$$

Proof. By induction on the derivation of $\mathsf{root}(m^S, e_1)$. The proof proceeds by cases according to the syntax of e_1. The result holds vacuously in cases $e_1 = x$, $e_1 = \mathsf{true}$, $e_1 = \mathsf{false}$, $e_1 = *$, and $e_1 = \mathsf{soft}\ m_1^{S_1}$, since $\neg\mathsf{root}(m^S, e_1)$ in these cases.

Case $e_1 = m_1^{S_1}$:

Since $\mathsf{root}(m^S, m_1^{S_1})$, we must have $m_1^{S_1} = m^S$. Therefore, from $\phi(e_1) \approx_\alpha e_2$, we have $e_2 = \phi(m^S)$, and the result follows via Rule R1.

Case $e_1 = \lambda(x{:}\tau')[pc'; \mathcal{H}'].\, e_3$:

From $\phi(e_1) \approx_\alpha e_2$, we have $e_2 = \lambda(x : \tau')[pc'; \mathcal{H}'].\, e_4$, where $\phi(e_3) \approx_\alpha e_4$. From the typing of e_1 and e_2, we have $\Gamma, x : \tau'; pc'; \mathcal{H}' \vdash e_3 : \tau'', \mathcal{H}'$ and $\Gamma, x{:}\tau'; pc'; \mathcal{H}' \vdash e_4 : \tau'', \mathcal{H}'$, for some pc', $\mathcal{H}'$, and τ''. From the derivation of $\mathsf{root}(m^S, e_1)$, we know $\mathsf{root}(m^S, e_3)$. Therefore, we can apply the induction hypothesis to obtain $\mathsf{root}(\phi(m^S), e_4)$, and the result follows via Rule R6.

Case $e_1 = v_1\ v_2$:

From the derivation of $\mathsf{root}(m^S, e_1)$, we know $\mathsf{root}(m^S, v_1)$ or $\mathsf{root}(m^S, v_2)$. Without loss of generality, assume $\mathsf{root}(m^S, v_1)$. From $\phi(e_1) \approx_\alpha e_2$, we have $e_2 = v_3\ v_4$, where $\phi(v_1) \approx_\alpha v_3$. From the typing of e_1 and e_2, we have $\Gamma; pc; \mathcal{H} \vdash v_1 : \tau', \top$ and $\Gamma; pc; \mathcal{H} \vdash v_3 : \tau', \top$, for some τ'. Therefore, we can apply the induction hypothesis to obtain $\mathsf{root}(\phi(m^S), v_1)$, and the result follows via Rule R7.

Case e_1 = if e_3 then e_4 else e_5:

From the derivation of root(m^S, e_1), we know root(m^S, e_k) for some $k \in \{3, 4, 5\}$. From $\phi(e_1) \approx_\alpha e_2$, we have e_2 = if e_3' then e_4' else e_5', where $\phi(e_i) \approx_\alpha e_i'$ for $i \in \{3, 4, 5\}$. From the typing of e_1 and e_2, we have $\Gamma; pc'; \mathcal{H} \vdash e_k : \tau', \mathcal{X}'$ and $\Gamma; pc'; \mathcal{H} \vdash e_k' : \tau', \mathcal{X}'$, for some pc', τ', and $\mathcal{X}'$. Therefore, we can apply the induction hypothesis to obtain root$(\phi(m^S), e_k')$, and the result follows via Rule R9.

Case $e_1 = \{\overrightarrow{x_i = v_i}\}^{S_1}$:

From the derivation of root(m^S, e_1), we know root(m^S, v_k) for some k. From $\phi(e_1) \approx_\alpha e_2$, we have $e_2 = \{\overrightarrow{x_i = u_i}\}^{S_1}$, where $\phi(v_i) \approx_\alpha u_i$ for all i. From the typing of e_1 and e_2, we have $\Gamma; pc; \mathcal{H} \vdash e_k : \tau', \top$ and $\Gamma; pc; \mathcal{H} \vdash e_k' : \tau', \top$, for some τ'. Therefore, we can apply the induction hypothesis to obtain root$(\phi(m^S), u_k)$, and the result follows via Rule R3.

Case $e_1 = v.x$:

From the derivation of root(m^S, e_1), we know root(m^S, v). From $\phi(e_1) \approx_\alpha e_2$, we have $e_2 = u.x$, where $\phi(v) \approx_\alpha u$. From the typing of e_1 and e_2, we have $\Gamma; pc; \mathcal{H} \vdash v : \tau', \top$ and $\Gamma; pc; \mathcal{H} \vdash u : \tau', \top$, for some τ'. Therefore, we can apply the induction hypothesis to obtain root$(\phi(m^S), u)$, and the result follows via Rule R4.

Case $e_1 = v_1.x := v_2$:

From the derivation of root(m^S, e_1), we know root(m^S, v_k) for some k. From $\phi(e_1) \approx_\alpha e_2$, we have $e_2 = u_1.x := u_2$, where $\phi(v_i) \approx_\alpha u_i$ for $i \in \{1, 2\}$. From the typing of e_1 and e_2, we have $\Gamma; pc; \mathcal{H} \vdash v_k : \tau', \top$ and $\Gamma; pc; \mathcal{H} \vdash u_k : \tau', \top$, for some τ'. Therefore, we can apply the induction hypothesis to obtain root$(\phi(m^S), u_k)$, and the result follows via Rule R5.

Case e_1 = soft e_3:

From the derivation of root(m^S, e_1), we know root(m^S, e_3) and e_3 is not a memory location. From $\phi(e_1) \approx_\alpha e_2$, we have e_2 = soft e_4, where $\phi(e_3) \approx_\alpha e_4$; therefore, e_4 is not a memory location. From the typing of e_1 and e_2, we have $\Gamma; pc; \mathcal{H} \vdash e_3 : \tau', \mathcal{X}$ and $\Gamma; pc; \mathcal{H} \vdash e_4 : \tau', \mathcal{X}$, for some τ'. Therefore, we can apply the induction hypothesis to obtain root$(\phi(m^S), e_4)$, and the result follows via Rule R2.

Case $e_1 = e_3 \| e_4$:

From the derivation of root(m^S, e_1), we know root(m^S, e_k) for some $k \in \{3, 4\}$. From $\phi(e_1) \approx_\alpha e_2$, we have $e_2 = e_3' \| e_4'$, where $\phi(e_i) \approx_\alpha e_i'$ for $i \in \{3, 4\}$. From the typing of e_1 and e_2, we have $\Gamma; pc; \top \vdash e_k : \tau', \top$ and $\Gamma; pc; \top \vdash e_k' : \tau', \top$, for some τ'. Therefore, we can apply the induction hypothesis to obtain root$(\phi(m^S), e_k')$, and the result follows via Rule R11.

Case e_1 = exists e_3 as $x : e_4$ else e_5:

From the derivation of root(m^S, e_1), we know root(m^S, e_k) for some $k \in \{3, 4, 5\}$. From $\phi(e_1) \approx_\alpha e_2$, we have e_2 = exists e_3' as $x : e_4'$ else e_5', where $\phi(e_i) \approx_\alpha e_i'$ for $i \in \{3, 4, 5\}$. From the typing of e_1 and e_2, we have $\Gamma'; pc'; \mathcal{H} \vdash e_k : \tau', \mathcal{X}'$ and $\Gamma'; pc'; \mathcal{H} \vdash e_k' : \tau', \mathcal{X}'$, for some Γ', pc', τ' and $\mathcal{X}'$. Therefore, we can apply the induction hypothesis to obtain root$(\phi(m^S), e_k')$, and the result follows via Rule R10.

Case e_1 = let $x = e_3$ in e_4:

From the derivation of root(m^S, e_1), we know root(m^S, e_k) for some $k \in \{3, 4\}$. From $\phi(e_1) \approx_\alpha e_2$, we have e_2 = let $x = e_3'$ in e_4', where $\phi(e_i) \approx_\alpha e_i'$ for $i \in \{3, 4\}$. From the typing of e_1 and e_2, we have $\Gamma'; pc'; \mathcal{H} \vdash e_k : \tau', \mathcal{X}'$ and $\Gamma'; pc'; \mathcal{H} \vdash e_k' : \tau', \mathcal{X}'$, for some Γ', pc', τ', and $\mathcal{X}'$. Therefore, we can apply

the induction hypothesis to obtain $\mathsf{root}(\phi(m^S), e'_k)$, and the result follows via Rule R8.

Case $e_1 = \mathsf{try}\ e_3\ \mathsf{catch}\ p\text{:}\ e_4$:

From the derivation of $\mathsf{root}(m^S, e_1)$, we know $\mathsf{root}(m^S, e_k)$ for some $k \in \{3, 4\}$. From $\phi(e_1) \approx_\alpha e_2$, we have $e_2 = \mathsf{try}\ e'_3\ \mathsf{catch}\ p\text{:}\ e'_4$, where $\phi(e_i) \approx_\alpha e'_i$ for $i \in \{3, 4\}$. From the typing of e_1 and e_2, we have $\Gamma; pc'; \mathcal{H} \vdash e_k : \tau', \mathcal{X}'$ and $\Gamma; pc'; \mathcal{H} \vdash e'_k : \tau', \mathcal{X}'$, for some pc', τ', and $\mathcal{X}'$. Therefore, we can apply the induction hypothesis to obtain $\mathsf{root}(\phi(m^S), e'_k)$, and the result follows via Rule R12.

Case $e_1 = [e_3]$:

From the derivation of $\mathsf{root}(m^S, e_1)$, we know $\mathsf{root}(m^S, e_3)$. From the typing of e_1, we know $\Gamma; pc \sqcap \ell; \mathcal{H} \vdash e_3 : \tau', \mathcal{X}$ for some ℓ and τ', where $\alpha \not\preccurlyeq \ell$ and $\vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \sqcap \ell$. Therefore, by Lemma 15, we must have $\alpha \not\preccurlyeq \mathsf{auth}^+(S)$, which contradicts the assumption $\vdash \alpha \preccurlyeq \mathsf{auth}^+(S)$. So, this case holds vacuously.

□

Lemma 17. *Suppose m^S is a high-authority location that is a GC root in the well-typed value v. Then v must have high-authority type.*

$$\vdash \alpha \preccurlyeq \mathsf{auth}^+(S) \wedge \mathsf{root}(m^S, v) \wedge \varnothing; \top; \top \vdash v : \tau, \top$$
$$\Rightarrow \vdash \alpha \preccurlyeq \mathsf{auth}^+(\tau)$$

Proof. Since $\mathsf{root}(m^S, v)$, the value v must either be equal to m^S, be a lambda abstraction $\lambda(x{:}\tau)[pc; \mathcal{H}].\, e$, or be a bracketed value $[v']$.

Case $v = m^S$:

We therefore have $\varnothing; \top; \top \vdash m^S : \tau, \top$. So $\vdash \mathsf{auth}^+(S) \preccurlyeq \mathsf{auth}^+(\tau)$. Therefore, $\vdash \alpha \preccurlyeq \mathsf{auth}^+(\tau)$, as desired.

Case $v = \lambda(x{:}\tau)[pc; \mathcal{H}].\, e$:

From the derivation of $\mathsf{root}(m^S, v)$, we have $\mathsf{root}(m^S, e)$. So, we must have $\tau = \tau_1 \xrightarrow{pc,\mathcal{H}} \tau_2$, for some τ_1 and some τ_2. Therefore, $\mathsf{auth}^+(\tau) = pc$.

Suppose $\alpha \not\preccurlyeq pc$. From the typing of v, we know $x{:}\tau_1; pc; \mathcal{H} \vdash e : \tau_2, \mathcal{H}$ and $\vdash_{wf} \tau_1 \xrightarrow{pc,\mathcal{H}} \tau_2 : \mathsf{type}$. Therefore, we also know $\vdash \mathsf{auth}^+(\tau_2) \preccurlyeq pc$, and hence, $\alpha \not\preccurlyeq \mathsf{auth}^+(\tau_2)$. Since $\mathsf{root}(m^S, e)$, by Lemma 15, we must have $\alpha \not\preccurlyeq \mathsf{auth}^+(S)$, a contradiction. So, we must have $\vdash \alpha \preccurlyeq pc = \mathsf{auth}^+(\tau)$, as desired.

Case $v = [v']$:

From the derivation of $\mathsf{root}(m^S, v)$, we have $\mathsf{root}(m^S, v')$. From the typing derivation of v, we know $\varnothing; \ell; \top \vdash v' : \tau', \top$ and $\vdash \mathsf{auth}^+(\tau') \preccurlyeq \ell$, where $\alpha \not\preccurlyeq \ell$. Therefore, $\alpha \not\preccurlyeq \mathsf{auth}^+(\tau')$. Since $\mathsf{root}(m^S, v')$, by Lemma 15, we must have $\alpha \not\preccurlyeq \mathsf{auth}^+(S)$, a contradiction. This case therefore holds vacuously.

□

Lemma 18. *Suppose m^S is a high-authority location that is non-collectible in the configuration $\langle v, M\rangle$, where M is well-formed and the value v has type τ. Then τ must be high-authority.*

$$\vdash^{\alpha}_{[wf]} M \wedge \vdash \alpha \preccurlyeq \mathsf{auth}^+(S) \wedge \mathsf{nc}(m^S, \langle v, M\rangle) \wedge \varnothing; \top; \top \vdash v : \tau, \top$$
$$\Rightarrow \vdash \alpha \preccurlyeq \mathsf{auth}^+(\tau)$$

Proof. By induction on the derivation of $\mathsf{nc}(m^S, \langle v, M\rangle)$.

Case NC1:

We have $\mathsf{root}(m^S, v)$. The result follows via Lemma 17.

Case NC2:

We have $\mathsf{root}(m_1^{S_1}, v)$, $M(m_1^{S_1}) = \{\overrightarrow{x_i = v_i}\}$, and $\mathsf{nc}(m^S, \langle v_c, M\rangle)$ for some c. Without loss of generality, assume $S_1 = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$. From $\vdash^{\alpha}_{[wf]} M$, we have $\varnothing; \top; \top \vdash v_c : \tau_c, \top$. Therefore, we can apply the induction hypothesis to obtain $\vdash \alpha \preccurlyeq \mathsf{auth}^+(\tau_c)$. From $\vdash^{\alpha}_{[wf]} M$, we know $\vdash_{wf} S_1 : \mathsf{rectype}$, and therefore, $\vdash \mathsf{auth}^+(\tau_c) \preccurlyeq a$. Hence, $\vdash \alpha \preccurlyeq a = \mathsf{auth}^+(S_1)$. The result follows via Lemma 17.

□

Corollary 19. *Suppose m^S is a high-authority location that is non-collectible in the configuration $\langle m_1^{S_1}, M\rangle$, where M is well-formed. Then $m_1^{S_1}$ is high-integrity, high-authority, and high-persistence.*

$$\vdash^{\alpha}_{[wf]} M \wedge \vdash \alpha \preccurlyeq \mathsf{auth}^+(S) \wedge \mathsf{nc}(m^S, \langle m_1^{S_1}, M\rangle)$$
$$\Rightarrow \vdash \alpha \preccurlyeq \mathsf{integ}(S_1) \sqcap \mathsf{auth}^+(S_1) \sqcap \mathsf{persist}(S_1)$$

Proof. Since $\vdash^{\alpha}_{[wf]} M$, we must have $\vdash_{wf} S_1 : \mathsf{rectype}$, and so, $\vdash \mathsf{auth}^+(S_1) \preccurlyeq \mathsf{integ}(S_1) \sqcap \mathsf{persist}(S_1)$. It therefore suffices to show that $m_1^{S_1}$ is high-authority. This follows from Lemma 18. □

We now show that limited adversary influence is sufficient to demonstrate immunity to storage attacks.

Lemma 20. *Let $\langle e_1, M_1\rangle$ be a well-formed configuration, wherein e_1 has type τ. Let $\langle e_2, M_2\rangle$, wherein e_2 also has type τ, be a configuration that is well-formed in a non-adversarial setting. Assume the two configurations are equivalent via a high-integrity homomorphism ϕ. If m^S is a high-authority non-collectible location in $\langle e_1, M_1\rangle$, then $\phi(m^S)$ is also non-collectible in $\langle e_2, M_2\rangle$.*

$$\begin{aligned}
&\vdash^{\alpha}_{[wf]} \langle e_1, M_1\rangle \wedge \varnothing; pc; \mathcal{H} \vdash e_1 : \tau, \mathcal{X} \\
&\wedge \vdash_{[wf]} M_2 \wedge \varnothing; pc; \mathcal{H} \vdash e_2 : \tau, \mathcal{X} \\
&\wedge \phi\langle e_1, M_1\rangle \approx_\alpha \langle e_2, M_2\rangle \wedge \vdash \alpha \preccurlyeq \mathsf{auth}^+(S) \wedge \mathsf{nc}(m^S, \langle e_1, M_1\rangle) \\
&\qquad \Rightarrow \mathsf{nc}(\phi(m^S), \langle e_2, M_2\rangle)
\end{aligned}$$

Proof. By induction on the derivation of $\mathsf{nc}(m^S, \langle e_1, M_1\rangle)$.

Case NC1:

We have $\mathsf{root}(m^S, e_1)$. By Lemma 16, we therefore have $\mathsf{root}(\phi(m^S), e_2)$, and the result follows by NC1.

Case NC2:

We have $\mathsf{root}(m_1^{S_1}, e_1)$ and $\mathsf{nc}(m^S, \langle v_c, M_1\rangle)$ for some $m_1^{S_1}$, where $M_1(m_1^{S_1}) = \{\overrightarrow{x_i = v_i}\}$. It therefore follows that $\mathsf{nc}(m^S, \langle m_1^{S_1}, M_1\rangle)$, $\vdash^{\alpha}_{[wf]} \langle m_1^{S_1}, M_1\rangle$, and $\vdash^{\alpha}_{[wf]} \langle v_c, M_1\rangle$.

Assume $m_1^{S_1} \neq m^S$. (Otherwise, we would have $\mathsf{root}(m^S, e_1)$, and the argument for Case NC1 applies.)

Since m^S is high-authority and $\mathsf{nc}(m^S, \langle m_1^{S_1}, M_1\rangle)$, by Corollary 19, we know $m_1^{S_1}$ is high-integrity, high-authority, and high-persistence.

Since $\mathsf{root}(m_1^{S_1}, e_1)$ and $m_1^{S_1}$ is high-authority, by Lemma 16, we know $\mathsf{root}(\phi(m_1^{S_1}), e_2)$.

From $\phi\langle e_1, M_1\rangle \approx_\alpha \langle e_2, M_2\rangle$, we know $\phi(M_1) \approx_\alpha M_2$. So, from $M_1(m_1^{S_1}) \neq \bot$, we have $M_2(\phi(m_1^{S_1})) = \{\overrightarrow{x_i = u_i}\}$, where $\phi(v_c) \approx_\alpha u_c$, for some $\overrightarrow{u_i}$.

Since $\vdash^{\alpha}_{[wf]} M_1$ and $\vdash_{[wf]} M_2$, we also know $\varnothing; \top; \top \vdash v_c : \tau_c, \top$ and $\varnothing; \top; \top \vdash u_c : \tau_c, \top$, for some τ_c. Therefore, we can apply the induction hypothesis to get $\mathsf{nc}(\phi(m^S), \langle u_c, M_2\rangle)$.

The result follows via NC2.

□

3.7.6 Referential security

Lemma 21. *Let m^S be part of a group G that is collectible in $\langle e, M\rangle$. If m^S is non-collectible in $\left\langle m_1^{S_1}, M\right\rangle$, then $m_1^{S_1}$ must also be in G.*

$$\mathsf{gc}(G, \langle e, M\rangle) \wedge m^S \in G \wedge \mathsf{nc}(m^S, \left\langle m_1^{S_1}, M\right\rangle) \Rightarrow m_1^{S_1} \in G$$

Proof. By induction on the derivation of $\mathsf{nc}(m^S, \left\langle m_1^{S_1}, M\right\rangle)$.

Case NC1:

We have $\mathsf{root}(m^S, m_1^{S_1})$. From this, it follows that $m_1^{S_1} = m^S$, and the result follows trivially.

Case NC2:

We have $M(m_1^{S_1}) = \{\overrightarrow{x_i = v_i}\}$, and $\mathsf{nc}(m^S, \langle v_c, M\rangle)$ for some c. Suppose $\mathsf{root}(m_2^{S_2}, v_c)$ for some $m_2^{S_2} \in G$. From this, we know $\mathsf{root}(m_2^{S_2}, M(m_1^{S_1}))$, and from the definition of $\mathsf{gc}(G, \langle e, M\rangle)$, we have $m_1^{S_1} \in G$, as desired.

We proceed by cases according to the derivation of $\mathsf{nc}(m^S, \langle v_c, M\rangle)$ to find such an $m_2^{S_2}$.

Case NC1:

We have $\mathsf{root}(m^S, v_c)$, so choose $m_2^{S_2} = m^S$.

Case NC2:

We have $\mathsf{root}(m_3^{S_3}, v_c)$, $M(m_3^{S_3}) = \{\overrightarrow{x_i = u_i}\}$, and $\mathsf{nc}(m^S, \langle u_{c'}, M\rangle)$ for some $m_3^{S_3}$ and some c'. It therefore follows that $\mathsf{nc}(m^S, \langle m_3^{S_3}, M\rangle)$. So, we can apply the induction hypothesis to obtain $m_3^{S_3} \in G$. Therefore, we can choose $m_2^{S_2} = m_3^{S_3}$.

□

Lemma 22. *All locations in a collectible group are collectible:*

$$\mathsf{gc}(G, \langle e, M\rangle) \wedge m^S \in G \Rightarrow \neg\mathsf{nc}(m^S, \langle e, M\rangle).$$

Proof. By contradiction. Let $m^S \in G$ be such that $\mathsf{nc}(m^S, \langle e, M\rangle)$. We proceed by cases according to the derivation of $\mathsf{nc}(m^S, \langle e, M\rangle)$.

Case NC1:

We have $\mathsf{root}(m^S, e)$. But from the definition of $\mathsf{gc}(G, \langle e, M\rangle)$, no such m^S can exist; a contradiction.

Case NC2:

We have $\mathsf{root}(m_1^{S_1}, e)$, $M(m_1^{S_1}) = \{\overrightarrow{x_i = v_i}\}$, and $\mathsf{nc}(m^S, \langle v_c, M\rangle)$ for some c. From this, it follows that $\mathsf{nc}(m^S, \langle m_1^{S_1}, M\rangle)$. Therefore, by Lemma 21, we have $m_1^{S_1} \in G$. So, from the definition of $\mathsf{gc}(G, \langle e, M\rangle)$, we have $\neg\mathsf{root}(m_1^{S_1}, e)$, a contradiction.

□

Lemma 23. *Let $C \subseteq \mathsf{dom}(M)$ be a set of locations that are collectible in a configuration $\langle e, M\rangle$:*

$$\forall m^S \in C.\ \neg\mathsf{nc}(m^S, \langle e, M\rangle).$$

Then there is a collectible group that contains C. In particular, let G be the largest superset of C such that from every location in G, some location in C is reachable through a chain of hard references:

$$\forall m_0^{S_0} \in G.\ \exists m_1^{S_1} \in C.\ \mathsf{nc}(m_1^{S_1}, \langle m_0^{S_0}, M \rangle). \tag{3.13}$$

Then G is a collectible group: $\mathsf{gc}(G, \langle e, M \rangle)$.

Proof. Suppose G is not a collectible group. Then either G contains a GC root in e, or there is a location outside G with a hard reference into G.

Suppose G contains a GC root m^S in e: $m^S \in G \wedge \mathsf{root}(m^S, e)$. By construction of G, let $m_1^{S_1} \in C$ be a location reachable through a chain of hard references from m^S: $\mathsf{nc}(m_1^{S_1}, \langle m^S, M \rangle)$. If $M(m^S) = \bot$, then from the derivation of $\mathsf{nc}(m_1^{S_1}, \langle m^S, M \rangle)$, we must have $m^S = m_1^{S_1}$, and so from $\mathsf{root}(m^S, e)$, we know $\mathsf{nc}(m_1^{S_1}, \langle e, M \rangle)$, a contradiction. Otherwise, assume $M(m^S) \neq \bot$ and $m^S \neq m_1^{S_1}$. Let $\overrightarrow{v_i}$ be such that $M(m^S) = \{\overrightarrow{x_i = v_i}\}$. From the derivation of $\mathsf{nc}(m_1^{S_1}, \langle m^S, M \rangle)$, we know there exists a c such that $\mathsf{nc}(m_1^{S_1}, \langle v_c, M \rangle)$. Therefore, by NC2, we have $\mathsf{nc}(m_1^{S_1}, \langle e, M \rangle)$, a contradiction.

Otherwise, let $m^S \notin G$ be such that $M(m^S)$ has a hard reference to some $m_0^{S_0} \in G$: $\mathsf{root}(m_0^{S_0}, M(m^S))$. By construction of G, let $m_1^{S_1} \in C$ be a location reachable through a chain of hard references from $m_0^{S_0}$: $\mathsf{nc}(m_1^{S_1}, \langle m_0^{S_0}, M \rangle)$.

We presently show that $\mathsf{nc}(m_1^{S_1}, \langle M(m^S), M \rangle)$. If $M(m_0^{S_0}) = \bot$, then from the derivation of $\mathsf{nc}(m_1^{S_1}, \langle m_0^{S_0}, M \rangle)$, we must have $m_0^{S_0} = m_1^{S_1}$, and so from $\mathsf{root}(m_0^{S_0}, M(m^S))$, we know $\mathsf{nc}(m_1^{S_1}, \langle M(m^S), M \rangle)$. Otherwise, assume $M(m_0^{S_0}) \neq \bot$ and $m_0^{S_0} \neq m_1^{S_1}$. Let $\overrightarrow{v_i}$ be such that $M(m_0^{S_0}) = \{\overrightarrow{x_i = v_i}\}$. From the derivation of $\mathsf{nc}(m_1^{S_1}, \langle m_0^{S_0}, M \rangle)$, we know there exists a c such that $\mathsf{nc}(m_1^{S_1}, \langle v_c, M \rangle)$. Therefore, by NC2, we have $\mathsf{nc}(m_1^{S_1}, \langle M(m^S), M \rangle)$.

So, we know $\mathsf{nc}(m_1^{S_1}, \langle M(m^S), M \rangle)$. It therefore follows that $\mathsf{nc}(m_1^{S_1}, \langle m^S, M \rangle)$. So, $G \cup \{m^S\}$ is a set larger than G satisfying property (3.13), a contradiction. □

Lemma 24. *Let e_1 and e_2 be well-typed equivalent expressions, and suppose $\phi(m^S)$ is a high-authority GC root of e_2. Then m^S is a GC root of e_1.*

$$\begin{aligned}
&\phi(e_1) \approx_\alpha e_2 \wedge \Gamma; pc; \mathcal{H} \vdash e_1 : \tau, \mathcal{X} \wedge \Gamma; pc; \mathcal{H} \vdash e_2 : \tau, \mathcal{X} \\
&\wedge \vdash \alpha \preccurlyeq \mathsf{auth}^+(S) \wedge \mathsf{root}(\phi(m^S), e_2) \\
&\quad \Rightarrow \mathsf{root}(m^S, e_1)
\end{aligned}$$

Proof. By induction on the derivation of $\mathsf{root}(\phi(m^S), e_2)$. Since e_2 is well-typed, by Lemma 15, we know that case R13 holds vacuously. □

Lemma 25. *Suppose M_1 and M_2 are well-formed memories. Let ϕ be a high-integrity homomorphism such that $\phi(M_1) \approx_\alpha M_2$. Let m^S and $m_1^{S_1}$ be locations in M_1 mapped by ϕ. Assume $m_1^{S_1}$ is high-authority and high-persistence. If $\mathsf{nc}(\phi(m_1^{S_1}), \langle \phi(m^S), M_2 \rangle)$, then $\mathsf{nc}(m_1^{S_1}, \langle m^S, M_1 \rangle)$:*

$$\begin{aligned}
&\vdash^\alpha_{[wf]} M_1 \wedge \vdash^\alpha_{[wf]} M_2 \wedge \phi(M_1) \approx_\alpha M_2 \\
&\wedge \vdash \alpha \preccurlyeq \mathsf{auth}^+(S_1) \sqcap \mathsf{persist}(S_1) \wedge \mathsf{nc}(\phi(m_1^{S_1}), \left\langle \phi(m^S), M_2 \right\rangle) \\
&\quad \Rightarrow \mathsf{nc}(m_1^{S_1}, \left\langle m^S, M_1 \right\rangle).
\end{aligned}$$

Proof. By induction on the derivation of $\mathsf{nc}(\phi(m_1^{S_1}), \langle \phi(m^S), M_2 \rangle)$.

Case NC1:

We must have $\phi(m^S) = \phi(m_1^{S_1})$. Since ϕ is injective, we know $m^S = m_1^{S_1}$, so the result follows by NC1.

Case NC2:

We know there exists some $\overrightarrow{u_i}$ and some c such that $M_2(\phi(m^S)) = \{\overrightarrow{x_i = u_i}\}$ and $\mathsf{nc}(\phi(m_1^{S_1}), \langle u_c, M_2 \rangle)$. Assume $\phi(m^S) \neq \phi(m_1^{S_1})$. (Otherwise, the argument for Case NC1 applies.) Since $\phi(m_1^{S_1})$ is high-authority and $\mathsf{nc}(\phi(m_1^{S_1}), \langle \phi(m^S), M_2 \rangle)$, by Corollary 19, we know that m^S is high-integrity, high-authority and high-persistence. Since $\phi(M_1) \approx_\alpha M_2$ and

$M_2(\phi(m^S)) \neq \bot$, then we must have $M_1(m^S) \neq \bot$. So let $\overrightarrow{v_i}$ be such that $M_1(m^S) = \{\overrightarrow{x_i = v_i}\}$.

We show that $\mathsf{nc}(m_1^{S_1}, \langle v_c, M_1 \rangle)$. From this, the result follows via an application of NC2:

$$\frac{\mathsf{root}(m^S, m^S) \qquad M_1(m^S) = \{\overrightarrow{x_i = v_i}\} \qquad \mathsf{nc}(m_1^{S_1}, \langle v_c, M_1 \rangle)}{\mathsf{nc}(m_1^{S_1}, \langle m^S, M_1 \rangle).}$$

Consider the two cases in the derivation of $\mathsf{nc}(\phi(m_1^{S_1}), \langle u_c, M_2 \rangle)$:

Sub-case NC1:

We have $\mathsf{root}(\phi(m_1^{S_1}), u_c)$. From $\phi(M_1) \approx_\alpha M_2$, we know $\phi(v_c) \approx_\alpha u_c$. From $\vdash^\alpha_{[wf]} M_1$ and $\vdash^\alpha_{[wf]} M_2$, we know $\varnothing; \top; \top \vdash v_c : \tau_c, \top$ and $\varnothing; \top; \top \vdash u_c : \tau_c, \top$, for some τ_c. Therefore, we can apply Lemma 24 to get $\mathsf{root}(m_1^{S_1}, v_c)$, and the result follows via NC1.

Sub-case NC2:

Assume $\neg\mathsf{root}(\phi(m_1^{S_1}), u_c)$. (Otherwise, the argument in sub-case NC1 applies.)

We know there exists some $m_2^{S_2} \in \mathsf{dom}(M_2)$, some $\overrightarrow{u_i}'$ and some c' such that $\mathsf{root}(m_2^{S_2}, u_c)$, $M_2(m_2^{S_2}) = \{\overrightarrow{x_i = u_i'}\}$, and $\mathsf{nc}(\phi(m_1^{S_1}), \langle u_{c'}', M_2 \rangle)$. From this, it follows that $\mathsf{nc}(\phi(m_1^{S_1}), \langle m_2^{S_2}, M_2 \rangle)$. Since $\phi(m_1^{S_1})$ is high-authority and $\mathsf{nc}(\phi(m_1^{S_1}), \langle m_2^{S_2}, M_2 \rangle)$, by Corollary 19, we know that $m_2^{S_2}$ is high-integrity, high-authority, and high-persistence.

Since ϕ is a high-integrity homomorphism, there exists an $m_3^{S_3}$ such that $m_2^{S_2} = \phi(m_3^{S_3})$. Therefore, we can apply the induction hypothesis to get $\mathsf{nc}(m_1^{S_1}, \langle m_3^{S_3}, M_1 \rangle)$.

We have $\mathsf{root}(\phi(m_3^{S_3}), u_c)$. From $\phi(M_1) \approx_\alpha M_2$, we know $\phi(v_c) \approx_\alpha u_c$. From $\vdash^\alpha_{[wf]} M_1$ and $\vdash^\alpha_{[wf]} M_2$, we know $\varnothing; \top; \top \vdash v_c : \tau_c, \top$ and

$\varnothing; \top; \top \vdash u_c : \tau_c, \top$, for some τ_c. Therefore, we can apply Lemma 24 to get $\mathsf{root}(m_3^{S_3}, v_c)$.

Since $m_2^{S_2} = \phi(m_3^{S_3})$, from $\neg\mathsf{root}(\phi(m_1^{S_1}), u_c)$ and $\mathsf{root}(m_2^{S_2}, u_c)$, we know $m_3^{S_3} \neq m_1^{S_1}$. Therefore, from the derivation of $\mathsf{nc}(m_1^{S_1}, \langle m_3^{S_3}, M_1 \rangle)$, we know there exists some $\overrightarrow{v_i}'$ and some c'' such that $M_1(m_3^{S_3}) = \{\overrightarrow{x_i = v_i'}\}$ and $\mathsf{nc}(m_1^{S_1}, \langle v'_{c''}, M_1 \rangle)$.

The result follows via NC2:

$$\frac{\mathsf{root}(m_3^{S_3}, v_c) \qquad M_1(m_3^{S_3}) = \{\overrightarrow{x_i = v_i'}\} \qquad \mathsf{nc}(m_1^{S_1}, \langle v'_{c''}, M_1 \rangle)}{\mathsf{nc}(m_1^{S_1}, \langle v_c, M_1 \rangle)}$$

□

Lemma 26. *Let G be a collectible group in $\langle e, M \rangle$ with $m_1^{S_1} \in G$. If $\mathsf{nc}(m_1^{S_1}, \langle m^S, M \rangle)$, then $m^S \in G$:*

$$\mathsf{gc}(G, \langle e, M \rangle) \wedge m_1^{S_1} \in G \wedge \mathsf{nc}(m_1^{S_1}, \langle m^S, M \rangle) \Rightarrow m^S \in G$$

Proof. By induction on the derivation of $\mathsf{nc}(m_1^{S_1}, \langle m^S, M \rangle)$.

Case NC1:

We must have $m^S = m_1^{S_1}$, so the result follows trivially.

Case NC2:

We know there exists some $\overrightarrow{v_i}$ and some c such that $M(m^S) = \{\overrightarrow{x_i = v_i}\}$ and $\mathsf{nc}(m_1^{S_1}, \langle v_c, M \rangle)$. Consider the two cases in the derivation of $\mathsf{nc}(m_1^{S_1}, \langle v_c, M \rangle)$:

Sub-case NC1:

We have $\mathsf{root}(m_1^{S_1}, v_c)$, so we therefore know $\mathsf{root}(m_1^{S_1}, M(m^S))$. The result then follows from the definition of $\mathsf{gc}(G, \langle e, M \rangle)$.

Sub-case NC2:

We know there exists some location $m_2^{S_2}$ such that $\mathsf{root}(m_2^{S_2}, v_c)$ and $\mathsf{nc}(m_1^{S_1}, \langle m_2^{S_2}, M \rangle)$. So, by the induction hypothesis, we know $m_2^{S_2} \in G$. From $\mathsf{root}(m_2^{S_2}, v_c)$, we know $\mathsf{root}(m_2^{S_2}, M(m^S))$. The result then follows from the definition of $\mathsf{gc}(G, \langle e, M \rangle)$.

□

Lemma 27. *Let M_1 and M_2 be well-formed memories. Suppose $\phi\langle e_1, M_1\rangle \approx_\alpha \langle e_2, M_2\rangle$. Let G be a collectible group in $\langle e_1, M_1\rangle$, and let $\phi(G)$ denote $\{\phi(m^S) : m^S \in G \cap \mathsf{dom}(\phi)\}$. Let C represent the high-authority, high-persistence members of $G \cap \mathsf{dom}(\phi)$:*

$$C = \{m^S \in G \cap \mathsf{dom}(\phi) : \vdash \alpha \preccurlyeq \mathsf{auth}^+(S) \sqcap \mathsf{persist}(S)\}.$$

Then there exists a set G' such that $\phi(G')$ is a subset of $\phi(G)$, is a collectible group in $\langle e_2, M_2\rangle$, and contains all members of $\phi(C)$:

$$\begin{aligned} &\vdash^{\alpha}_{[wf]} M_1 \wedge \vdash^{\alpha}_{[wf]} M_2 \wedge \phi\langle e_1, M_1\rangle \approx_\alpha \langle e_2, M_2\rangle \wedge \mathsf{gc}(G, \langle e_1, M_1\rangle) \\ &\quad \Rightarrow \exists G'.\, \mathsf{gc}(\phi(G'), \langle e_2, M_2\rangle) \wedge \phi(C) \subseteq \phi(G') \subseteq \phi(G) \end{aligned}$$

Proof. First, we show that $\phi(C)$ is a set of collectible locations. Suppose it isn't. Then let $m^S \in G \cap \mathsf{dom}(\phi)$ be such that $\phi(m^S) \in \phi(C)$ is non-collectible: $\mathsf{nc}(\phi(m^S), \langle e_2, M_2\rangle)$. Since $\phi(m^S)$ is high-authority and high-persistence, by induction on the derivation of $\mathsf{nc}(\phi(m^S), \langle e_2, M_2\rangle)$, we can show that m^S must be non-collectible: $\mathsf{nc}(m^S, \langle e_1, M_1\rangle)$. Therefore, by Lemma 22, G cannot be a collectible group, a contradiction.

Let $G' \subseteq \mathsf{dom}(\phi)$ be such that $\phi(G')$ is the largest superset of $\phi(C)$ such that from every location in $\phi(G')$, some location in $\phi(C)$ is reachable through a chain of hard references: $\forall m_0^{S_0} \in \phi(G').\ \exists m_1^{S_1} \in \phi(C).\ \mathsf{nc}(m_1^{S_1}, \langle m_0^{S_0}, M_2\rangle)$. By Lemma 23, we know $\phi(G')$ is a collectible group.

We now show that $\phi(G')$ is also a subset of $\phi(G)$ by showing $G' \subseteq G$. Suppose $m^S \in G'$. By construction of G', let $m_1^{S_1} \in C$ be such that $\mathsf{nc}(\phi(m_1^{S_1}), \langle \phi(m^S), M_2 \rangle)$. From this, by Lemma 25, we know $\mathsf{nc}(m_1^{S_1}, \langle m^S, M_1 \rangle)$. By Lemma 26, then, we must have $m^S \in G$. So $G' \subseteq G$. □

Lemma 28 (Equivalence substitution)**.** *If* $\phi(e_1) \approx_\alpha e_2$ *and* $\phi(e_1') \approx_\alpha e_2'$*, then* $\phi(e_1\{e_1'/x\}) \approx_\alpha e_2\{e_2'/x\}$.

Proof. Simple induction on the derivation of $\phi(e_1) \approx_\alpha e_2$. □

Lemma 29 (Auto-bracketing equivalence)**.** *Auto-bracketing preserves equivalence:*

$$\phi(e_1) \approx_\alpha e_2 \Rightarrow \phi(e_1 \blacktriangleright_\alpha \tau) \approx_\alpha e_2 \blacktriangleright_\alpha \tau.$$

Lemma 30. *Suppose* $\phi(e_1) \approx_\alpha e_2$*. Let* m^S *be such that* $m^S \notin \mathsf{locs}(e_1)$ *and let* $\phi' = \phi[m^S \mapsto m_1^S]$*. Then* $\phi'(e_1) \approx_\alpha e_2$*.*

Lemma 31. *Evaluating in a low-integrity context preserves memory equivalence.*

Let $\langle e_1, M_1 \rangle$ *be a well-formed configuration, wherein* e_1 *is well-typed in a low-integrity context. Let* M_2 *be a well-formed memory and suppose* $\langle e_1, M_1 \rangle \xrightarrow{\mathsf{e}}{}^* \langle e_1', M_1' \rangle$*.*

$$\vdash^\alpha_{[wf]} \langle e_1, M_1 \rangle \wedge \alpha \not\leq pc \wedge \varnothing; pc; \mathcal{H} \vdash e_1 : \tau, \mathcal{X}$$
$$\wedge \vdash^\alpha_{[wf]} M_2 \wedge \langle e_1, M_1 \rangle \xrightarrow{\mathsf{e}}{}^* \langle e_1', M_1' \rangle .$$

Then the following holds.

1. $\phi(M_1) \approx_\alpha M_2 \Rightarrow \phi(M_1') \approx_\alpha M_2$ *and*
2. $\vdash_{[wf]} \langle e_1, M_1 \rangle \wedge \phi(M_2) \approx_\alpha M_1 \Rightarrow \phi(M_2) \approx_\alpha M_1'$.

Proof. If we can show this is true for the case where a single $\xrightarrow{\mathsf{e}}$ step is taken, then the rest follows by induction on the number of $\xrightarrow{\mathsf{e}}$ steps taken. (We know the induction hypothesis will apply because of Corollary 11 and Corollary 12.)

We show the single-step case by induction on the derivation of $\langle e_1, M_1\rangle \xrightarrow{\mathsf{e}} \langle e_1', M_1'\rangle$. The proof proceeds by cases according to the evaluation rules.

In cases SELECT, DANGLE-SELECT, SOFT-SELECT, DANGLE-ASSIGN, APPLY, EXISTS-TRUE, EXISTS-FALSE, TRY-VAL, TRY-CATCH, TRY-ESC, PARALLEL-RESULT, IF-TRUE, IF-FALSE, LET, FAIL-PROP, BRACKET-SELECT, BRACKET-SOFT-SELECT, BRACKET-ASSIGN, BRACKET-SOFT-ASSIGN, BRACKET-SOFT, BRACKET-EXISTS, BRACKET-APPLY, BRACKET-TRY, BRACKET-IF, BRACKET-LET, DOUBLE-BRACKET, and BRACKET-FAIL, the result holds trivially, since $M_1' = M_1$.

Case CREATE ($\left\langle \{\overrightarrow{x_i = v_i}\}^S, M_1\right\rangle \xrightarrow{\mathsf{e}} \left\langle m^S, M_1[m^S \mapsto \{\overrightarrow{x_i = v_i \blacktriangleright_\alpha \tau_i}\}]\right\rangle$, where m is fresh and $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$):

1. Suppose $\phi(M_1) \approx_\alpha M_2$. We show that $\phi(M_1') \approx_\alpha M_2$.

   From the derivation of $\varnothing; pc; \mathcal{H} \vdash e_1 : \tau, \mathcal{X}$, we know $\vdash p \preccurlyeq pc$ and $\vdash \mathsf{integ}(\tau_i) \preccurlyeq pc$ for all i. Therefore, we know m^S is neither high-integrity nor high-persistence. The result then follows from the fact that $m^S \notin \mathsf{dom}(\phi)$ and the assumption $\phi(M_1) \approx_\alpha M_2$.

2. Suppose $\vdash_{[wf]} \langle e_1, M_1\rangle$ and $\phi(M_2) \approx_\alpha M_1$. We show that $\phi(M_2) \approx_\alpha M_1'$.

   This follows from $\phi(M_2) \approx_\alpha M_1$.

Case ASSIGN ($\langle m^S.x_c := v, M_1\rangle \xrightarrow{\mathsf{e}} \langle * \blacktriangleright_\alpha p, M_1[m^S.x_c \mapsto v \blacktriangleright_\alpha \tau_c]\rangle$, where $M_1(m^S) \neq \bot$ and $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$):

Let $\overrightarrow{v_i}'$ and $\overrightarrow{u_i}$ be such that $M_1'(m^S) = \{\overrightarrow{x_i = v_i'}\}$ and $M_2(\phi(m^S)) = \{\overrightarrow{x_i = u_i}\}$. We therefore have $v_c' = v \blacktriangleright_\alpha \tau_c$.

From $\varnothing; pc; \mathcal{H} \vdash m^S.x_c := v : \tau, \mathcal{X}$, we know $\vdash \tau' \sqcap pc \leq \tau_c$, for some τ'. We therefore know $\vdash \mathsf{integ}(\tau_c) \preccurlyeq pc$. So, from $\alpha \not\preccurlyeq pc$, we have $\alpha \not\preccurlyeq \mathsf{integ}(\tau_c)$. Therefore, v_c' is a bracketed value: there exists a v_c'' such that $v_c' = [v_c'']$.

From $\vdash^{\alpha}_{[wf]} M_2$ and $\alpha \not\in \mathsf{integ}(\tau_c)$, we also know u_c is a bracketed value: $u_c = [u'_c]$, for some u'_c.

1. Suppose $\phi(M_1) \approx_\alpha M_2$. We show that $\phi(M'_1) \approx_\alpha M_2$.

   Assume $m^S \in \mathsf{dom}(\phi)$ (otherwise, the result follows directly from $\phi(M_1) \approx_\alpha M_2$). Since $\phi(M_1) \approx_\alpha M_2$, it suffices to show $\phi(v'_c) \approx_\alpha u_c$. This is trivial, since $v'_c = [v''_c]$ and $u_c = [u'_c]$.

2. Suppose $\vdash_{[wf]} \langle e_1, M_1\rangle$ and $\phi(M_2) \approx_\alpha M_1$. We show that $\phi(M_2) \approx_\alpha M'_1$.

   Assume $m^S \in \mathsf{im}(\phi)$ (otherwise, the result follows directly from $\phi(M_2) \approx_\alpha M_1$). Since $\phi(M_2) \approx_\alpha M_1$, it suffices to show $\phi(u_c) \approx_\alpha v'_c$. This is trivial, since $u_c = [u'_c]$ and $v'_c = [v''_c]$.

Case SOFT-ASSIGN ($\langle(\mathsf{soft}\ m^S).x_c := v, M_1\rangle \xrightarrow{\mathsf{e}} \langle e'_1 \blacktriangleright_\alpha (a \sqcap p), M'_1\rangle$, where $\langle m^S.x_c := v, M_1\rangle \xrightarrow{\mathsf{e}} \langle e'_1, M'_1\rangle$ and $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$):

We proceed by cases according to the evaluation rules for $\langle m^S.x_c := v, M_1\rangle \xrightarrow{\mathsf{e}} \langle e'_1, M'_1\rangle$.

Sub-case ASSIGN ($\langle m^S.x_c := v, M_1\rangle \xrightarrow{\mathsf{e}} \langle * \blacktriangleright_\alpha p, M_1[m^S.x_c \mapsto v \blacktriangleright_\alpha \tau_c]\rangle$):

Let $\overrightarrow{v_i}'$ and $\overrightarrow{u_i}$ be such that $M'_1(m^S) = \{\overrightarrow{x_i = v'_i}\}$ and $M_2(\phi(m^S)) = \{\overrightarrow{x_i = u_i}\}$. We therefore have $v'_c = v \blacktriangleright_\alpha \tau_c$.

From $\varnothing; pc; \mathcal{H} \vdash (\mathsf{soft}\ m^S).x_c := v : \tau, \mathcal{X}$, we know $\vdash \tau' \sqcap pc \leq \tau_c$, for some τ'. We therefore know $\vdash \mathsf{integ}(\tau_c) \preccurlyeq pc$. So, from $\alpha \not\in pc$, we have $\alpha \not\in \mathsf{integ}(\tau_c)$. Therefore, v'_c is a bracketed value: there exists a v''_c such that $v'_c = [v''_c]$.

From $\vdash^{\alpha}_{[wf]} M_2$ and $\alpha \not\in \mathsf{integ}(\tau_c)$, we also know u_c is a bracketed value: $u_c = [u'_c]$, for some u'_c.

1. Suppose $\phi(M_1) \approx_\alpha M_2$. We show $\phi(M_1') \approx_\alpha M_2$.

   Assume $m^S \in \mathsf{dom}(\phi)$ (otherwise, the result follows directly from $\phi(M_1) \approx_\alpha M_2$). Since $\phi(M_1) \approx_\alpha M_2$, it suffices to show $\phi(v_c') \approx_\alpha u_c$. This is trivial, since $v_c' = [v_c'']$ and $u_c = [u_c']$.

2. Suppose $\vdash_{[wf]} \langle e_1, M_1 \rangle$ and $\phi(M_2) \approx_\alpha M_1$. We show $\phi(M_2) \approx_\alpha M_1'$.

   Assume $m^S \in \mathsf{im}(\phi)$ (otherwise, the result follows directly from $\phi(M_2) \approx_\alpha M_1$). Since $\phi(M_2) \approx_\alpha M_1$, it suffices to show $\phi(u_c) \approx_\alpha v_c'$. This is trivial, since $u_c = [u_c']$ and $v_c' = [v_c'']$.

Sub-case DANGLE-ASSIGN ($\langle m^S.x_c := v, M_1 \rangle \xrightarrow{\mathsf{e}} \langle \bot_p \blacktriangleright_\alpha p, M_1 \rangle$):

The result holds trivially, since $M_1' = M_1$.

Case EVAL-CONTEXT ($\langle E[e_3], M_1 \rangle \xrightarrow{\mathsf{e}} \langle E[e_3'], M_1' \rangle$, where $\langle e_3, M_1 \rangle \xrightarrow{\mathsf{e}} \langle e_3', M_1' \rangle$):

A case analysis on the syntax of $E[\cdot]$ shows that from the derivation of $\varnothing; pc; \mathcal{H} \vdash E[e_3] : \tau, \mathcal{X}$, we know $\varnothing; pc; \mathcal{H}' \vdash e_3 : \tau', \mathcal{X}'$, for some $\mathcal{H}'$, τ', and $\mathcal{X}'$. Therefore, we can apply the induction hypothesis and obtain the result.

Case BRACKET-CONTEXT ($\langle [e_3], M_1 \rangle \xrightarrow{\mathsf{e}} \langle [e_3'], M_1' \rangle$, where $\langle e_3, M_1 \rangle \xrightarrow{\mathsf{e}} \langle e_3', M_1' \rangle$):

From the derivation of $\varnothing; pc; \mathcal{H} \vdash [e_3] : \tau, \mathcal{X}$, we know $\varnothing; pc \sqcap \ell; \mathcal{H} \vdash e_3 : \tau', \mathcal{X}$, where $\alpha \not\leq \ell$ and $\tau = \tau' \sqcap \ell$. Therefore, we can apply the induction hypothesis and obtain the result.

□

We can now state our referential security theorem, which encompasses both referential integrity and, via Lemma 20, immunity to storage attacks.

Theorem 1 (Referential security)**.** *Let $\langle e_1, M_1\rangle$ be a well-formed configuration wherein e_1 has type τ. Let e_2 be an expression also of type τ, and let M_2 be well-formed, such that $\langle e_2, M_2\rangle$ is a well-formed non-adversarial configuration. Let ϕ be a high-integrity homomorphism from M_1 to M_2 such that $\langle e_1, M_1\rangle$ is equivalent to $\langle e_2, M_2\rangle$ via ϕ. Suppose $\langle e_1, M_1\rangle$ takes some number of steps in the presence of an adversary to another configuration $\langle e_1', M_1'\rangle$.*

$$\begin{aligned}
&\vdash^{\alpha}_{[wf]} \langle e_1, M_1\rangle \wedge \varnothing; pc; \mathcal{H} \vdash e_1 : \tau, \mathcal{X} \\
&\wedge \vdash_{[wf]} \langle e_2, M_2\rangle \wedge \varnothing; pc; \mathcal{H} \vdash e_2 : \tau, \mathcal{X} \wedge \vdash^{\alpha}_{[wf]} M_2 \\
&\wedge \phi\langle e_1, M_1\rangle \approx_\alpha \langle e_2, M_2\rangle \wedge \langle e_1, M_1\rangle \rightarrow^*_\alpha \langle e_1', M_1'\rangle
\end{aligned}$$

Then either $\langle e_2, M_2\rangle$ diverges, or it can take some number of steps in the absence of an adversary to another configuration

$$\exists e_2', M_2'.\ \langle e_2, M_2\rangle \rightarrow^* \langle e_2', M_2'\rangle,$$

and there exists a high-integrity homomorphism ϕ' from M_1' to M_2' that extends ϕ, such that $\langle e_1', M_1'\rangle$ is equivalent to $\langle e_2', M_2'\rangle$ via ϕ':

$$\phi'\langle e_1', M_1'\rangle \approx_\alpha \langle e_2', M_2'\rangle$$

Proof. If we can show this is true for the case where a single $\rightarrow_\alpha$ step is taken to reach $\langle e_1', M_1'\rangle$, then the rest follows by induction on the number of $\rightarrow_\alpha$ steps taken. (We know the induction hypothesis will apply because of Corollary 11 and the fact that $\vdash_{[wf]} \langle e_2, M_2\rangle \wedge \vdash^{\alpha}_{[wf]} M_2 \Rightarrow \vdash^{\alpha}_{[wf]} \langle e_2, M_2\rangle$.)

We show the single-step case by induction on the derivation of $\langle e_1, M_1\rangle \rightarrow_\alpha \langle e_1', M_1'\rangle$. Assume that $\langle e_2, M_2\rangle$ does not diverge. The proof proceeds by cases

according to the evaluation rules. For each case, we need to show two things about e_2', M_2', and ϕ':

i. Expression equivalence: $\phi'(e_1') \approx_\alpha e_2'$ and

ii. Memory equivalence: $\phi'(M_1') \approx_\alpha M_2'$.

It will be obvious by its construction that ϕ' is a high-integrity homomorphism that extends ϕ.

Case CREATE ($\left\langle \{\overrightarrow{x_i = v_i}\}^S, M_1 \right\rangle \to_\alpha \left\langle m_1^S, M_1[m_1^S \mapsto \{\overrightarrow{x_i = v_i \blacktriangleright_\alpha \tau_i}\}] \right\rangle$, where m_1 is fresh and $S = \{\overrightarrow{x_i : \tau_i}\}_s$):

From $\phi(e_1) \approx_\alpha e_2$, we know $e_2 = \{\overrightarrow{x_i = u_i}\}^S$ with $\phi(v_i) \approx_\alpha u_i$ for all i. By CREATE,

$$\langle e_2, M_2 \rangle \to \left\langle m_2^S, M_2[m_2^S \mapsto \{\overrightarrow{x_i = u_i \blacktriangleright_\alpha \tau_i}\}] \right\rangle,$$

where m_2 is fresh. Choose $\phi' = \phi[m_1^S \mapsto m_2^S]$.

i. We need to show $\phi'(m_1^S) \approx_\alpha m_2^S$.

This follows by construction of ϕ'.

ii. We need to show $\phi'(M_1') \approx_\alpha M_2'$, where $M_1' = M_1[m_1^S \mapsto \{\overrightarrow{x_i = v_i \blacktriangleright_\alpha \tau_i}\}]$ and $M_2' = M_2[m_2^S \mapsto \{\overrightarrow{x_i = u_i \blacktriangleright_\alpha \tau_i}\}]$.

First, let $m^{S'} \in \mathsf{dom}(\phi')$ be such that $M_1'(m^{S'}) \neq \bot$. We show that $M_2'(\phi'(m^{S'})) \neq \bot$ and $\phi'(M_1'(m^{S'})) \approx_\alpha M_2'(\phi'(m^{S'}))$.

If $m^{S'} = m_1^S$, then $M_1'(m^{S'}) = M_1'(m_1^S) = \{\overrightarrow{x_i = v_i \blacktriangleright_\alpha \tau_i}\}$ and $M_2'(\phi'(m^{S'})) = M_2'(m_2^S) = \{\overrightarrow{x_i = u_i \blacktriangleright_\alpha \tau_i}\}$, so the result follows from the assumption $\phi(e_1) \approx_\alpha e_2$ via Lemmas 29 and 30. Otherwise, $m^{S'} \neq m_1^S$, so $M_1'(m^{S'}) = M_1(m^{S'})$ and $M_2'(\phi'(m^{S'})) = M_2(\phi(m^{S'}))$. The result therefore follows from the assumption $\phi(M_1) \approx_\alpha M_2$.

Now, let $m^{S'} \in \mathsf{dom}(\phi')$ be such that $\vdash \alpha \preccurlyeq \mathsf{auth}^+(S') \sqcap \mathsf{persist}(S')$ and $M_1'(m^{S'}) = \bot$. We show that $M_2'(\phi'(m^{S'})) = \bot$. Since $M_1'(m^{S'}) = \bot$, we must have $m^{S'} \neq m_1^S$, so $M_1'(m^{S'}) = M_1(m^{S'})$ and $M_2'(\phi'(m^{S'})) = M_2(\phi(m^{S'}))$. The result therefore follows from the assumption $\phi(M_1) \approx_\alpha M_2$.

Case SELECT ($\langle m_1^S.x_c, M_1\rangle \rightarrow_\alpha \langle v_c \blacktriangleright_\alpha p, M_1\rangle$, where $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$ and $M_1(m_1^S) = \{\overrightarrow{x_i = v_i}\}$):

From $\phi(e_1) \approx_\alpha e_2$, we know $e_2 = m_2^S.x_c$, where $\phi(m_1^S) = m_2^S$. Therefore, from $\phi(M_1) \approx_\alpha M_2$, we have $M_2(m_2^S) = \{\overrightarrow{x_i = u_i}\}$ for some $\overrightarrow{u_i}$, where $\phi(v_i) \approx_\alpha u_i$. So, by SELECT, $\langle e_2, M_2\rangle \rightarrow \langle u_c \blacktriangleright_\alpha p, M_2\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(v_c \blacktriangleright_\alpha p) \approx_\alpha u_c \blacktriangleright_\alpha p$.

   This follows via Lemma 29.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Case DANGLE-SELECT ($\langle m_1^S.x_c, M_1\rangle \rightarrow_\alpha \langle \bot_p \blacktriangleright_\alpha p, M_1\rangle$, where $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$ and $M_1(m_1^S) = \bot$):

From $\vdash^\alpha_{[wf]} \langle m_1^S.x_c, M_1\rangle$, $\mathsf{nc}(m_1^S, \langle m_1^S.x_c, M_1\rangle)$, and $M_1(m_1^S) = \bot$, we know $\alpha \not\preccurlyeq p$. Therefore, $\bot_p \blacktriangleright_\alpha p = [\bot_p]$. From $\phi(e_1) \approx_\alpha e_2$, we know $e_2 = m_2^S.x_c$.

If $M_2(m_2^S) = \bot$, then by DANGLE-SELECT, $\langle e_2, M_2\rangle \rightarrow \langle [\bot_p], M_2\rangle$.

Otherwise, without loss of generality, assume $M_2(m_2^S) = \{\overrightarrow{x_i = u_i}\}$. Since $\alpha \not\preccurlyeq p$, we have $u_c \blacktriangleright_\alpha p = [u_c']$ for some u_c'. So, by SELECT, $\langle e_2, M_2\rangle \rightarrow \langle [u_c'], M_2\rangle$.

Choose $\phi' = \phi$.

i. We need to show that $\phi([\bot_p]) \approx_\alpha [u]$ for $u \in \{u'_c, \bot_p\}$.

This is trivial.

ii. We need to show that $\phi(M_1) \approx_\alpha M_2$.

This is given.

Case SOFT-SELECT ($\langle(\mathsf{soft}\ m_1^S).x_c, M_1\rangle \rightarrow_\alpha \langle v', M_1\rangle$, where $\langle m_1^S.x_c, M_1\rangle \xrightarrow{\mathsf{e}} \langle v, M_1\rangle$, $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$), and $v' = v \blacktriangleright_\alpha (a \sqcap p)$):

From $\phi(e_1) \approx_\alpha e_2$, we know $e_2 = ((\mathsf{soft}\ m_2^S).x_c)$, where $\phi(m_1^S) = m_2^S$. If we can find u so that $\langle m_2^S.x_c, M_2\rangle \xrightarrow{\mathsf{e}} \langle u, M_2\rangle$, then by SOFT-SELECT, we have $\langle e_2, M_2\rangle = \langle(\mathsf{soft}\ m_2^S).x_c, M_2\rangle \rightarrow \langle u \blacktriangleright_\alpha (a \sqcap p), M_2\rangle$. Choose $\phi' = \phi$.

We proceed by cases according to the evaluation rules for $\langle m_1^S.x_c, M_1\rangle \xrightarrow{\mathsf{e}} \langle v, M_1\rangle$.

Sub-case SELECT ($v = v_c \blacktriangleright_\alpha p$, where $M_1(m_1^S) = \{\overrightarrow{x_i = v_i}\}$):

From $\phi(M_1) \approx_\alpha M_2$, we know $M_2(m_2^S) = \{\overrightarrow{x_i = u_i}\}$ for some $\overrightarrow{u_i}$, where $\phi(v_i) \approx_\alpha u_i$. So, by SELECT, we have $\langle m_2^S.x_c, M_2\rangle \xrightarrow{\mathsf{e}} \langle u_c \blacktriangleright_\alpha p, M_2\rangle$. Therefore, $u = u_c \blacktriangleright_\alpha p$.

i. We need to show $\phi((v_c \blacktriangleright_\alpha p) \blacktriangleright_\alpha (a \sqcap p)) \approx_\alpha (u_c \blacktriangleright_\alpha p) \blacktriangleright_\alpha (a \sqcap p)$.

This follows via Lemma 29.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

This is given.

Sub-case DANGLE-SELECT ($v = \bot_p \blacktriangleright_\alpha p$, where $M_1(m_1^S) = \bot$):

First, suppose $\vdash \alpha \preccurlyeq a \sqcap p$. Then $v \blacktriangleright_\alpha (a \sqcap p) = \bot_p$, $u \blacktriangleright_\alpha (a \sqcap p) = u$, and $\vdash \alpha \preccurlyeq p$. Therefore, we have $m_1^S \in \mathsf{dom}(\phi)$, and so, $\phi(m_1^S) = m_2^S$. From $\phi(M_1) \approx_\alpha M_2$, then, we know $M_2(m_2^S) = \bot$. So, by SELECT, we have $\langle m_2^S.x_c, M_2\rangle \xrightarrow{\mathsf{e}} \langle \bot_p, M_2\rangle$. Therefore, $u = \bot_p$.

i. We need to show $\phi(\bot_p) \approx_\alpha \bot_p$.

This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

This is given.

Now, suppose $\alpha \not\leq a \sqcap p$. Then $v \blacktriangleright_\alpha (a \sqcap p) = [\bot_p]$. If $M_2(m_2^S) = \bot$, then by DANGLE-SELECT, $\langle m_2^S.x_c, M_2\rangle \xrightarrow{\mathsf{e}} \langle \bot_p \blacktriangleright_\alpha p, M_2\rangle$. Otherwise, without loss of generality, assume $M_2(m_2^S) = \{\overrightarrow{x_i = u_i}\}$. So, by SELECT, $\langle m_2^S.x_c, M_2\rangle \xrightarrow{\mathsf{e}} \langle u_c \blacktriangleright_\alpha p, M_2\rangle$. Therefore, $u \blacktriangleright_\alpha (a \sqcap p) \in \{(\bot_p \blacktriangleright_\alpha p) \blacktriangleright_\alpha (a \sqcap p), (u_c \blacktriangleright_\alpha p) \blacktriangleright_\alpha (a \sqcap p)\} = \{[\bot_p], [u_c']\}$, for some u_c'.

i. We need to show $\phi([\bot_p]) \approx_\alpha [u']$ for $u' \in \{u_c', \bot_p\}$.

This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

This is given.

Case ASSIGN ($\langle m_1^S.x_c := v, M_1\rangle \to_\alpha \langle * \blacktriangleright_\alpha p, M_1[m_1^S.x_c \mapsto v \blacktriangleright_\alpha \tau_c]\rangle$, where $M_1(m_1^S) \neq \bot$ and $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$):

From $\phi(e_1) \approx_\alpha e_2$, we know $e_2 = (m_2^S.x_c := u)$ with $\phi(m_1^S) = m_2^S$ and $\phi(v) \approx_\alpha u$. From $\phi(M_1) \approx_\alpha M_2$, we have $M_2(m_2^S) \neq \bot$. So, by ASSIGN, $\langle m_2^S.x_c := u, M_2\rangle \to \langle * \blacktriangleright_\alpha p, M_2[m_2^S.x_c \mapsto u \blacktriangleright_\alpha \tau_c]\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(* \blacktriangleright_\alpha p) \approx_\alpha * \blacktriangleright_\alpha p$.

This follows via Lemma 29.

ii. We need to show $\phi(M_1[m_1^S.x_c \mapsto v \blacktriangleright_\alpha \tau_c]) \approx_\alpha M_2[m_2^S.x_c \mapsto u \blacktriangleright_\alpha \tau_c]$.

First, let $m_0^{S_0} \in \mathsf{dom}(\phi)$ be such that $M_1'(m_0^{S_0}) \neq \bot$. We show that $M_2'(\phi(m_0^{S_0})) \neq \bot$ and $\phi(M_1'(m_0^{S_0})) \approx_\alpha M_2'(\phi(m_0^{S_0}))$.

If $m_0^{S_0} = m_1^S$, then $\phi(m_0^{S_0}) = m_2^S$. Let $\overrightarrow{v_i}'$ and $\overrightarrow{u_i}'$ be such that $M_1'(m_1^S) = \{\overrightarrow{x_i = v_i'}\}$ and $M_2'(m_2^S) = \{\overrightarrow{x_i = u_i'}\}$. Since $\phi(v) \approx_\alpha u$, by Lemma 29, we

have $\phi(v \blacktriangleright_\alpha \tau_c) \approx_\alpha u \blacktriangleright_\alpha \tau_c$. Since $v'_c = v \blacktriangleright_\alpha \tau_c$ and $u'_c = u \blacktriangleright_\alpha \tau_c$, we therefore have $\phi(v'_c) \approx_\alpha u'_c$. Therefore, from $\phi(M_1) \approx_\alpha M_2$, it follows that $\phi(\{\overrightarrow{x_i = v'_i}\}) \approx_\alpha \{\overrightarrow{x_i = u'_i}\}$.

Otherwise, $m_0^{S_0} \neq m_1^S$, so $M'_1(m_0^{S_0}) = M_1(m_0^{S_0})$ and $M'_2(\phi(m_0^{S_0})) = M_2(\phi(m_0^{S_0}))$. The result therefore follows from $\phi(M_1) \approx_\alpha M_2$.

Now, let $m_0^{S_0} \in \mathsf{dom}(\phi)$ be such that $\vdash \alpha \preccurlyeq \mathsf{auth}^+(S_0) \sqcap \mathsf{persist}(S_0)$ and $M'_1(m_0^{S_0}) = \bot$. We show that $M'_2(\phi(m_0^{S_0})) = \bot$. This follows from $\phi(M_1) \approx_\alpha M_2$ by construction of M'_1 and M'_2.

Case DANGLE-ASSIGN ($\langle m_1^S.x_c := v, M_1\rangle \rightarrow_\alpha \langle \bot_p \blacktriangleright_\alpha p, M_1\rangle$, where $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$ and $M_1(m^S) = \bot$):

From $\vdash^\alpha_{[wf]} \langle m_1^S.x_c := v, M_1\rangle$, $\mathsf{nc}(m_1^S, \langle m_1^S.x_c := v, M_1\rangle)$, and $M_1(m_1^S) = \bot$, we know $\alpha \not\preccurlyeq p$. Therefore, $\bot_p \blacktriangleright_\alpha p = [\bot_p]$.

From $\phi(e_1) \approx_\alpha e_2$, we know $e_2 = m_2^S.x_c := u$ with $\phi(m_1^S) = m_2^S$ and $\phi(v) \approx_\alpha u$. If $M_2(m_2^S) = \bot$, then by DANGLE-ASSIGN, $\langle e_2, M_2\rangle \rightarrow \langle [\bot_p], M_2\rangle$.

Otherwise, $M_2(m_2^S) \neq \bot$. Since $\alpha \not\preccurlyeq p$, we have $* \blacktriangleright_\alpha p = [*]$. So, by ASSIGN, $\langle e_2, M_2\rangle \rightarrow \langle [*], M_2[m_2^S.x_c \mapsto u \blacktriangleright_\alpha \tau_c]\rangle$.

Choose $\phi' = \phi$.

i. We need to show $\phi([\bot_p]) \approx_\alpha [u']$ for $u' \in \{*, \bot_p\}$.

   This is trivial.

ii. We need to show $\phi(M'_1) \approx_\alpha M'_2$.

   First, let $m_0^{S_0} \in \mathsf{dom}(\phi)$ be such that $M'_1(m_0^{S_0}) \neq \bot$. We show that $M'_2(\phi(m_0^{S_0})) \neq \bot$ and $\phi(M'_1(m_0^{S_0})) \approx_\alpha M'_2(\phi(m_0^{S_0}))$.

   Since $M'_1 = M_1$ and $M'_1(m_0^{S_0}) \neq \bot$, we must have $m_0^{S_0} \neq m_1^S$, so $M'_1(m_0^{S_0}) = M_1(m_0^{S_0})$ and $M'_2(\phi(m_0^{S_0})) = M_2(\phi(m_0^{S_0}))$. The result therefore follows from $\phi(M_1) \approx_\alpha M_2$.

Now, let $m_0^{S_0} \in \mathsf{dom}(\phi)$ be such that $\vdash \alpha \preccurlyeq \mathsf{auth}^+(S_0) \sqcap \mathsf{persist}(S_0)$ and $M_1'(m_0^{S_0}) = \bot$. We show that $M_2'(\phi(m_0^{S_0})) = \bot$. This follows from $\phi(M_1) \approx_\alpha M_2$ by construction of M_2'.

Case SOFT-ASSIGN ($\langle(\mathsf{soft}\, m_1^S).x_c := v, M_1\rangle \rightarrow_\alpha \langle v' \blacktriangleright_\alpha (a \sqcap p), M_1'\rangle$, where $\langle m_1^S.x_c := v, M_1\rangle \xrightarrow{\mathsf{e}} \langle v', M_1'\rangle$ and $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$):

From $\phi(e_1) \approx_\alpha e_2$, we know $e_2 = ((\mathsf{soft}\, m_2^S).x_c := u)$ with $\phi(m_1^S) = m_2^S$ and $\phi(v) \approx_\alpha u$. If we can find a configuration $\langle u', M_2'\rangle$ so that $\langle m_2^S.x_c := u, M_2\rangle \xrightarrow{\mathsf{e}} \langle u', M_2'\rangle$, then by SOFT-ASSIGN, we have $\langle e_2, M_2\rangle = \langle(\mathsf{soft}\, m_2^S).x_c := u, M_2\rangle \rightarrow \langle u' \blacktriangleright_\alpha (a \sqcap p), M_2'\rangle$. Choose $\phi' = \phi$.

We proceed by cases according to the evaluation rules for $\langle m_1^S.x_c := v, M_1\rangle \xrightarrow{\mathsf{e}} \langle v', M_1'\rangle$.

Sub-case ASSIGN ($v' = * \blacktriangleright_\alpha p$ and $M_1' = M_1[m_1^S.x_c \mapsto v \blacktriangleright_\alpha \tau_c]$, where $M_1(m_1^S) \neq \bot$):

From $\phi(M_1) \approx_\alpha M_2$, we know $M_2(m_2^S) \neq \bot$. So, by ASSIGN, we have

$$\left\langle m_2^S.x_c := u, M_2\right\rangle \xrightarrow{\mathsf{e}} \left\langle * \blacktriangleright_\alpha p, M_2[m_2^S.x_c \mapsto u \blacktriangleright_\alpha \tau_c]\right\rangle.$$

So $u' = * \blacktriangleright_\alpha p$ and $M_2' = M_2[m_2^S.x_c \mapsto u \blacktriangleright_\alpha \tau_c]$.

i. We need to show $\phi((* \blacktriangleright_\alpha p) \blacktriangleright_\alpha (a \sqcap p)) \approx_\alpha (* \blacktriangleright_\alpha p) \blacktriangleright_\alpha (a \sqcap p)$.
This follows via Lemma 29.

ii. We need to show $\phi(M_1[m_1^S.x_c \mapsto v \blacktriangleright_\alpha \tau_c]) \approx_\alpha M_2[m_2^S.x_c \mapsto u \blacktriangleright_\alpha \tau_c]$.
First, let $m_0^{S_0} \in \mathsf{dom}(\phi)$ be such that $M_1'(m_0^{S_0}) \neq \bot$. We show that $M_2'(\phi(m_0^{S_0})) \neq \bot$ and $\phi(M_1'(m_0^{S_0})) \approx_\alpha M_2'(\phi(m_0^{S_0}))$.
If $m_0^{S_0} = m_1^S$, then $\phi(m_0^{S_0}) = m_2^S$. Let $\overrightarrow{v_i}'$ and $\overrightarrow{u_i}'$ be such that $M_1'(m_1^S) = \{\overrightarrow{x_i = v_i'}\}$ and $M_2'(m_2^S) = \{\overrightarrow{x_i = u_i'}\}$. Since $\phi(v) \approx_\alpha u$, by Lemma 29, we have $\phi(v \blacktriangleright_\alpha \tau_c) \approx_\alpha u \blacktriangleright_\alpha \tau_c$. Since $v_c' = v \blacktriangleright_\alpha \tau_c$

and $u'_c = u \blacktriangleright_\alpha \tau_c$, we therefore have $\phi(v'_c) \approx_\alpha u'_c$. Therefore, from $\phi(M_1) \approx_\alpha M_2$, it follows that $\phi(\{\overrightarrow{x_i = v'_i}\}) \approx_\alpha \{\overrightarrow{x_i = u'_i}\}$. Otherwise, $m_0^{S_0} \neq m_1^S$, so $M'_1(m_0^{S_0}) = M_1(m_0^{S_0})$ and $M'_2(\phi(m_0^{S_0})) = M_2(\phi(m_0^{S_0}))$. The result therefore follows from $\phi(M_1) \approx_\alpha M_2$.

Now, let $m_0^{S_0} \in \mathsf{dom}(\phi)$ be such that $\vdash \alpha \preccurlyeq \mathsf{auth}^+(S_0) \sqcap \mathsf{persist}(S_0)$ and $M'_1(m_0^{S_0}) = \bot$. We show that $M'_2(\phi(m_0^{S_0})) = \bot$. This follows from $\phi(M_1) \approx_\alpha M_2$ by construction of M'_1 and M'_2.

Sub-case DANGLE-ASSIGN ($v' = \bot_p \blacktriangleright_\alpha p$ and $M'_1 = M_1$, where $M_1(m_1^S) = \bot$):

First, suppose $\vdash \alpha \preccurlyeq a \sqcap p$. Then $v' \blacktriangleright_\alpha (a \sqcap p) = \bot_p$ and $u' \blacktriangleright_\alpha (a \sqcap p) = u'$, and from $\phi(M_1) \approx_\alpha M_2$, we know $M_2(m_2^S) = \bot$. So, by DANGLE-ASSIGN, we have $\langle m_2^S.x_c := u, M_2\rangle \xrightarrow{\mathsf{e}} \langle \bot_p \blacktriangleright_\alpha p, M_2\rangle$. Therefore, $u' = \bot_p \blacktriangleright_\alpha p = \bot_p$ and $M'_2 = M_2$.

i. We need to show $\phi(\bot_p) \approx_\alpha \bot_p$.

   This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Now, suppose $\alpha \not\preccurlyeq a \sqcap p$. Then $v' \blacktriangleright_\alpha (a \sqcap p) = [\bot_p]$.

If $M_2(m_2^S) = \bot$, then by DANGLE-ASSIGN, $\langle m_2^S.x_c := u, M_2\rangle \xrightarrow{\mathsf{e}} \langle \bot_p \blacktriangleright_\alpha p, M_2\rangle$. Otherwise, $M_2(m_2^S) \neq \bot$, and by ASSIGN, $\langle m_2^S.x_c := u, M_2\rangle \xrightarrow{\mathsf{e}} \langle * \blacktriangleright_\alpha p, M_2[m_2^S.x_c \mapsto u \blacktriangleright_\alpha \tau_c]\rangle$. Therefore, $u' \blacktriangleright_\alpha (a \sqcap p) \in \{(\bot_p \blacktriangleright_\alpha p) \blacktriangleright_\alpha (a \sqcap p), (* \blacktriangleright_\alpha p) \blacktriangleright_\alpha (a \sqcap p)\} = \{[\bot_p], [*]\}$.

i. We need to show $\phi([\bot_p]) \approx_\alpha [u']$ for $u' \in \{*, \bot_p\}$.

   This is trivial.

ii. We need to show $\phi(M'_1) \approx_\alpha M'_2$.

   First, let $m_0^{S_0} \in \mathsf{dom}(\phi)$ be such that $M'_1(m_0^{S_0}) \neq \bot$. We show that $M'_2(\phi(m_0^{S_0})) \neq \bot$ and $\phi(M'_1(m_0^{S_0})) \approx_\alpha M'_2(\phi(m_0^{S_0}))$.

Since $M_1' = M_1$ and $M_1'(m_0^{S_0}) \neq \bot$, we must have $m_0^{S_0} \neq m_1^S$, so $M_1'(m_0^{S_0}) = M_1(m_0^{S_0})$ and $M_2'(\phi(m_0^{S_0})) = M_2(\phi(m_0^{S_0}))$. The result therefore follows from $\phi(M_1) \approx_\alpha M_2$.

Now, let $m_0^{S_0} \in \mathsf{dom}(\phi)$ be such that $\vdash \alpha \preccurlyeq \mathsf{persist}(S_0)$ and $M_1'(m_0^{S_0}) = \bot$. We show that $M_2'(\phi(m_0^{S_0})) = \bot$. This follows from $\phi(M_1) \approx_\alpha M_2$ by construction of M_2'.

Case APPLY ($\langle(\lambda(x{:}\tau)[pc;\mathcal{H}].\, e_3)\ v_1, M_1\rangle \to_\alpha \langle e_3\{v_1/x\}, M_1\rangle$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = ((\lambda(x : \tau)[pc;\mathcal{H}].\, e_4)\ v_2)$, where $\phi(e_3) \approx_\alpha e_4$ and $\phi(v_1) \approx_\alpha v_2$. By APPLY, we have $\langle(\lambda(x{:}\tau)[pc;\mathcal{H}].\, e_4)\ v_2, M_2\rangle \to \langle e_4\{v_2/x\}, M_2\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(e_3\{v_1/x\}) \approx_\alpha e_4\{v_2/x\}$.

   This follows by Lemma 28.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Case EXISTS-TRUE ($\langle \mathsf{exists\ soft}\ m_1^S\ \mathsf{as}\ x : e_3\ \mathsf{else}\ e_4, M_1\rangle \to_\alpha$
$\langle (e_3\{m_1^S/x\}) \blacktriangleright_\alpha (a \sqcap p), M_1\rangle$, where $M_1(m_1^S) \neq \bot$ and $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = \mathsf{exists\ soft}\ m_2^S\ \mathsf{as}\ x : e_5\ \mathsf{else}\ e_6$, where $\phi(m_1^S) = m_2^S$ and $\phi(e_3) \approx_\alpha e_5$. Since $\phi(M_1) \approx_\alpha M_2$ and $M_1(m_1^S) \neq \bot$, we know $M_2(m_2^S) \neq \bot$, so by EXISTS-TRUE, we have

$$\left\langle \mathsf{exists\ soft}\ m_2^S\ \mathsf{as}\ x : e_5\ \mathsf{else}\ e_6, M_2\right\rangle \to \left\langle (e_5\{m_2^S/x\}) \blacktriangleright_\alpha (a \sqcap p), M_2\right\rangle$$

Choose $\phi' = \phi$.

i. We need to show $\phi((e_3\{m_1^S/x\}) \blacktriangleright_\alpha (a \sqcap p)) \approx_\alpha (e_5\{m_2^S/x\}) \blacktriangleright_\alpha (a \sqcap p)$.

   This follows by Lemmas 28 and 29.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

This is given.

Case EXISTS-FALSE ($\langle \mathsf{exists\ soft}\ m_1^S\ \mathsf{as}\ x : e_3\ \mathsf{else}\ e_4, M_1\rangle \rightarrow_\alpha \langle e_4 \blacktriangleright_\alpha (a \sqcap p), M_1\rangle$, where $S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}$ and $M_1(m_1^S) = \bot$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = \mathsf{exists\ soft}\ m_2^S\ \mathsf{as}\ x : e_5\ \mathsf{else}\ e_6$, where $\phi(m_1^S) = m_2^S$ and $\phi(e_4) \approx_\alpha e_6$. We proceed by cases according to whether $\vdash \alpha \preccurlyeq a \sqcap p$.

Sub-case $\vdash \alpha \preccurlyeq a \sqcap p$:

We therefore know $e_4 \blacktriangleright_\alpha (a \sqcap p) = e_4$ and $\vdash \alpha \preccurlyeq p$. So, from $\phi(M_1) \approx_\alpha M_2$ and $M_1(m_1^S) = \bot$, we know $M_2(m_2^S) = \bot$, so by EXISTS-FALSE, we have $\langle \mathsf{exists\ soft}\ m_2^S\ \mathsf{as}\ x : e_5\ \mathsf{else}\ e_6, M_2\rangle \rightarrow \langle e_6, M_2\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(e_4) \approx_\alpha e_6$.

This is given.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

This is given.

Sub-case $\alpha \not\preccurlyeq a \sqcap p$:

We therefore have $e_4 \blacktriangleright_\alpha (a \sqcap p) = [e_4']$ for some e_4'. If $M_2(m_2^S) = \bot$, then by EXISTS-FALSE, we have $\langle \mathsf{exists\ soft}\ m_2^S\ \mathsf{as}\ x : e_5\ \mathsf{else}\ e_6, M_2\rangle \rightarrow \langle [e_6'], M_2\rangle$, for some e_6'. Otherwise, by EXISTS-TRUE, we have $\langle \mathsf{exists\ soft}\ m_2^S\ \mathsf{as}\ x : e_5\ \mathsf{else}\ e_6, M_2\rangle \rightarrow \langle [e_5'], M_2\rangle$, for some e_5'. Choose $\phi' = \phi$.

i. We need to show $\phi([e_4']) \approx_\alpha [e_2'']$ for $e_2'' \in \{e_5', e_6'\}$.

This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

This is given.

Case TRY-VAL ($\langle \mathsf{try}\ v_1\ \mathsf{catch}\ p\colon e_3, M_1\rangle \rightarrow_\alpha \langle v_1, M_1\rangle$, where $\forall p'.\ v_1 \neq \bot_{p'}$ and $\forall v_1'.\ v_1 \neq [v_1']$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = (\mathsf{try}\ v_2\ \mathsf{catch}\ p\colon e_4)$, where $\phi(v_1) \approx_\alpha v_2$ and $\phi(e_3) \approx_\alpha e_4$. From this, it follows that by TRY-VAL, we have $\langle \mathsf{try}\ v_2\ \mathsf{catch}\ p\colon e_4, M_2\rangle \rightarrow \langle v_2, M_2\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(v_1) \approx_\alpha v_2$.

   This is given.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Case TRY-CATCH ($\langle \mathsf{try}\ \bot_{p'}\ \mathsf{catch}\ p\colon e_3, M_1\rangle \rightarrow_\alpha \langle e_3, M_1\rangle$, where $\vdash p \preccurlyeq p'$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = (\mathsf{try}\ \bot_{p'}\ \mathsf{catch}\ p\colon e_4)$, where $\phi(e_3) \approx_\alpha e_4$. By TRY-CATCH, we have $\langle \mathsf{try}\ \bot_{p'}\ \mathsf{catch}\ p\colon e_4, M_2\rangle \rightarrow \langle e_4, M_2\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(e_3) \approx_\alpha e_4$.

   This is given.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Case TRY-ESC ($\langle \mathsf{try}\ \bot_{p'}\ \mathsf{catch}\ p\colon e_3, M_1\rangle \rightarrow_\alpha \langle \bot_{p'}, M_1\rangle$, where $p \not\preccurlyeq p'$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = (\mathsf{try}\ \bot_{p'}\ \mathsf{catch}\ p\colon e_4)$. By TRY-ESC, we have $\langle \mathsf{try}\ \bot_{p'}\ \mathsf{catch}\ p\colon e_4, M_2\rangle \rightarrow \langle \bot_{p'}, M_2\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(\bot_{p'}) \approx_\alpha \bot_{p'}$.

   This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Case PARALLEL-RESULT ($\langle v_1 \| v_2, M_1 \rangle \to_\alpha \langle *, M_1 \rangle$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = u_1 \| u_2$, where $\phi(v_i) \approx_\alpha u_i$ for $i \in \{1, 2\}$. By PARALLEL-RESULT, we have $\langle u_1 \| u_2, M_2 \rangle \to \langle *, M_2 \rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(*) \approx_\alpha *$.

   This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Case IF-TRUE ($\langle \mathsf{if\ true\ then}\ e_3\ \mathsf{else}\ e_4, M_1 \rangle \to_\alpha \langle e_3, M_1 \rangle$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = (\mathsf{if\ true\ then}\ e_5\ \mathsf{else}\ e_6)$, where $\phi(e_3) \approx_\alpha e_5$. By IF-TRUE, we have $\langle \mathsf{if\ true\ then}\ e_5\ \mathsf{else}\ e_6, M_2 \rangle \to \langle e_5, M_2 \rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(e_3) \approx_\alpha e_5$.

   This is given.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Case IF-FALSE ($\langle \mathsf{if\ false\ then}\ e_3\ \mathsf{else}\ e_4, M_1 \rangle \to_\alpha \langle e_4, M_1 \rangle$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = (\mathsf{if\ false\ then}\ e_5\ \mathsf{else}\ e_6)$, where $\phi(e_4) \approx_\alpha e_6$. By IF-FALSE, we have $\langle \mathsf{if\ false\ then}\ e_5\ \mathsf{else}\ e_6, M_2 \rangle \to \langle e_6, M_2 \rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(e_4) \approx_\alpha e_6$.

   This is given.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Case LET ($\langle \mathsf{let}\ x = v_1\ \mathsf{in}\ e_3, M_1\rangle \rightarrow_\alpha \langle e_3\{v_1/x\}, M_1\rangle$, where $\forall p.\ v_1 \neq \bot_p$ and $\forall v_1'.\ v_1 \neq [v_1']$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = (\mathsf{let}\ x = v_2\ \mathsf{in}\ e_4)$, where $\phi(v_1) \approx_\alpha v_2$ and $\phi(e_3) \approx_\alpha e_4$. By LET, we have $\langle \mathsf{let}\ x = v_2\ \mathsf{in}\ e_4, M_2\rangle \rightarrow \langle e_4\{v_2/x\}, M_2\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(e_3\{v_1/x\}) \approx_\alpha e_4\{v_2/x\}$.

   This follows by Lemma 28.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Case EVAL-CONTEXT ($\langle E[e_3], M_1\rangle \rightarrow_\alpha \langle E[e_3'], M_1'\rangle$, where $\langle e_3, M_1\rangle \xrightarrow{\mathsf{e}} \langle e_3', M_1'\rangle$):

We proceed by cases according to the syntax of $E[\,\cdot\,]$. We only show the case $E[\,\cdot\,] = \mathsf{try}\ [\,\cdot\,]\ \mathsf{catch}\ p\colon e_4$; the other cases follow similarly.

Sub-case $E[\,\cdot\,] = (\mathsf{try}\ [\,\cdot\,]\ \mathsf{catch}\ p\colon e_4)$:

From $\phi(e_1) \approx_\alpha e_2$, we know $e_2 = (\mathsf{try}\ e_5\ \mathsf{catch}\ p\colon e_6)$, where $\phi(e_3) \approx_\alpha e_5$ and $\phi(e_4) \approx_\alpha e_6$. To apply the induction hypothesis, we need:

- $\vdash^{\alpha}_{[wf]} \langle e_3, M_1\rangle$ and $\vdash_{[wf]} \langle e_5, M_2\rangle$
  These follow from

  $$\vdash^{\alpha}_{[wf]} \langle \mathsf{try}\ e_3\ \mathsf{catch}\ p\colon e_4, M_1\rangle$$

  and

  $$\vdash_{[wf]} \langle \mathsf{try}\ e_5\ \mathsf{catch}\ p\colon e_6, M_2\rangle\,.$$

- $\varnothing; pc; \mathcal{H}' \vdash e_3 : \tau', \mathcal{X}'$ and $\varnothing; pc; \mathcal{H}' \vdash e_5 : \tau', \mathcal{X}'$, for some $\mathcal{H}'$, τ', $\mathcal{X}'$
  These follow from the typing derivations for $\mathsf{try}\ e_3\ \mathsf{catch}\ p\colon e_4$ and $\mathsf{try}\ e_5\ \mathsf{catch}\ p\colon e_6$.

- $\vdash^{\alpha}_{[wf]} M_2$ and $\phi\langle e_3, M_1\rangle \approx_\alpha \langle e_5, M_2\rangle$.

  These are given.

Therefore, we can apply the induction hypothesis to get a configuration $\langle e_5', M_2'\rangle$ and a high-integrity homomorphism ϕ' from M_1' to M_2' that extends ϕ, such that $\langle e_5, M_2\rangle \to \langle e_5', M_2'\rangle$ and

$$\phi'\langle e_3', M_1'\rangle \approx_\alpha \langle e_5', M_2'\rangle. \tag{3.14}$$

So, by EVAL-CONTEXT, we have

$$\langle \mathsf{try}\ e_5\ \mathsf{catch}\ p\colon e_6, M_2\rangle \to \langle \mathsf{try}\ e_5'\ \mathsf{catch}\ p\colon e_6, M_2'\rangle\,.$$

From (3.14), we know $\phi'\langle \mathsf{try}\ e_3'\ \mathsf{catch}\ p\colon e_4, M_1'\rangle \approx_\alpha \langle \mathsf{try}\ e_5'\ \mathsf{catch}\ p\colon e_6, M_2'\rangle$, as desired.

In case $E[\,\cdot\,] = \mathsf{soft}\ [\,\cdot\,]$, where $e_2 = \mathsf{soft}\ e_4$, to apply the induction hypothesis, we need the additional fact that since e_3 is not a value, neither is e_4, and therefore $\vdash^{\alpha}_{[wf]} \langle \mathsf{soft}\ e_3, M_1\rangle$ and $\vdash_{[wf]} \langle \mathsf{soft}\ e_4, M_2\rangle$ imply $\vdash^{\alpha}_{[wf]} \langle e_3, M_1\rangle$ and $\vdash_{[wf]} \langle e_4, M_2\rangle$.

Case FAIL-PROP ($\langle F[\bot_p], M_1\rangle \to_\alpha \langle \bot_p, M_1\rangle$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = F'[\bot_p]$. By FAIL-PROP, we have $\langle F'[\bot_p], M_2\rangle \to \langle \bot_p, M_2\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(\bot_p) \approx_\alpha \bot_p$.

   This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Case GC ($\langle e_1, M_1\rangle \to_\alpha \langle e_1, M_1[G \mapsto \bot]\rangle$, where $\mathsf{gc}(G, \langle e_1, M_1\rangle)$):

By Lemma 27, let G' be such that $C \subseteq G' \subseteq \phi(G)$ and $\mathsf{gc}(G', \langle e_2, M_2\rangle)$, where

$$C = \{\phi(m^S) : m^S \in G \cap \mathsf{dom}(\phi) \wedge \vdash \alpha \preccurlyeq \mathsf{auth}^+(S) \sqcap \mathsf{persist}(S)\}$$

and

$$\phi(G) = \{\phi(m^S) : m^S \in G \cap \mathsf{dom}(\phi)\}.$$

Then, by GC, we have $\langle e_2, M_2\rangle \to \langle e_2, M_2[G' \mapsto \bot]\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(e_1) \approx_\alpha e_2$.

   This is given.

ii. We need to show $\phi(M_1[G \mapsto \bot]) \approx_\alpha M_2[G' \mapsto \bot]$.

   First, let $m_0^{S_0} \in \mathsf{dom}(\phi)$ be such that $M_1'(m_0^{S_0}) \neq \bot$. We show that $M_2'(\phi(m_0^{S_0})) \neq \bot$ and $\phi(M_1'(m_0^{S_0})) \approx_\alpha M_2'(\phi(m_0^{S_0}))$.

   Since $M_1'(m_0^{S_0}) \neq \bot$, we know $m_0^{S_0} \notin G$ and $\phi(m_0^{S_0}) \notin G'$. Therefore, $M_1'(m_0^{S_0}) = M_1(m_0^{S_0})$ and $M_2'(\phi(m_0^{S_0})) = M_2(\phi(m_0^{S_0}))$, so the result follows from the assumption $\phi(M_1) \approx_\alpha M_2$.

   Now, let $m_0^{S_0} \in \mathsf{dom}(\phi)$ be such that $\vdash \alpha \preccurlyeq \mathsf{auth}^+(S_0) \sqcap \mathsf{persist}(S_0)$ and $M_1'(m_0^{S_0}) = \bot$. We show that $M_2'(\phi(m_0^{S_0})) = \bot$. If $m_0^{S_0} \in G$, then $\phi(m_0^{S_0}) \in G'$, and the result follows by construction of M_2'. Otherwise, $m_0^{S_0} \notin G$, and so, $\phi(m_0^{S_0}) \notin G'$. Therefore, $M_1'(m_0^{S_0}) = M_1(m_0^{S_0})$ and $M_2'(\phi(m_0^{S_0})) = M_2(\phi(m_0^{S_0}))$, so the result follows from the assumption $\phi(M_1) \approx_\alpha M_2$.

Cases BRACKET-SELECT and BRACKET-SOFT-SELECT

($\langle [v].x_c, M_1\rangle \to_\alpha \langle [v.x_c], M_1\rangle$, where $v \in \{m_1^{S_1}, \mathsf{soft}\ m_1^{S_1}\}$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = [u].x_c$. From the grammar, we must have $u \in \{m_2^{S_2}, \mathsf{soft}\ m_2^{S_2}\}$. If $u = m_2^{S_2}$, then by BRACKET-SELECT,

$\left\langle [m_2^{S_2}].x_c, M_2 \right\rangle \to \left\langle [m_2^{S_2}.x_c], M_2 \right\rangle$. Otherwise, $u = \mathsf{soft}\ m_2^{S_2}$, and by BRACKET-SOFT-SELECT, $\left\langle [\mathsf{soft}\ m_2^{S_2}].x_c, M_2 \right\rangle \to \left\langle [(\mathsf{soft}\ m_2^{S_2}).x_c], M_2 \right\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi([v.x_c]) \approx_\alpha [u.x_c]$.

   This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Case BRACKET-ASSIGN ($\langle [v_1].x_c := v_2, M_1 \rangle \to_\alpha \langle [v_1.x_c := v_2], M_1 \rangle$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = [u_1].x_c := u_2$. By BRACKET-ASSIGN, we have $\langle [u_1].x_c := u_2, M_2 \rangle \to \langle [u_1.x_c := u_2], M_2 \rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi([v_1.x_c := v_2]) \approx_\alpha [u_1.x_c := u_2]$.

   This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Case BRACKET-SOFT ($\langle \mathsf{soft}\ [v], M_1 \rangle \to_\alpha \langle [\mathsf{soft}\ v], M_1 \rangle$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = \mathsf{soft}\ [e_3]$. Since $\langle e_2, M_2 \rangle$ is assumed to not diverge, $\langle e_3, M_2 \rangle$ cannot diverge either. So, by Corollary 14, there is a configuration $\langle u, M_2' \rangle$ such that $\langle e_3, M_2 \rangle \xrightarrow{\mathsf{e}}{}^* \langle u, M_2' \rangle$. Then by EVAL-CONTEXT, BRACKET-CONTEXT, and BRACKET-SOFT, we have

$$\langle \mathsf{soft}\ [e_3], M_2 \rangle \to^* \langle \mathsf{soft}\ [u], M_2' \rangle \to \langle [\mathsf{soft}\ u], M_2' \rangle .$$

Choose $\phi' = \phi$.

i. We need to show $\phi([\mathsf{soft}\ v]) \approx_\alpha [\mathsf{soft}\ u]$.

   This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2'$.

This follows by Lemma 31.

Case BRACKET-EXISTS

($\langle \mathsf{exists}\ [v]\ \mathsf{as}\ x : e_3\ \mathsf{else}\ e_4, M_1\rangle \to_\alpha \langle [\mathsf{exists}\ v\ \mathsf{as}\ x : e_3\ \mathsf{else}\ e_4], M_1\rangle$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = \mathsf{exists}\ [u]\ \mathsf{as}\ x : e_5\ \mathsf{else}\ e_6$. By BRACKET-EXISTS, we have

$$\langle \mathsf{exists}\ [u]\ \mathsf{as}\ x : e_5\ \mathsf{else}\ e_6, M_2\rangle \to \langle [\mathsf{exists}\ u\ \mathsf{as}\ x : e_5\ \mathsf{else}\ e_6], M_2\rangle$$

Choose $\phi' = \phi$.

i. We need to show $\phi([\mathsf{exists}\ v\ \mathsf{as}\ x : e_3\ \mathsf{else}\ e_4]) \approx_\alpha [\mathsf{exists}\ u\ \mathsf{as}\ x : e_5\ \mathsf{else}\ e_6]$.

This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

This is given.

Case BRACKET-APPLY ($\langle [v_1]\ v_2, M_1\rangle \to_\alpha \langle [v_1\ v_2], M_1\rangle$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = ([u_1]\ u_2)$. By BRACKET-APPLY, we have $\langle [u_1]\ u_2, M_2\rangle \to \langle [u_1\ u_2], M_2\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi([v_1\ v_2]) \approx_\alpha [u_1\ u_2]$.

This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

This is given.

Case BRACKET-TRY ($\langle \mathsf{try}\ [v]\ \mathsf{catch}\ p\colon e_3, M_1\rangle \to_\alpha \langle [\mathsf{try}\ v\ \mathsf{catch}\ p\colon e_3], M_1\rangle$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = (\mathsf{try}\ [e_4]\ \mathsf{catch}\ p\colon e_5)$. Since $\langle e_2, M_2\rangle$ is assumed to not diverge, $\langle e_4, M_2\rangle$ cannot diverge either. So, by Corollary 14, there is a configuration $\langle u, M_2'\rangle$ such that $\langle e_4, M_2\rangle \xrightarrow{\mathsf{e}}{}^* \langle u, M_2'\rangle$. Then by EVAL-CONTEXT, BRACKET-CONTEXT, and BRACKET-TRY, we have

$$\langle \mathsf{try}\ [e_4]\ \mathsf{catch}\ p\colon e_5, M_2\rangle \to^* \langle \mathsf{try}\ [u]\ \mathsf{catch}\ p\colon e_5, M_2'\rangle \to \langle [\mathsf{try}\ u\ \mathsf{catch}\ p\colon e_5], M_2'\rangle\,.$$

Choose $\phi' = \phi$.

i. We need to show $\phi([\mathsf{try}\ v\ \mathsf{catch}\ p\colon e_3]) \approx_\alpha [\mathsf{try}\ u\ \mathsf{catch}\ p\colon e_5]$.

   This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2'$.

   This follows by Lemma 31.

Case BRACKET-IF ($\langle \mathsf{if}\ [v]\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4, M_1\rangle \to_\alpha \langle [\mathsf{if}\ v\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4], M_1\rangle$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = (\mathsf{if}\ [u]\ \mathsf{then}\ e_5\ \mathsf{else}\ e_6)$. By BRACKET-IF, we have

$$\langle \mathsf{if}\ [u]\ \mathsf{then}\ e_5\ \mathsf{else}\ e_6, M_2\rangle \to \langle [\mathsf{if}\ u\ \mathsf{then}\ e_5\ \mathsf{else}\ e_6], M_2\rangle\,.$$

Choose $\phi' = \phi$.

i. We need to show $\phi([\mathsf{if}\ v\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4]) \approx_\alpha [\mathsf{if}\ u\ \mathsf{then}\ e_5\ \mathsf{else}\ e_6]$.

   This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Case BRACKET-LET ($\langle \mathsf{let}\ x = [v]\ \mathsf{in}\ e_3, M_1\rangle \to_\alpha \langle [e_3\{[v]/x\}], M_1\rangle$, where $\forall p.\ v \neq \bot_p$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = (\mathsf{let}\ x = [u]\ \mathsf{in}\ e_4)$. If $\forall p.\ u \neq \bot_p$, then by BRACKET-LET, we have

$$\langle \mathsf{let}\ x = [u]\ \mathsf{in}\ e_4, M_2\rangle \to \langle [e_4\{[u]/x\}], M_2\rangle\,.$$

Otherwise, let p be such that $u = \bot_p$. Then, by BRACKET-FAIL, we have

$$\langle \mathsf{let}\ x = [u]\ \mathsf{in}\ e_4, M_2\rangle \to \langle [\bot_p], M_2\rangle\,.$$

Choose $\phi' = \phi$.

i. We need to show $\phi([e_3\{[v]/x\}]) \approx_\alpha [e_2'']$, where $e_2'' \in \{e_4\{[u]/x\}, \bot_p\}$.

   This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Case DOUBLE-BRACKET ($\langle [[v]], M_1\rangle \to_\alpha \langle [v], M_1\rangle$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = [e_2'']$, so we trivially have $\langle [e_2''], M_2\rangle \to^* \langle [e_2''], M_2\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi([v]) \approx_\alpha [e_2'']$.

   This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

   This is given.

Case BRACKET-CONTEXT ($\langle [e_3], M_1\rangle \to_\alpha \langle [e_3'], M_1'\rangle$, where $\langle e_3, M_1\rangle \xrightarrow{\mathsf{e}} \langle e_3', M_1'\rangle$):

From $\phi(e_1) \approx_\alpha e_2$, we know that $e_2 = [e_4]$, and we trivially have $\langle [e_4], M_2\rangle \xrightarrow{\mathsf{e}}{}^* \langle [e_4], M_2\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi([e_3']) \approx_\alpha [e_4]$.

This is trivial.

ii. We need to show $\phi(M_1') \approx_\alpha M_2$.

This follows by Lemma 31.

Case BRACKET-FAIL ($\langle F[[\bot_p]], M_1\rangle \to_\alpha \langle [\bot_p], M_1\rangle$):

From $\phi(e_1) \approx_\alpha e_2$, an easy case analysis on the syntax of $F[\,\cdot\,]$ shows that $\langle e_2, M_2\rangle \to \langle [\bot_p], M_2\rangle$ via BRACKET-FAIL. Choose $\phi' = \phi$.

i. We need to show $\phi([\bot_p]) \approx_\alpha [\bot_p]$.

This is trivial.

ii. We need to show $\phi(M_1) \approx_\alpha M_2$.

This is given.

Case α-CREATE ($\langle e_1, M_1\rangle \to_\alpha \left\langle e_1, M_1[m^S \mapsto \{\overrightarrow{x_i = [v_i]}\}]\right\rangle$, where m^S is fresh, $\varnothing; \top; \top \vdash \{\overrightarrow{x_i = [v_i]}\}^S : R_\top, \top$, $\vdash^\alpha_{[wf]} M[m^S \mapsto \{\overrightarrow{x_i = [v_i]}\}]$, and $\alpha \notin \mathsf{persist}(S)$) :

We trivially have $\langle e_2, M_2\rangle \to^* \langle e_2, M_2\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(e_1) \approx_\alpha e_2$.

This is given.

ii. We need to show $\phi(M_1[m^S \mapsto \{\overrightarrow{x_i = [v_i]}\}]) \approx_\alpha M_2$.

Since $m^S \notin \mathsf{dom}(\phi)$, this follows from the assumption $\phi(M_1) \approx_\alpha M_2$ by construction of M_1'.

Case α-ASSIGN ($\langle e_1, M_1\rangle \to_\alpha \langle e_1, M_1[m^S.x_c \mapsto [v]]\rangle$, where $m^S \in \mathsf{dom}(M_1)$, $M_1(m^S) \neq \bot$, $S = \{\overrightarrow{x_i : \tau_i}\}_s$, $\varnothing; \top; \top \vdash [v] : \tau_c, \top$, and $\vdash^\alpha_{[wf]} M_1[m^S.x_c \mapsto [v]]$):

We trivially have $\langle e_2, M_2\rangle \to^* \langle e_2, M_2\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(e_1) \approx_\alpha e_2$.

This is given.

ii. We need to show $\phi(M_1[m^S.x_c \mapsto [v]]) \approx_\alpha M_2$.

If $m^S \notin \mathsf{dom}(\phi)$, then this follows from the assumption $\phi(M_1) \approx_\alpha M_2$ by construction of M_1'.

Suppose $m^S \in \mathsf{dom}(\phi)$. Then it suffices to show that $\phi(M_1'(m^S)) \approx_\alpha M_2(\phi(m^S))$, since the rest follows from $\phi(M_1) \approx_\alpha M_2$.

Let $\overrightarrow{v_i}$, $\overrightarrow{v_i'}$, and $\overrightarrow{u_i}$ be such that $M_1(m^S) = \{\overrightarrow{x_i = v_i}\}$ $M_1'(m^S) = \{\overrightarrow{x_i = v_i'}\}$ and $M_2(m^S) = \{\overrightarrow{x_i = u_i}\}$. From $\phi(M_1) \approx_\alpha M_2$, we know $\phi(v_i) \approx_\alpha u_i$ for all i. By construction of M_1', we have $v_c' = [v]$ and $v_i' = v_i$ for $i \neq c$. Therefore, it remains to be shown that $\phi([v]) \approx_\alpha u_c$.

From $\varnothing; \top; \top \vdash [v] : \tau_c, \top$, we know that $\alpha \notin \mathsf{integ}(\tau_c)$. Therefore, from $\vdash^\alpha_{[wf]} M_2$, we must have $u_c = [u]$, for some u. The result $\phi([v]) \approx_\alpha u_c$ then follows trivially.

Case α-FORGET ($\langle e_1, M_1\rangle \to_\alpha \langle e_1, M_1[m^S \mapsto \bot]\rangle$, where $m^S \in \mathsf{dom}(M_1)$ and $\alpha \notin \mathsf{persist}(S)$):

We trivially have $\langle e_2, M_2\rangle \to^* \langle e_2, M_2\rangle$. Choose $\phi' = \phi$.

i. We need to show $\phi(e_1) \approx_\alpha e_2$.

This is given.

ii. We need to show $\phi(M_1[m^S \mapsto \bot]) \approx_\alpha M_2$.

This follows from the assumption $\phi(M_1) \approx_\alpha M_2$, since $\alpha \notin \mathsf{persist}(S)$.

□

3.8 Related work

This chapter identifies and addresses a new problem, referential security. As a result, little prior work is closely related.

Some prior work has tried to improve referential integrity through system mechanisms, for example improving the referential integrity of Web hyperlinks [21, 39]. Systems mechanisms for improving referential integrity (and other aspects of trustworthiness) are orthogonal to the language model presented here, but could be used to justify assigning persistence, integrity, and authority levels to nodes.

Liblit and Aiken [44] develop a type system for distributed data structures. Its explicit two-level hierarchy distinguishes between local pointers meaningful only to a single processor, and global pointers that are valid everywhere. The type system ensures that local pointers do not leak into a global context. This work was extended in [45] to add types for dealing with private vs. shared data. However, this line of work does not consider security properties that require defence against an adversary.

Riely and Hennessey study type safety in a distributed system of partially trusted mobile agents [67] but do not consider referential security.

Our approach builds on prior work on information-flow security, much of which is summarized by [70]. The Fabric platform described in Chapter 2 has a high-level language that, like $\lambda_{persist}$, includes integrity annotations and abstracts away the locations of objects. Fabric does not enforce referential security, however, so adding the features described here is an obvious next step.

Chugh et al. [15] develop an approach to dynamically loading untrusted, mobile JavaScript code and ensuring that the code satisfies necessary security properties. However, referential security properties are not covered.

3.A Appendix

3.A.1 Full syntax of $\lambda_{persist}$

Variables	$x, y \in \mathsf{Var}$	Policy levels	$w, a, p, \ell \in \mathcal{L}$
Memory locations	$m \in \mathsf{Mem}$	PC labels	$pc ::= w$
		Storage labels	$s ::= (a, p)$
Labelled record types	$S ::= \{\overrightarrow{x_i : \tau_i}\}_s$	Reference labels	$r ::= (a^+, a^-, p)$
Labelled reference types	$R ::= \{\overrightarrow{x_i : \tau_i}\}_r$	Persistence failure handlers	$\mathcal{H} ::= \overrightarrow{p_i}$
Base types	$b ::= \mathsf{bool} \mid \tau_1 \xrightarrow{pc,\mathcal{H}} \tau_2 \mid R \mid \mathsf{soft}\ R$	Types	$\tau ::= b_w \mid \mathbf{1}$
Values	$v, u ::= x \mid \mathsf{true} \mid \mathsf{false} \mid * \mid m^S \mid \mathsf{soft}\ m^S \mid \lambda(x : \tau)[pc; \mathcal{H}].\, e\ \ (\mid \bot_p)$		
Terms	$e ::= v \mid v_1\ v_2 \mid \mathsf{if}\ v_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \{\overrightarrow{x_i = v_i}\}^S \mid v.x \mid v_1.x := v_2$ $\mid \mathsf{soft}\ e \mid e_1 \Vdash e_2 \mid \mathsf{exists}\ v\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ $\mid \mathsf{try}\ e_1\ \mathsf{catch}\ p\text{:}\ e_2$		

3.A.2 Full small-step operational semantics for ordinary (non-adversarial) execution of $\lambda_{persist}$

$$[\textsc{Apply}] \quad \langle (\lambda(x{:}\tau)[pc;\mathcal{H}].\, e)\ v, M\rangle \xrightarrow{\mathsf{e}} \langle e\{v/x\}, M\rangle$$

$$[\textsc{Let}] \quad \frac{\forall p.\ v \neq \bot_p}{\langle \mathsf{let}\ x = v\ \mathsf{in}\ e, M\rangle \xrightarrow{\mathsf{e}} \langle e\{v/x\}, M\rangle}$$

$$[\textsc{If-True}] \quad \langle \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2, M\rangle \xrightarrow{\mathsf{e}} \langle e_1, M\rangle$$

$$[\textsc{If-False}] \quad \langle \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2, M\rangle \xrightarrow{\mathsf{e}} \langle e_2, M\rangle$$

$$[\textsc{Create}] \quad \frac{m = \mathsf{newloc}(M)}{\left\langle \{\overrightarrow{x_i = v_i}\}^S, M\right\rangle \xrightarrow{\mathsf{e}} \left\langle m^S, M[m^S \mapsto \{\overrightarrow{x_i = v_i}\}]\right\rangle}$$

$$\left[\begin{array}{c}\textsc{Parallel-}\\ \textsc{Result}\end{array}\right] \quad \langle v_1 \| v_2, M\rangle \xrightarrow{\mathsf{e}} \langle *, M\rangle \qquad [\textsc{Select}] \quad \frac{M(m^S) = \{\overrightarrow{x_i = v_i}\}}{\left\langle m^S.x_c, M\right\rangle \xrightarrow{\mathsf{e}} \langle v_c, M\rangle}$$

$$[\textsc{Assign}] \quad \frac{M(m^S) \neq \bot \qquad \forall p.\ v \neq \bot_p}{\left\langle m^S.x_c := v, M\right\rangle \xrightarrow{\mathsf{e}} \left\langle *, M[m^S.x_c \mapsto v]\right\rangle}$$

$$\left[\begin{array}{c}\textsc{Dangle-}\\ \textsc{Select}\end{array}\right] \quad \frac{M(m^S) = \bot \qquad p = \mathsf{persist}(m^S)}{\left\langle m^S.x_c, M\right\rangle \xrightarrow{\mathsf{e}} \langle \bot_p, M\rangle} \qquad \left[\begin{array}{c}\textsc{Dangle-}\\ \textsc{Assign}\end{array}\right] \quad \frac{M(m^S) = \bot \qquad p = \mathsf{persist}(m^S)}{\left\langle m^S.x_c := v, M\right\rangle \xrightarrow{\mathsf{e}} \langle \bot_p, M\rangle}$$

$$\left[\begin{array}{c}\textsc{Exists-}\\ \textsc{True}\end{array}\right] \quad \frac{M(m^S) \neq \bot}{\left\langle \mathsf{exists\ soft}\ m^S\ \mathsf{as}\ x :\ e_1\ \mathsf{else}\ e_2, M\right\rangle \xrightarrow{\mathsf{e}} \left\langle e_1\{m^S/x\}, M\right\rangle}$$

$$\left[\begin{array}{c}\textsc{Exists-}\\ \textsc{False}\end{array}\right] \quad \frac{M(m^S) = \bot}{\left\langle \mathsf{exists\ soft}\ m^S\ \mathsf{as}\ x :\ e_1\ \mathsf{else}\ e_2, M\right\rangle \xrightarrow{\mathsf{e}} \langle e_2, M\rangle}$$

$$\left[\begin{array}{c}\textsc{Eval-}\\ \textsc{Context}\end{array}\right] \quad \frac{\langle e, M\rangle \xrightarrow{\mathsf{e}} \langle e', M'\rangle}{\langle E[e], M\rangle \xrightarrow{\mathsf{e}} \langle E[e'], M'\rangle} \qquad \left[\begin{array}{c}\textsc{Fail-}\\ \textsc{Prop}\end{array}\right] \quad \langle F\,[\bot_p]\,, M\rangle \xrightarrow{\mathsf{e}} \langle \bot_p, M\rangle$$

$$E ::= \mathsf{soft}\ [\,\cdot\,] \mid \mathsf{let}\ x = [\,\cdot\,]\ \mathsf{in}\ e \mid [\,\cdot\,] \| e \mid e \| [\,\cdot\,] \mid \mathsf{try}\ [\,\cdot\,]\ \mathsf{catch}\ p{:}\ e$$
$$F ::= \mathsf{soft}\ [\,\cdot\,] \mid \mathsf{let}\ x = [\,\cdot\,]\ \mathsf{in}\ e$$

$$\left[\begin{array}{c}\textsc{Soft-}\\ \textsc{Select}\end{array}\right] \quad \frac{\left\langle m^S.x_c, M\right\rangle \xrightarrow{\mathsf{e}} \langle v, M\rangle}{\left\langle (\mathsf{soft}\ m^S).x_c, M\right\rangle \xrightarrow{\mathsf{e}} \langle v, M\rangle} \qquad \left[\begin{array}{c}\textsc{Soft-}\\ \textsc{Assign}\end{array}\right] \quad \frac{\left\langle m^S.x_c := v, M\right\rangle \xrightarrow{\mathsf{e}} \langle v', M'\rangle}{\left\langle (\mathsf{soft}\ m^S).x_c := v, M\right\rangle \xrightarrow{\mathsf{e}} \langle v', M'\rangle}$$

$$[\textsc{Try-Val}] \quad \frac{\forall p'.\ v \neq \bot_{p'}}{\langle \mathsf{try}\ v\ \mathsf{catch}\ p{:}\ e, M\rangle \xrightarrow{\mathsf{e}} \langle v, M\rangle} \qquad \left[\begin{array}{c}\textsc{Try-}\\ \textsc{Catch}\end{array}\right] \quad \frac{p \preccurlyeq p'}{\langle \mathsf{try}\ \bot_{p'}\ \mathsf{catch}\ p{:}\ e, M\rangle \xrightarrow{\mathsf{e}} \langle e, M\rangle}$$

$$[\textsc{Try-Esc}] \quad \frac{p \not\preccurlyeq p'}{\langle \mathsf{try}\ \bot_{p'}\ \mathsf{catch}\ p{:}\ e, M\rangle \xrightarrow{\mathsf{e}} \langle \bot_{p'}, M\rangle}$$

$$[\textsc{Prog-Step}] \quad \frac{\langle e, M\rangle \xrightarrow{\mathsf{e}} \langle e', M'\rangle}{\langle e, M\rangle \to \langle e', M'\rangle} \qquad [\mathrm{GC}] \quad \frac{\mathsf{gc}(G, \langle e, M\rangle)}{\langle e, M\rangle \to \langle e, M[G \mapsto \bot]\rangle}$$

3.A.3 Full subtyping rules for $\lambda_{persist}$

$$[\text{S1}]\ \frac{n > m}{\vdash \{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}_r \leq \{x_1{:}\tau_1, \ldots, x_m{:}\tau_m\}_r} \qquad [\text{S2}]\ \frac{\vdash R_1 \leq R_2}{\vdash \mathsf{soft}\ R_1 \leq \mathsf{soft}\ R_2}$$

$$[\text{S3}]\ \frac{\vdash b_1 \leq b_2}{\vdash (b_1)_{w_1} \leq (b_2)_{w_2}} \qquad [\text{S4}]\ \frac{\vdash w_2 \preccurlyeq w_1 \qquad \vdash \tau_2 \leq \tau_1 \qquad \vdash \tau_1' \leq \tau_2' \qquad \vdash pc_1 \preccurlyeq pc_2 \qquad \vdash \mathcal{H}_2 \preccurlyeq \mathcal{H}_1}{\vdash \tau_1 \xrightarrow{pc_1,,\mathcal{H}_1} \tau_1' \leq \tau_2 \xrightarrow{pc_2,,\mathcal{H}_2} \tau_2'}$$

$$[\text{S5}]\ \frac{\vdash a_1^+ \preccurlyeq a_2^+ \qquad \vdash a_2^- \preccurlyeq a_1^- \qquad \vdash p_2 \preccurlyeq p_1}{\vdash \{\overrightarrow{x_i : \tau_i}\}_{(a_1^+, a_1^-, p_1)} \leq \{\overrightarrow{x_i : \tau_i}\}_{(a_2^+, a_2^-, p_2)}}$$

3.A.4 Full typing rules for $\lambda_{persist}$

$$[\text{T-Bool}]\ \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma; pc; \mathcal{H} \vdash b : \mathsf{bool}_\top, \top} \quad [\text{T-Unit}]\ \Gamma; pc; \mathcal{H} \vdash * : \mathbf{1}, \top \quad [\text{T-Var}]\ \frac{\Gamma(x) = \tau}{\Gamma; pc; \mathcal{H} \vdash x : \tau, \top}$$

$$[\text{T-Bottom}]\ \frac{p \neq \top \qquad \vdash \mathcal{H} \preccurlyeq p}{\Gamma; pc; \mathcal{H} \vdash \bot_p : \tau, p} \quad [\text{T-Loc}]\ \frac{\vdash_{wf} S : \mathsf{rectype} \qquad S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}}{\Gamma; pc; \mathcal{H} \vdash m^S : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top}$$

$$[\text{T-Parallel}]\ \frac{\Gamma; pc; \top \vdash e_i : \tau_i, \top\ ^{(\forall i)}}{\Gamma; pc; \mathcal{H} \vdash e_1 \parallel e_2 : \mathbf{1}, \top} \quad [\text{T-Soft}]\ \frac{\Gamma; pc; \mathcal{H} \vdash e : R_w, \mathcal{X}}{\Gamma; pc; \mathcal{H} \vdash \mathsf{soft}\ e : (\mathsf{soft}\ R)_w, \mathcal{X}}$$

$$[\text{T-If}]\ \frac{\begin{array}{c}\Gamma; pc; \mathcal{H} \vdash v : \mathsf{bool}_w, \top \\ \Gamma; pc \sqcap w; \mathcal{H} \vdash e_i : \tau, \mathcal{X}_i\ ^{(\forall i)} \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc \sqcap w\end{array}}{\Gamma; pc; \mathcal{H} \vdash \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau \sqcap w, \mathcal{X}_1 \sqcap \mathcal{X}_2}$$

$$[\text{T-Abs}]\ \frac{\Gamma, x{:}\tau'; pc'; \mathcal{H}' \vdash e : \tau, \mathcal{H}' \qquad \vdash_{wf} (\tau' \xrightarrow{pc', \mathcal{H}'} \tau)_\top : \mathsf{type} \qquad \vdash pc' \preccurlyeq pc}{\Gamma; pc; \mathcal{H} \vdash \lambda(x{:}\tau')[pc'; \mathcal{H}'].\, e : (\tau' \xrightarrow{pc', \mathcal{H}'} \tau)_\top, \top}$$

$$[\text{T-App}]\ \frac{\begin{array}{c}\Gamma; pc; \mathcal{H} \vdash v_1 : (\tau' \xrightarrow{pc', \mathcal{H}'} \tau)_w, \top \qquad \Gamma; pc; \mathcal{H} \vdash v_2 : \tau', \top \\ \vdash pc' \preccurlyeq pc \sqcap w \qquad \vdash \mathcal{H} \preccurlyeq \mathcal{H}'\end{array}}{\Gamma; pc; \mathcal{H} \vdash v_1\ v_2 : \tau \sqcap w, \mathcal{H}'}$$

$$[\text{T-Record}]\ \frac{\begin{array}{c}\vdash_{wf} S : \mathsf{rectype} \qquad S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)} \qquad \Gamma; pc; \mathcal{H} \vdash v_i : \tau_i', \top\ ^{(\forall i)} \\ \vdash \tau_i' \leq \tau_i\ ^{(\forall i)} \qquad \vdash \mathsf{auth}^+(\tau_i') \preccurlyeq pc\ ^{(\forall i)} \qquad \vdash \mathsf{integ}(\tau_i) \preccurlyeq pc\ ^{(\forall i)} \qquad \vdash p \preccurlyeq pc\end{array}}{\Gamma; pc; \mathcal{H} \vdash \{\overrightarrow{x_i = v_i}\}^S : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top, \top}$$

$$[\text{T-Select}]\ \frac{\Gamma; pc; \mathcal{H} \vdash v : (\{\overrightarrow{x_i : \tau_i}\}_{(a^+,a^-,p)})_w, \top \qquad \vdash a^+ \preccurlyeq pc \qquad w' = w \sqcap p \qquad \vdash \mathcal{H} \preccurlyeq p}{\Gamma; pc; \mathcal{H} \vdash v.x_c : \tau_c \sqcap w', p}$$

$$[\text{T-Assign}]\ \frac{\begin{array}{c}\Gamma; pc; \mathcal{H} \vdash v_1 : (\{\overrightarrow{x_i : \tau_i}\}_{(a^+,a^-,p)})_w, \top \qquad \vdash a^+ \preccurlyeq pc \qquad \Gamma; pc; \mathcal{H} \vdash v_2 : \tau, \top \\ \vdash \tau \sqcap pc \sqcap w \leq \tau_c \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc \sqcap w \qquad \vdash \mathcal{H} \preccurlyeq p\end{array}}{\Gamma; pc; \mathcal{H} \vdash v_1.x_c := v_2 : \mathbf{1}, p}$$

$$\left[\begin{array}{l}\text{T-Soft-}\\ \text{Select}\end{array}\right]\ \frac{\begin{array}{c}\Gamma; pc; \mathcal{H} \vdash v : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_{(a^+,a^-,p)})_w, \top \qquad \vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc \\ w' = w \sqcap a^- \sqcap p \qquad \vdash \mathcal{H} \preccurlyeq p\end{array}}{\Gamma; pc; \mathcal{H} \vdash v.x_c : \tau_c \sqcap w', p}$$

$$\left[\begin{array}{l}\text{T-Soft-}\\ \text{Assign}\end{array}\right]\ \frac{\begin{array}{c}\Gamma; pc; \mathcal{H} \vdash v_1 : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_{(a^+,a^-,p)})_w, \top \qquad \Gamma; pc; \mathcal{H} \vdash v_2 : \tau, \top \\ \vdash \tau \sqcap pc \sqcap w \leq \tau_c \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc \sqcap w \qquad \vdash \mathcal{H} \preccurlyeq p\end{array}}{\Gamma; pc; \mathcal{H} \vdash v_1.x_c := v_2 : \mathbf{1}, p}$$

$$[\text{T-Exists}]\ \frac{\begin{array}{c}\Gamma; pc; \mathcal{H} \vdash v : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_r)_w, \top \qquad \vdash \mathsf{auth}^+(r) \preccurlyeq pc \sqcap w \\ w' = \mathsf{auth}^-(r) \sqcap \mathsf{persist}(r) \sqcap w \qquad \Gamma, x{:}(\{\overrightarrow{x_i : \tau_i}\}_r)_w; pc \sqcap w'; \mathcal{H} \vdash e_1 : \tau, \mathcal{X}_1 \\ \Gamma; pc \sqcap w'; \mathcal{H} \vdash e_2 : \tau, \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc \sqcap w'\end{array}}{\Gamma; pc; \mathcal{H} \vdash \mathsf{exists}\ v\ \mathsf{as}\ x :\ e_1\ \mathsf{else}\ e_2 : \tau \sqcap w', \mathcal{X}_1 \sqcap \mathcal{X}_2}$$

$$[\text{T-Try}]\ \frac{\begin{array}{c}\Gamma; pc; \mathcal{H}, p \vdash e_1 : \tau, \mathcal{X}_1 \qquad w = \bigsqcap_{p' \in \mathcal{X}_1} (p \sqcup p') \\ \Gamma; pc \sqcap w \sqcap \mathsf{integ}(\tau); \mathcal{H} \vdash e_2 : \tau, \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc\end{array}}{\Gamma; pc; \mathcal{H} \vdash \mathsf{try}\ e_1\ \mathsf{catch}\ p{:}\ e_2 : \tau \sqcap w, (\mathcal{X}_1/p) \sqcap \mathcal{X}_2}$$

$$[\text{T-Let}]\ \frac{\begin{array}{c}\Gamma; pc; \mathcal{H} \vdash e_1 : \tau', \mathcal{X}_1 \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \qquad w = (\bigsqcap \mathcal{X}_1) \sqcap \mathsf{integ}(\tau') \\ pc' = pc \sqcap w \qquad \Gamma, x{:}\tau'; pc'; \mathcal{H} \vdash e_2 : \tau, \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc'\end{array}}{\Gamma; pc; \mathcal{H} \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau \sqcap w, \mathcal{X}_1 \sqcap \mathcal{X}_2}$$

$$[\text{T-Subsume}]\ \frac{\Gamma; pc; \mathcal{H} \vdash e : \tau', \mathcal{X}' \qquad \vdash \tau' \leq \tau \qquad \vdash \mathcal{H} \preccurlyeq \mathcal{X} \qquad \vdash \mathcal{X} \preccurlyeq \mathcal{X}'}{\Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}}$$

CHAPTER 4

CONCLUSION

Federated systems provide new services and capabilities by integrating distributed information systems across independent administrative domains. We are entering an era in which federated systems are widely used to share information and computation. This dissertation examines the challenge of designing and building federated information systems that are secure and reliable. In doing so, it makes two key contributions.

The first contribution is the design and implementation of Fabric, a platform for building federated information systems with confidentiality and integrity assurances. Fabric aims to make secure distributed applications easier to develop, and to enable the secure integration of information systems controlled by different organizations.

The second contribution is the identification of referential security vulnerabilities, which arise in federated systems with persistent information. We formally characterize these vulnerabilities, and introduce a high-level language for modelling, analyzing, and preventing them.

4.1 Securely sharing computation and storage

Chapter 2 presented Fabric, a new distributed platform for general secure sharing of information and computation resources. Fabric provides a high-level abstraction for secure, consistent, distributed general-purpose computations using distributed, persistent information. Persistent information is conveniently presented as language-level objects connected by pointers. Fabric exposes security assumptions and policies explicitly and declaratively. It flexibly supports both data-shipping and function-shipping styles of computation. Results from im-

plementing complex, realistic systems in Fabric, such as CMS and SIF, suggest it has the expressive power and performance to be useful in practice.

Fabric led to some technical contributions. Fabric extends the Jif programming language with new features for distributed programming, while showing how to integrate those features with secure information flow. This integration requires a new trust ordering on information-flow labels, and new implementation mechanisms such as writer maps and hierarchical two-phase commit.

4.2 Defining and enforcing referential security

While Fabric perhaps goes farther toward the goal of securely and transparently sharing distributed resources than prior systems, it does not guarantee availability in the way that it does confidentiality and integrity. Chapter 3 identified formalized a class of referential security properties that is important for availability of distributed systems with persistence, and showed that referential security requirements can be expressed through label annotations. It introduced $\lambda_{persist}$, a high-level language for modelling referential security issues in a distributed system, and it demonstrated how to enforce these security properties, through static analysis expressed as a type system for the language. The type system was validated by formal proofs that $\lambda_{persist}$ programs enforce the new security properties.

4.3 Future work

There are many hard problems left to solve to make the task of building secure, reliable federated systems easy enough to be achievable by programmers who are not security experts.

The obvious next step is to integrate the features of $\lambda_{persist}$ with Fabric's language. This would help evaluate how well its types guide programmers in designing distributed computing systems.

With advanced type systems, such as those in Fabric and $\lambda_{persist}$, programmer annotation burden is a common concern. More techniques are needed for reducing this burden while maintaining safety guarantees. For example, Fabric applications have `@s` and `@w` annotations for specifying the stores on which to place objects, and the workers on which to make remote calls. Inferring these annotations by automatically partitioning programs and data, while maintaining security and optimizing performance, is an interesting problem and would reduce annotation burden.

Persistent objects introduce the problem of schema evolution and its security implications. We have recently developed support in Fabric for basic class evolution [3]. Supporting full schema evolution securely and intuitively remains an open and difficult problem.

The performance of Fabric is limited by its strong consistency guarantees. Much of the computational overhead at the worker is due to transaction logging. Reducing this overhead, perhaps through a principled weakening of consistency guarantees, or by leveraging recent work on transactional memory, would be valuable.

Fabric's hierarchical commit protocol depends on the availability of the transaction coordinator. As discussed in Section 2.4.3, Fabric weakens its safety guarantees for stronger availability guarantees by timing out prepared transactions. Adding consensus mechanisms to the commit protocol appears to be a promising approach to recovering safety.

Future information systems must be secure and reliable while supporting

mutually distrusting participants. An important goal is to put the construction of these systems within the reach of the everyday programmer. Higher-level programming models are a key aspect of attaining this goal, and this has been a guiding principle of the work presented here. I hope that the contributions of this dissertation will in some way help achieve this goal.

BIBLIOGRAPHY

[1] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *Proc. 21st ACM Symposium on Operating System Principles (SOSP)*, pages 159–174, Stevenson, WA, USA, October 2007.

[2] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: Scaling file servers via cooperative caching. In *Proc. 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 129–142, Boston, MA, USA, May 2005.

[3] Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. Sharing mobile code securely with information flow control. In *Proc. IEEE 2012 Symposium on Security and Privacy*, San Francisco, CA, USA, May 2012.

[4] Malcom Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In *Proc. International Conference on Deductive Object Oriented Databases*, pages 223–240, Kyoto, Japan, December 1989.

[5] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proc. 14th ACM Symposium on Operating System Principles (SOSP)*, pages 217–230, Asheville, NC, USA, December 1993.

[6] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proc. 1st ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 78–86, Portland, OR, USA, September 1986.

[7] Chavdar Botev, Hubert Chao, Theodore Chao, Yim Cheng, Raymond Doyle, Sergey Grankin, Jon Guarino, Saikat Guha, Pei-Chen Lee, Dan Perry, Christopher Re, Ilya Rifkin, Tingyan Yuan, Dora Abdullah, Kathy Carpenter, David Gries, Dexter Kozen, Andrew Myers, David Schwartz, and Jayavel Shanmugasundaram. Supporting workflow in a course management system. In *Proc. 36th ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 262–266, St. Louis, MO, USA, February 2005.

[8] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[9] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proc. ACM SIGMOD 1993 International Conference on Management of Data*, pages 12–21, Washington, DC, USA, May 1993.

[10] Miguel Castro, Atul Adya, Barbara Liskov, and Andrew C. Myers. HAC: Hybrid adaptive caching for distributed storage systems. In *Proc. 16th ACM Symposium on Operating System Principles (SOSP)*, pages 102–115, St. Malo, France, October 1997.

[11] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[12] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proc. 21st ACM Symposium on Operating System Principles (SOSP)*, pages 31–44, Stevenson, WA, USA, October 2007.

[13] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *Proc. 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 242–253, Venice, Italy, July 2006.

[14] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proc. 16th USENIX Security Symposium*, pages 1–16, Boston, MA, USA, August 2007.

[15] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *Proc. ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI)*, pages 50–62, Dublin, Ireland, June 2009.

[16] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *Proc. IEEE 2008 Symposium on Security and Privacy*, pages 354–368, Oakland, CA, USA, May 2008.

[17] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems (TODS)*, 4(4):397–434, December 1979.

[18] Common Criteria for Information Technology Security Evaluation: Part 1: Introduction and general model, July 2009. CCMB-2009-07-001, Version 3.1, Revision 3. Available from `http://www.commoncriteriaportal.org/`.

[19] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proc. 5th International Symposium on Formal Methods for Components and Objects*, pages 266–296, Amsterdam, The Netherlands, November 2006. Available from `http://groups.inf.ed.ac.uk/links/`.

[20] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating System Principles (SOSP)*, pages 202–215, Banff, AB, Canada, October 2001.

[21] Hugh C. Davis. Referential integrity of links in open hypermedia systems. In *Proc. 9th ACM Conference on Hypertext and Hypermedia*, pages 207–216, Pittsburgh, PA, USA, June 1998.

[22] Mark Day, Barbara Liskov, Umesh Maheshwari, and Andrew C. Myers. References to remote mobile objects in Thor. *ACM Letters on Programming Languages and Systems*, 2(1–4):115–126, March–December 1993.

[23] Linda G. DeMichiel, L. Ümit Yalçinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification, Version 2.0*. Sun Microsystems, Palo Alto, CA, USA, August 2001.

[24] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, MA, USA, June 1982. ISBN 978-0201101508.

[25] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[26] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[27] Cédric Fournet, Guervan le Guernic, and Tamara Rezk. A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms. In *Proc. 17th ACM Conference on Computer and Communications Security (CCS)*, pages 432–441, Chicago, IL, USA, November 2009.

[28] Robert J. Fowler. Decentralized object finding using forwarding addresses. Technical Report 85-12-1, University of Washington, Seattle, WA, USA, December 1985. Ph.D. thesis.

[29] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proc. IEEE 1982 Symposium on Security and Privacy*, pages 11–20, Oakland, CA, USA, April 1982.

[30] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, Reading, MA, USA, June 2000. ISBN 978-0201310085.

[31] Jim Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag. ISBN 3-540-08755-9.

[32] Todd M. Greanier. Flatten your objects: Discover the secrets of the Java Serialization API. *JavaWorld*, July 2000. Available from `http://www.javaworld.com/javaworld/jw-07-2000/jw-0714-flatten.html`.

[33] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Yves Lafon, Jean-Jacques Moreau, Anish Karmarkar, and Henrik Frystyk Nielsen. SOAP version 1.2 part 1: Messaging framework (second edition). W3C recommendation, W3C, Cambridge, MA, USA, April 2007. Available from `http://www.w3.org/TR/2007/REC-soap12-part1-20070427/`.

[34] Health Insurance Portability and Privacy Act of 1996. US Public Law No. 104–191, 110 Stat. 1936.

[35] Maurice P. Herlihy and Jeannette M. Wing. Avalon: Language support for reliable distributed systems. In *Proc. 17th International Symposium on Fault-Tolerant Computing*, pages 89–94, Pittsburgh, PA, USA, July 1987.

[36] Boniface Hicks, Kiyan Ahmadizadeh, and Patrick McDaniel. Understanding practical application development in security-typed languages. In *22nd Annual Computer Security Applications Conference (ACSAC)*, pages 153–164, Miami Beach, FL, USA, December 2006.

[37] Russell Housley, Tim Polk, Warwick Ford, and David Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Internet RFC-3280, April 2002.

[38] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. 6(1):109–133, February 1988.

[39] Frank Kappe. A scalable architecture for maintaining referential integrity

in distributed information systems. *Journal of Universal Computer Science*, 1(2):84–104, February 1995.

[40] Linda T. Kohn, Janet M. Corrigan, and Molla S. Donaldson, editors. *To Err is Human: Building a Safer Health System*. The National Academies Press, Washington, DC, USA, April 2000. ISBN 978-0309068376.

[41] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proc. 21st ACM Symposium on Operating System Principles (SOSP)*, pages 321–334, Stevenson, WA, USA, October 2007.

[42] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–201, Cambridge, MA, USA, November 2000.

[43] Charles Lamb, Gordon Landis, Jack A. Orenstein, and Daniel Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[44] Ben Liblit and Alexander Aiken. Type systems for distributed data structures. In *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL)*, pages 199–213, Boston, MA, January 2000.

[45] Ben Liblit, Alexander Aiken, and Katherine A. Yelick. Type systems for distributed data sharing. In *Proc. 10th International Static Analysis Symposium*, volume 2694 of *LNCS*, pages 273–294, San Diego, CA, USA, June 2003. Springer-Verlag.

[46] Barbara Liskov, Atul Adya, Miguel Castro, Mark Day, Sanjay Ghemawat, Robjert Gruber, Umesh Maheshwari, Andrew C. Myers, and Liuba Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proc. ACM SIGMOD 1996 International Conference on Management of Data*, pages 318–329, Montréal, QC, Canada, June 1996.

[47] Barbara H. Liskov. The Argus language and system. In *Distributed Systems: Methods and Tools for Specification*, volume 150 of *Lecture Notes in Computer Science*, pages 343–430. Springer-Verlag Berlin, 1985.

[48] 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 *Proc. 22nd ACM Symposium on Operating System Principles (SOSP)*, pages 321–334, Big Sky, MT, USA, October 2009.

[49] John MacCormick, Nick Murph, Marc Najor, Chandramohan A. Thekkat, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 105–120, San Francisco, CA, USA, December 2004.

[50] Francis McCabe, David Booth, Christopher Ferris, David Orchard, Mike Champion, Eric Newcomer, and Hugo Haas. Web services architecture. W3C note, W3C, Cambridge, MA, USA, February 2004. Available from `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/`.

[51] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, February 1990. ISBN 978-0262631327.

[52] J. Eliot B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Office Information Systems*, 8(2):103–139, April 1990.

[53] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, USA, January 1999.

[54] Andrew C. Myers. Mostly-static decentralized information flow control. Technical Report MIT/LCS/TR-783, Massachusetts Institute of Technology, Cambridge, MA, USA, January 1999. Ph.D. thesis.

[55] 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.

[56] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow. Software release, July 2006. Available from `http://www.cs.cornell.edu/jif/`.

[57] Daniel Myers, Jennifer Carlisle, James Cowling, and Barbara Liskov. MapJAX: Data structure abstractions for asynchronous web applications. In *Proc. 2007 USENIX Annual Technical Conference*, pages 101–114, Santa Clara, CA, USA, June 2007.

[58] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, Montréal, QC, Canada, June 1998.

[59] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. 12th International Compiler Construction Conference*, pages 138–152, Warsaw, Poland, April 2003. LNCS 2622.

[60] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proc. USENIX Annual Technical Conference, FREENIX Track*, pages 183–192, Monterey, CA, USA, June 1999.

[61] OMG. *The Common Object Request Broker: Architecture and Specification*, December 1991. OMG TC Document Number 91.12.1, Revision 1.1.

[62] Oracle Corporation. Java SE 7 Remote Method Invocation (RMI)-related APIs & Developer Guides, 2011. Available from `http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/`.

[63] Krzysztof Ostrowski, Ken Birman, Danny Dolev, and Jong Hoon Ahnn. Programming with live distributed objects. In *Proc. 22nd European Conference on Object-Oriented Programming (ECOOP)*, pages 463–489, Paphos, Cyprus, July 2008.

[64] Venugopalan Ramasubramanian and Emin Gün Sirer. Beehive: $O(1)$ lookup performance for power-law query distributions in peer-to-peer overlays. In *Proc. 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 99–112, San Francisco, CA, USA, March 2004.

[65] Sean Rhea, Brighten Dodfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: A public DHT service and its uses. In *Proc. ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 73–84, Philadelphia, PA, USA, August 2005.

[66] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the OceanStore prototype. In *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, San Francisco, CA, USA, March 2003.

[67] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL)*, pages 93–104, San Antonio, TX, USA, January 1999.

[68] Anthony I. T. Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th ACM Symposium on Operating System Principles (SOSP)*, pages 188–201, Banff, AB, Canada, October 2001.

[69] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.

[70] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[71] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: Concurrent programming for modern architectures. In *Proc. 12th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 271, San Jose, CA, USA, March 2007.

[72] Manuel Serrano, Erick Gallesio, and Florian Loitsch. HOP, a language for programming the Web 2.0. In *Proc. 1st Dynamic Languages Symposium*, pages 975–985, Portland, OR, USA, October 2006.

[73] Liuba Shrira, Hong Tian, and Doug Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Proc. 9th ACM/IFIP/USENIX International Middleware Conference (Middleware)*, pages 42–61, Leuven, Belgium, December 2008.

[74] Chunqiang Tang, DeQing Chen, Sandhya Dwarjadas, and Michael L. Scott. Integrating remote invocation and distributed shared state. In *Proc. 18th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 30–39, Santa Fe, NM, USA, April 2004.

[75] Erik M. Volz. Personal communication, September 2011.

[76] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *Proc. IEEE 1998 Symposium on Security and Privacy*, pages 52–63, Oakland, CA, USA, May 1998.

[77] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *Proc. 16th International World Wide Web Conference (WWW)*, pages 341–350, Banff, AB, Canada, 2007.

[78] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.

[79] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.

[80] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proc. 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–308, San Francisco, CA, USA, April 2008.

[81] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE 2003 Symposium on Security and Privacy*, pages 236–250, Berkeley, CA, USA, May 2003.