

Compositional Security for Reentrant Applications

Ethan Cecchetti Siqui Yao Haobin Ni Andrew C. Myers

Department of Computer Science
Cornell University
{ethan,yaosiqui,haobin,andru}@cs.cornell.edu

Abstract

The disastrous vulnerabilities in smart contracts sharply remind us of our ignorance: we do not know how to write code that is secure in composition with malicious code. Information flow control has long been proposed as a way to achieve compositional security, offering strong guarantees even when combining software from different trust domains. Unfortunately, this appealing story breaks down in the presence of reentrancy attacks. We formalize a general definition of reentrancy and introduce a security condition that allows software modules like smart contracts to protect their key invariants while retaining the expressive power of safe forms of reentrancy. We present a security type system that provably enforces secure information flow; in conjunction with run-time mechanisms, it enforces secure reentrancy even in the presence of unknown code; and it helps locate and correct recent high-profile vulnerabilities.

1 Introduction

Compositional security remains a fundamental concern for software security. Code might appear secure, yet expose vulnerabilities when it interacts with other code. Blockchain smart contracts offer multiple prominent recent examples of this problem [43, 44, 46], but other instances exist. JavaScript code is difficult to secure when running on the same web page as code from a different source [13, 27, 38]. Web browsers themselves have fallen victim to attacks when executing code on web pages [1, 2]. In these settings, securing code in isolation is not sufficient. Reasoning about the behavior of a combination of interacting systems, however, is notoriously difficult. This work therefore aims for a way to build software with *compositional* security guarantees, meaning the security of an entire system follows from the security of its components.

Complex control flow, and in particular reentrant executions, pose a fundamental challenge for compositional security. Developers are increasingly building applications from separate communicating services that may belong to different trust domains [18, 56]. In such architectures, one service waiting for another to respond must be prepared to handle separate incoming requests. These *reentrant calls* effectively interrupt the execution of the application and, if the developer is not careful, can catch it in an inconsistent state, creating security vulnerabilities [3].

Reentrancy security has received much more attention since July 2016, when the Decentralized Autonomous Organization (DAO)—an Ethereum smart contract intended to function as a distributed venture capital fund—lost \$50 million in tokens to such an attack, making global news [46]. Since then, a variety of methods have emerged to analyze or eliminate reentrancy attacks [4, 14, 17, 26, 36], but vulnerabilities continue to appear. For example, a January 2019 audit uncovered a reentrancy vulnerability in the Uniswap decentralized exchange [15]. The attack leveraged a subtle interaction between two contracts that were secure in isolation, and a third malicious contract. The first contract implicitly assumed the second would not call the malicious contract. Because the interface could not specify this expectation, developers used the exchange for a token standard that allowed for such calls. This choice led to the theft of \$25 million worth of tokens in April 2020 [44], over a year after the original vulnerability disclosure.

We follow our previous suggestion [12] and use a general language-based technique to obtain compositional security even in the presence of reentrant executions. We define and enforce security using a semantic specification of trust in the form of information flow labels. Information flow control (IFC) has long been an appealing technique for obtaining compositional security and has proven useful in practice [20]. IFC type systems can guide software development with compile-time checking and provably enforce strong security guarantees such as noninterference. But while IFC is a good starting point for compositional security, existing approaches break down in the presence of reentrancy. Standard IFC rules either reject useful, secure applications by blocking requests from untrusted sources, or they allow insecure applications that are vulnerable to reentrancy attacks. We extend standard IFC rules to define a secure type system that efficiently and provably prevents attacks, yet is expressive enough to build interesting applications.

This approach addresses fundamental shortcomings of existing solutions. Current stand-alone reentrancy analyses [4, 26, 36] are non-compositional. That is, analyzing two pieces of code separately might not yield useful guarantees about their combination—the exact failing that led to the Uniswap attack. These tools also focus specifically on blockchain smart contracts. While smart contracts have provided notable recent examples of reentrancy vulnerabilities, similar exploits appear elsewhere [1–3] and there is no reason to limit solutions. The focus on smart contracts and the absence of trust specifications forces the tools to rely on contract boundaries—a syntactic construct—as a proxy for semantic security boundaries. This choice leads to a reentrancy definition we call *object reentrancy* that can judge the security of two semantically equivalent implementations differently, merely because the code has different structure.

There exist other language-based approaches that provide compositional guarantees and consider reentrancy, but they are again smart-contract focused and use object-based reentrancy definitions. Moreover, some limit expressiveness by outlawing reentrancy entirely [14, 17], while others provide only heuristic reentrancy protection [9, 50, 51]. In addition, they universally assume that all code is written in the same language. This strong assumption clearly does not apply to open systems where anyone can submit code, like Ethereum contracts or JavaScript on web pages. Even in closed systems with controlled environments and known code, new code might need to interact with legacy applications that do not respect the language rules.

We address these shortcomings by defining a new general-purpose security type system that tracks the integrity of data and computation. In addition to providing standard IFC data security guarantees, the type system combines with a run-time mechanism to provably eliminate dangerous reentrancy while allowing safe reentrancy. The guarantees, moreover, continue to hold even when trusted code interacts with untrusted code that does not obey the same restrictions.

The remainder of the paper is structured as follows:

- Examples in Section 2 show the complexity of reentrancy.
- Section 3 provides background on information flow control and exposes its failure to handle reentrancy.
- Section 4 presents a new definition of security in the presence of reentrancy.
- Section 5 defines SeRIF, a core calculus that eliminates insecure reentrancy by combining a static IFC type system with a dynamic locking mechanism.
- Section 6 shows formally that SeRIF enforces our formal, compositional security condition.
- Section 7 describes a prototype type checker implementation and our experience using it on realistic programs.
- Section 8 discusses related work in more detail and Section 9 concludes.

2 Motivation

By their very nature, reentrancy vulnerabilities are often hard to spot. For instance, the attack on Ethereum’s Decentralized Autonomous Organization (DAO) was considered subtle at the time [16], despite being one

```

1 contract Uniswap {
2   Token tX, tY;
3
4   function sellXForY(uint xSold) returns uint {
5     uint prod = tX.getBal(this) * tY.getBal(this);
6     uint yKept = prod / (tX.getBal(this) + xSold);
7     uint yBought = tY.getBal(this) - yKept;
8
9     assert tX.transferTo(msg.sender, this, xSold);
10    assert tY.transferTo(this, msg.sender, yBought);
11    return yBought;
12  }
13 }
14
15 contract Token {
16   function transferTo(address from, address to,
17     uint amount) returns bool {
18     ... // check and update balances
19
20     from.alertSend(to, amount);
21     to.alertReceive(from, amount);
22     return true;
23   }
24 }

```

Figure 1. Distilled Solidity [54] code for the Uniswap bug.

of the simplest examples of reentrancy. To build intuition, we present three running examples of applications with reentrancy. Though we have distilled them to their core components, the vulnerabilities have undermined security in real-world applications.

2.1 Uniswap

We begin with the Uniswap/Lendf.me reentrancy vulnerability first identified in January 2019 [15] and later exploited in April 2020 [44]. The vulnerability arises from the combination of two contracts. Though each may be considered secure in isolation, they combine in unexpected ways, demonstrating the need for *compositional* reentrancy security.

Uniswap is a smart contract platform where users can exchange one token for another. Figure 1 shows a simplified portion of the Uniswap contract: the exchange function "sellXForY" allows users to sell tokens of type X for tokens of type Y . Uniswap determines the exchange rate by the amount of X and Y it currently holds. It holds the product of the two amounts constant, allowing Uniswap to maintain the same total asset value as exchange rates fluctuate. The tokens themselves are implemented by independent contracts.

To perform an exchange, Uniswap queries its balance with each token, computes how much of token Y the user bought, and transfers tokens by calling `transferTo` on each token contract. Tokens execute transfers by first checking and updating balances, and then notifying the sender and recipient, allowing each in turn to execute arbitrary code.

Both contracts appear secure in isolation, following the best-practice recommendation of modifying state before making external calls to avoid reentrancy concerns [55]. However, when combined, they expose a dangerous exploit. Suppose the exchange begins with 6 units each of X and Y .

1. An attacker \mathcal{A} calls `sellXForY` selling 6 units of X .

```

1  getOrCompute(key, computeFun) {
2      i = _getIdx(key) // index of mapping if it exists
3      if (mappings[i] == null) {
4          mappings[i] = computeFun();
5      }
6      return mappings[i];
7  }

```

Figure 2. The `getOrCompute` function of a key–value store. Here `mappings` is an array that the store resizes as mappings are added.

2. Uniswap correctly computes $\text{prod} = 36$ and $\text{yBought} = 3$.
3. Uniswap calls token X to transfer 6 units from \mathcal{A} .
4. The token notifies \mathcal{A} , giving it control of the execution.
5. Before returning, \mathcal{A} calls `sellXForY` again to sell 6 more units of X , reentering the Uniswap contract.
6. Uniswap now has 12 units of X , but still 6 units of Y , so it computes $\text{prod} = 72$, not 36, and $\text{yBought} = 2$.

When the dust settles, Uniswap has 18 units of X and only 1 unit of Y , having given \mathcal{A} an extra unit of Y and having broken the invariant that the product of the balances is 36. If desired, \mathcal{A} can reclaim their original 12 units of X for only 2 units of Y , keeping the other 3 as illicit profit.

The fundamental problem is a mismatch between Uniswap’s notion of secure behavior and the token’s. The token correctly checks that all transfers are valid and authorized and follows programming patterns that avoid (internal) reentrancy concerns. No user can transfer more tokens than they have. Uniswap, however, implicitly assumes that `transferTo` transfers tokens and returns *without allowing an adversary to call Uniswap before it reestablishes the invariant that $\text{prod} = 36$* .

This insight suggests two approaches to fixing the bug: (1) token contracts could respect Uniswap’s assumption by not calling unknown, untrusted code, or (2) Uniswap could stop relying on the assumption. Current platforms provide no way to guarantee the first option. Uniswap could state its assumption in documentation, but there is no technical means of specifying or enforcing it. Tokens that violate it could continue to freely interface with Uniswap, with disastrous results. The exchange can, however, implement the second option by acquiring a run-time lock on entry to the contract. It could then recognize the above attack and produce an error at step 5.

Our approach detects this vulnerability and can specify and correctly analyze either proposed solution. Among existing tools, only Nomos [17] can express the assumption of approach (1), which it mandates to statically eliminate all reentrancy. Other tools either cannot properly secure the application [9, 50, 51] or force the use of computationally expensive dynamic locks even when they are unnecessary [4, 14].

2.2 Key–Value Store

Smart contracts have made reentrancy concerns highly visible, but reentrancy is not unique to that domain. It has led to multiple critical security vulnerabilities in Internet Explorer [1, 2], and is a known concern for any application executing user-provided code [3].

For example, key–value stores often compute missing mappings with user-supplied functions [42, 48]. A careless implementation of this functionality can enable dangerous reentrancy. Consider the code in Figure 2, along with a clear method that frees mappings and installs a new empty array. An attacker can call `getOrCompute`, providing as arguments an unmapped key and a malicious function that calls `clear` and then returns a value. First `getOrCompute` computes `i`, then it calls the malicious function, which calls `clear`

```

1 contract TownCrier {
2   address[] requesters, callbacks;
3
4   function deliver(uint reqId, bytes data) {
5     if (msg.sender == SERVICE_ADDR
6         && requesters[reqId] != 0) {
7       requesters[reqId] = 0;
8       SERVICE_ADDR.call{value: FEE}("");
9       callbacks[reqId].call(bytes);
10    }
11  }
12
13  function cancel(uint reqId) {
14    if (msg.sender == requesters[reqId]) {
15      requesters[reqId] = 0;
16      msg.sender.call{value: FEE}("");
17    }
18  }
19 }

```

Figure 3. Solidity [54] code for simplified partial Town Crier contract. Here `SERVICE_ADDR` is TC’s trusted wallet address, and `FEE` is the request fee.

and replaces the mappings array. Finally `getOrCompute` attempts to write the attacker-provided value into index `i` of the new array.

If `i` is large—which is likely if the store previously contained many mappings—the write would be past the end of the new empty array. In languages like C/C++ without array bounds checking, an attacker-provided value would thus be written into an arbitrary memory location, enabling remote code execution or other critical security vulnerabilities. Even memory-safe languages like Java explicitly recommend developers check for reentrant modifications and throw exceptions [42].

Notably, while this attack appears very similar to concurrent-modification attacks on key–value stores, it requires no concurrency. Single-threaded applications or applications using simple thread-level locking are still vulnerable.

2.3 Town Crier

Banning all reentrancy might seem appealing, but this solution would be overly restrictive. Town Crier (TC) [64] is an example where safe reentrancy enables important functionality. TC provides authenticated data to smart contracts upon request. Users place requests with a smart-contract front end, and TC processes them asynchronously and delivers the data to user-specified callbacks when it is available. TC also allows users to cancel pending requests for a refund. Figure 3 shows simplified versions of TC’s `deliver` and `cancel` methods.

Invoking a user-provided callback in `deliver` opens the possibility of reentrant calls. Unlike in the previous examples, however, these calls are safe. By ensuring that the request status is updated (lines 7 and 15) before calling untrusted code (lines 9 and 16), TC prevents attackers from receiving refunds for canceling requests that are mid-delivery or already canceled. Honest users, however, can still respond to data they receive from one request by creating or canceling *other* requests.

For instance, a user contract may ask TC to function as a real-world timer and alert it at a specific real-world time. When woken up, the contract might determine that it needs to wait longer and request that TC send another alert, say, 2 hours later. A different user could make multiple parallel requests to retrieve the same data, e.g., a stock price, from several sources. Once enough responses have arrived, the user might wish

to cancel the outstanding requests to reduce costs. Both of these patterns require safe reentrant calls into TC. This work aims to allow this *secure* reentrancy while still eliminating the vulnerabilities described above.

3 Information Flow Control

To obtain compositional security, it is natural to build on top of information flow control (IFC), a classic way to obtain compositional security guarantees such as noninterference [23]. Most IFC work has focused on data confidentiality [49, 58], but IFC can also protect integrity [8, 60] and availability [65]. As our goal is to guard against attackers performing unexpected calls into trustworthy code, we track only integrity.

IFC systems assign labels to computation and data within a system. As information flows through the system, the label on the destination of information is constrained to be no less restrictive than the label on its source. Since our goal is to enforce integrity, less trusted information should be prevented from influencing more trusted information.

Secure information flow is statically enforceable by a type system [49]. When linking separate code modules together, the security guarantees offered by the type system are automatically compositional, as long as the linked modules agree on types at interface boundaries and account for the confidentiality and integrity of the code itself [5]. Of course, real-world systems often have to interact with user-provided code or legacy applications that do not obey the rules of the type system. As we show, such noncompliant code can only violate the security guarantees of code that expresses trust in it.

3.1 Label model

We specify integrity using a set of integrity labels \mathcal{L} and give each piece of data x a label ℓ_x representing its trust level. The labels have a reflexive, transitive relation $\ell_1 \Rightarrow \ell_2$, which we read “ ℓ_1 acts for ℓ_2 ,” to denote that ℓ_1 is at least as trusted as ℓ_2 . That is, anything that can influence data labeled ℓ_1 can also influence data labeled ℓ_2 .¹ Data x can thus safely influence data y only when $\ell_x \Rightarrow \ell_y$. Influence can be either *explicit*—by assigning x directly to y —or *implicit*—by conditioning on x and assigning different values to y in each branch. For explicit flows, a simple check that $\ell_x \Rightarrow \ell_y$ at the point of assignment is sufficient. To control implicit flows, a *program counter label*, written pc , tracks the integrity of the computation itself, as is standard [49]. Inside a branch conditioned on x , the value of x has influenced control flow, so we require the constraint $\ell_x \Rightarrow pc$. Assigning a variable y to some value then requires $pc \Rightarrow \ell_y$, ensuring transitively that $\ell_x \Rightarrow \ell_y$.

\mathcal{L} must also have some additional structure. Any pair of labels ℓ_1 and ℓ_2 must have a join, denoted $\ell_1 \vee \ell_2$, and a meet, denoted $\ell_1 \wedge \ell_2$. The join is the least upper bound and the meet is the greatest lower bound, so

$$\begin{aligned} \ell_1 \vee \ell_2 \Rightarrow \ell &\iff \ell_1 \Rightarrow \ell \text{ and } \ell_2 \Rightarrow \ell \\ \ell \Rightarrow \ell_1 \wedge \ell_2 &\iff \ell \Rightarrow \ell_1 \text{ and } \ell \Rightarrow \ell_2. \end{aligned}$$

We can then safely label information influenced by both ℓ_1 and ℓ_2 with label $\ell_1 \vee \ell_2$, for example. Lastly, the join and meet operators must distribute: $\ell_1 \vee (\ell_2 \wedge \ell_3) = (\ell_1 \vee \ell_2) \wedge (\ell_1 \vee \ell_3)$. These properties collectively make $(\mathcal{L}, \Rightarrow)$ a *distributive lattice*.

This additional structure supports the precision and flexibility of our approach to enforcing reentrancy security, discussed in Section 5.2. Luckily, existing label models are typically distributive lattices, including two-point lattices, subset lattices of permissions [61], and free distributive lattices over a set of principals [6, 39]. In smart-contract systems, for example, it is natural to view contracts themselves as principals with

¹Most IFC systems use *flows-to*, denoted \sqsubseteq . We use acts-for as we find it intuitive, and the two mean the same thing when only tracking integrity.

different trust relationships among them. We might then employ *decentralized* information flow control [40] where labels are constructed from principals (e.g., contracts) that can influence data or computation.

3.2 Endorsement

Strictly enforcing IFC allows systems to enforce strong security properties like noninterference, which forbids *any* influence from untrusted information to trusted information. Noninterference, however, is too restrictive to build real applications, so practical IFC systems allow *downgrading*. Downgrading integrity, known as *endorsement* [66], treats information with a low-integrity label as being more trustworthy than its source would indicate.

From the IFC perspective, services like smart contracts endorse frequently, though implicitly. They expose functions that accept calls from untrusted users, yet modify trusted local state. In other words, untrusted state affects trusted state, which an IFC system should only allow via endorsement.

Existing IFC languages support these trusted functions, but make them explicit. For example, the Jif language [37] supports *autoendorse* methods that can be called by an untrusted caller and that boost the integrity of the *pc* label on entry.

Viewed from the perspective of *pc* integrity, reentrancy attacks all exhibit a distinctive pattern: they involve trusted (high-integrity) code calling lower-integrity code, which then calls back into high-integrity code by exploiting endorsement. However, existing endorsement mechanisms in Jif and other systems [19, 32, 35, 61] do not prevent this potentially dangerous control-flow pattern. These IFC systems are thus vulnerable to reentrancy attacks. Preventing reentrancy attacks requires new restrictions on endorsement.

4 Reentrancy and Security

The examples in Section 2 show the need across application domains to constrain reentrancy without eliminating it entirely. We build on our previous work [12] to provide flexible definitions of reentrancy and security based on information flow control. This choice gives access to existing IFC tools and techniques with their strong data security guarantees, while making possible a precise, semantic specification of security.

4.1 Defining Reentrancy

Prior work [4, 14, 26, 36] focuses on smart contracts and defines reentrancy in those terms: if contract *A* calls contract *B*, which calls back into contract *A*, the second call, and thus the entire execution, is considered reentrant. If no calls to *A* occur before the call to *B* returns, the execution is non-reentrant. We refer to this notion of reentrancy as *object reentrancy*, viewing contracts as a form of object.

We avoid object reentrancy because it relies on object boundaries—a fundamentally syntactic construct—to define security. Instead we define reentrancy with respect to the integrity level of computation. As integrity levels are part of a semantic security specification, using them to define a security-relevant property is sensible. This view leads to the following informal definition.

Definition 1 (*ℓ*-Reentrancy (informal)). If computation C_1 calls computation C_2 , which then (possibly indirectly) calls C_3 , the execution is reentrant with respect to label ℓ , or *ℓ-reentrant*, if C_1 and C_3 are trusted at ℓ , but C_2 is not.

Note that C_1 and C_3 may be the same or different, as long as they are both trusted at ℓ .

Figure 4 depicts how *ℓ*-reentrancy relates to object reentrancy. If an entire object is trusted at ℓ and nothing else is (Figure 4a), *ℓ*-reentrancy and object reentrancy align. However, object and trust boundaries may differ, leading to different definitions. If a trusted operation in *A* calls untrusted *B*, a call to an *untrusted* portion of *A* (Figure 4b), would be considered reentrant in an object-based definition but not *ℓ*-reentrancy.

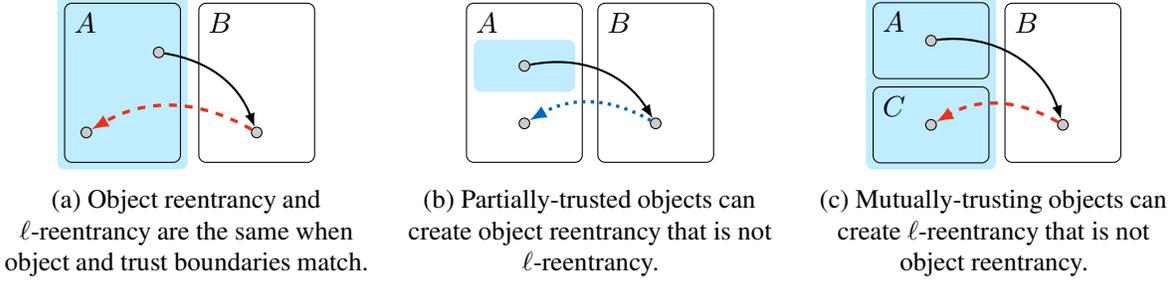


Figure 4. Comparing ℓ -reentrancy to object reentrancy. Boxes represent objects, the blue shaded region is high-integrity code, and arrows represent calls.

Such a call could correspond to a Town Crier user updating a request callback during data delivery or a web app accessing untrusted user profile data while modifying a trusted billing key–value store. These operations are never dangerous, as low-integrity operations cannot damage high-integrity data. By contrast, one application may be split across multiple mutually trusting objects. For example, such a split in Ethereum’s Parity Wallet led to two famous attacks [10, 43]. For an application split across A and C , if A calls B , then a call from B into C (Figure 4c) is a reentrant call into the application. By relying on trust levels, ℓ -reentrancy properly identifies this pattern as reentrancy, while object reentrancy does not.

To employ ℓ -reentrancy, each operation needs an integrity level. Conveniently, the pc label used to control implicit information flows (Section 3.1) provides such a label. It combines the integrity of the code and the integrity of data influencing the control flow to specify how trusted an operation is to execute when it does, making it ideal to define a property of trusted and untrusted operations calling each other.

4.2 Reentrancy Security

While ℓ -reentrancy defines reentrancy based on integrity patterns of the control flow, it does not tell us when it is secure. An option taken by some work [14, 17] is to declare all reentrancy (according to their definition) dangerous and to outlaw it entirely. With an appropriate definition of reentrancy, this would eliminate vulnerabilities, but safe reentrancy has legitimate uses, as illustrated by the Town Crier example.

To eliminate the need for difficult manual reentrancy analysis, we define “secure reentrancy” as reentrancy that programmers can ignore when analyzing correctness. In general, a safe way to accomplish this goal is to ensure that reentrancy cannot enable program behaviors that would not exist without it. These behaviors could be program invariants, such as Uniswap holding the product of its asset quantities constant or the key–value store never writing to unallocated memory; they could be statements about how state changes, like Town Crier’s request ID monotonically increasing; or they could be more complex properties like noninterference.

Programmers cannot hope to guarantee properties that unknown or untrusted code can directly violate, so our definition ignores such properties entirely. Specifically, ℓ -reentrancy security considers only properties defined over state trusted at label ℓ . We refer to these as ℓ -integrity properties, leading to the following security definition, depicted visually in Figure 5.

Definition 2 (Reentrancy Security (informal)). A program is ℓ -reentrancy-secure if every ℓ -integrity property, such as a program invariant, that holds for all non- ℓ -reentrant executions holds for all executions.

Definition 2 specifies a semantic notion of security and helps identify safe forms of reentrancy. For instance, a high-integrity computation making a low-integrity call as its last operation—in tail position—no longer needs high integrity. That is, any reentrant call will have the same effect as making a second, non-reentrant call after the first computation returns. We refer to this secure form of reentrancy as *tail reentrancy*.

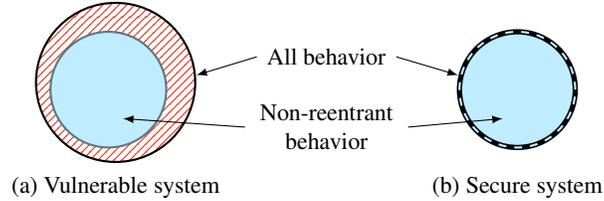


Figure 5. The set of possible behaviors in a secure vs a vulnerable system. In a vulnerable system, reentrancy can introduce behaviors not possible without it. In a secure system, all behaviors are possible in non-reentrant executions.

Tail reentrancy also provides a principled explanation for a common smart-contract programming best practice: performing all state modifications before calling other contracts [55]. Done properly, this design pattern ensures that all reentrant calls are tail-reentrant, and thus safe.

Definition 2 is also flexible. For a specific application, we could refine it to require only that reentrancy does not violate particular programmer-specified application properties. To keep annotation burden low and to avoid the need to specify detailed program properties, our definition requires that ℓ -reentrant executions maintain *all* properties that hold without reentrancy. However, the later formal definition (Definition 9) allows such refinement simply by restricting a universal quantifier.

4.3 Enforcing Reentrancy Security

As described above, ℓ -reentrancy occurs when high-integrity code calls low-integrity code that then calls back into high-integrity code before returning. IFC only permits this pattern through the autoendorse mechanism described in Section 3.2. Many services, including the examples in Section 2, require untrusted users to make requests into trusted code, making some version of autoendorse necessary. We therefore allow it, but with additional restrictions.

In particular, endorsement of control flow is restricted by *locking integrity*. When a function endorses the integrity of the control flow to label ℓ , integrity ℓ is locked, preventing further endorsement up to ℓ until the original call returns. Locking allows an honest user to invoke a service one or more times in sequence using a call-and-return pattern, but prevents an adversary from reentering into high-integrity code.

The semantics of these locks is to prevent autoendorsement from granting integrity that is locked. A trusted operation is then always given the chance to reestablish any high-integrity invariants or properties it may have temporarily invalidated before an attacker can invoke another trusted operation. To safely autoendorse from integrity pc_1 to integrity pc_2 , for any operation pc_2 is trusted to perform, either pc_1 must already be trusted at that level or the requisite integrity must be unlocked. Formally, when integrity ℓ_L is locked, then for all labels ℓ , if $\ell_L \Rightarrow \ell$ and $pc_2 \Rightarrow \ell$, then $pc_1 \Rightarrow \ell$. The definition of lattice join quickly shows that this rule is equivalent to $pc_1 \Rightarrow pc_2 \vee \ell_L$.

We could track and enforce locks statically, as part of the type system, or dynamically in the runtime. Static locking—proving that a dynamic lock would never prevent execution—imposes no overhead and avoids unexpected errors at run time. Unfortunately, purely static locks interact poorly with code that may not enforce the same guarantees. If some unknown code might call autoendorse functions—violating a static lock, meaning a dynamic lock would halt execution—a sound type system must assume the worst and prevent all calls to that code when integrity may be locked. This highly restrictive outcome would violate a core design goal of this work: providing compositional security even when interacting with unknown code. Dynamic locks avoid this constraining over-approximation at the expense of run-time cost.

We therefore take a hybrid approach and separate locked integrity into a static component and a dynamic one. The type system automatically adds endorsed control flow to the static component, but programmers can explicitly move integrity from the static component to the dynamic one. This approach achieves the

$$\begin{aligned}
f, m, x &\in \mathcal{V} \quad (\text{variable, method, and field names}) \\
\ell, pc &\in \mathcal{L} \quad (\text{integrity labels}) \\
t &::= \text{unit} \mid \text{bool} \mid \text{ref } \tau \mid C \\
\tau &::= t^\ell \\
CL &::= \text{class } C[\ell] \text{ extends } C \{ \bar{f} : \bar{\tau} ; K ; \bar{M} \} \\
K &::= C(\bar{f} : \bar{\tau}) \{ \text{super}(\bar{f}) ; \text{this}.\bar{f} = \bar{f} \} \\
M &::= \tau m \{ pc \gg pc ; \ell \} (\bar{x} : \bar{\tau}) \{ e \} \\
v &::= x \mid () \mid \text{true} \mid \text{false} \mid \iota \mid \text{null} \mid \text{new } C(\bar{v}) \\
e &::= v \mid \text{if} \{ pc \} v \text{ then } e \text{ else } e \\
&\quad \mid \text{ref } v \tau \mid !v \mid v := v \\
&\quad \mid (C)v \mid v.f \mid v.m(\bar{v}) \\
&\quad \mid \text{endorse } v \text{ from } \ell \text{ to } \ell \mid \text{lock } \ell \text{ in } e \\
&\quad \mid \text{let } x = e \text{ in } e
\end{aligned}$$

Figure 6. Syntax for SeRIF

run-time efficiency and predictability of static mechanisms when security can be proved statically, while still supporting safe interaction with unknown or untrusted code through more expressive dynamic locks.

The calculus does not specify how to implement dynamic locks. They could be built into the runtime, tracked by a security monitor, or even implemented as a library. So long as all code trusted at level ℓ is well-typed and agrees on *some* protocol to enforce the dynamic portion of the locks, the system will preserve ℓ -reentrancy security. There is no requirement that untrusted check integrity locks statically or dynamically.

5 A Core Calculus for Secure Reentrancy

We present the Secure-Reentrancy Information Flow Calculus (SeRIF), an object-oriented core calculus that models how a programming language can implement the above ideas. Figure 6 gives the syntax for SeRIF. It extends Featherweight Java (FJ) [30] with information flow labels and, to support mutation, also reference cells [45, Chapter 13].

SeRIF employs fine-grained IFC, so each type τ consists of a base type t and an integrity label ℓ . For simplicity, we limit base types to unit, bool, references, and object types. To simplify proofs, null references are allowed.

Class and method definitions extend those in FJ with integrity labels. To model distributed systems, we consider code a form of data that may come from multiple sources, so each class definition CL includes a label ℓ_C for the integrity of the code.

A method definition M contains labels $pc_1 \gg pc_2; \ell$. Most IFC systems give functions a single pc label, but SeRIF has two: pc_1 specifies the minimum integrity required to call m , while pc_2 specifies the integrity at which m operates. Separating these labels supports autoendorsement as described in Section 3.2. If $pc_1 \not\approx pc_2$, then m is an autoendorse function. Both pc labels are bounded by ℓ_C , so code may only perform operations that ℓ_C is trusted to perform. The label ℓ specifies the locks method m promises not to violate.

The if syntax includes the pc label used for the branches. We make this label explicit only to simplify the operational semantics. In practice, it is easy to infer automatically.

The endorse expression endorses data as in other IFC systems with downgrading. The term lock ℓ in e converts static locks to dynamic ones. In the operational semantics, e executes with ℓ dynamically locked, so the type system can safely release any static lock on ℓ when type-checking e .

Expression subterms consist mostly of (open) values, not arbitrary expressions. In particular, let statements are the only way to sequentially compose computation.

Because SeRIF is object-oriented, it can model interacting services and reentrancy concerns. An application or contract implementation is a class, and a contract or instance of that application is an object of that class type, allowing easy interaction between different services. Moreover, inheritance allows applications that share common features to inherit from a common parent. For instance, a blockchain smart contract system can be modeled by having all contracts inherit from a `Contract` class that implements tracking of currency.

5.1 SeRIF Operational Semantics

SeRIF has a small-step substitution-based semantics. Most rules are standard for an object-oriented language with mutable references [30, 45], with a few additions for security.

Because expressions are built mostly out of values, evaluation contexts are simple. Indeed, let expressions be the only surface syntax to serve as evaluation contexts. We introduce three new syntactic forms as evaluation contexts to enable precise tracking of function boundaries, execution integrity, and dynamic locks. These *statements* are denoted by s .

$$\begin{aligned} E & ::= [\cdot] \mid \text{let } x = E \text{ in } e \mid \text{return}_\tau E \mid E \text{ at-pc } pc \mid E \text{ with-lock } \ell \\ s & ::= E[e] \end{aligned}$$

Semantic steps are defined on a pair of a statement s and a *semantic configuration*: a four-tuple $\mathcal{C} = (CT, \sigma, \mathcal{M}, L)$. Unlike in FJ, the class table CT is explicit, as the security definitions in Section 6 quantify over possible class tables. A heap σ maps locations to value–type pairs, and Σ_σ denotes the location-to-type mapping induced by σ . That is, $\Sigma_\sigma(\iota) = \tau$ if and only if $\sigma(\iota) = (v, \tau)$ for some v . The final two elements, \mathcal{M} and L are both lists of integrity labels. \mathcal{M} tracks the integrity of executing code, and L tracks the dynamic portion of the currently-locked integrity. For notational ease, we reference the components of \mathcal{C} freely when only one group is in scope and we write $\mathcal{C}[X/L]$ to denote $(CT, \sigma, \mathcal{M}, X)$, and similarly for σ and \mathcal{M} .

Figure 7 presents selected semantic rules. The complete semantics is in Figure 9 (Appendix A). In the semantic rules, v refers to a closed value, not a variable. In addition to many standard rules, the rules E-LOCK and E-UNLOCK dynamically lock and unlock labels. The semantics abstracts out the many possible lock implementations, merely tracking the set of locked labels and defining where to check them. The rules for conditionals (E-IFT and E-IFF) now include tracking terms.

The key rule is E-CALL. It looks up the definition of a method with *mbody* (Appendix A) and performs several dynamic checks: it verifies that the arguments all have the correct types, that the caller has sufficient integrity to invoke the function, and that calling the method does not violate any dynamically locked label $\ell \in L$.

Dynamic Security Checks. Four rules—E-REF, E-ASSIGN, E-CALL, and E-RETURN—contain dynamic checks for type safety and information security. These checks prevent untrusted code from placing ill-typed values in the heap or passing them to trusted code. They similarly prevent untrusted code from modifying trusted heap locations in any way. Such checks are critical for trusted code to safely interact with ill-typed attacker code in any information flow system. While we do not detail how to implement dynamic typing or label checks here, there is considerable research into both. Gradually typed languages do run-time type checking [53], and distributed IFC systems include run-time label checks [e.g., 22, 35, 62]. Moreover, when all high-integrity code is well-typed, it is sufficient to isolate memory between objects, as in Ethereum contracts [57], and to execute run-time checks when entering trusted code.

$$\begin{array}{c}
\text{[E-IFT]} \quad \frac{}{\langle \text{if } \{pc\} \text{ true then } e_1 \text{ else } e_2 \mid \mathcal{C} \rangle \longrightarrow \langle e_1 \text{ at-pc } pc \mid \mathcal{C} \rangle} \\
\text{[E-ATPC]} \quad \frac{}{\langle v \text{ at-pc } pc \mid \mathcal{C} \rangle \longrightarrow \langle v \mid \mathcal{C} \rangle} \\
\text{[E-REF]} \quad \frac{\iota \notin \text{dom}(\sigma) \quad \Sigma_\sigma \vdash v : \tau \quad \mathcal{M} = \mathcal{M}', \ell_m \quad \ell_m \triangleleft \tau}{\langle \text{ref } v \ \tau \mid \mathcal{C} \rangle \longrightarrow \langle \iota \mid \mathcal{C}[\sigma[\iota \mapsto (v, \tau)]]/\sigma \rangle} \\
\text{[E-ASSIGN]} \quad \frac{\Sigma_\sigma(\iota) = \tau \quad \Sigma_\sigma \vdash v : \tau \quad \mathcal{M} = \mathcal{M}', \ell_m \quad \ell_m \triangleleft \tau}{\langle \iota := v \mid \mathcal{C} \rangle \longrightarrow \langle () \mid \mathcal{C}[\sigma[\iota \mapsto (v, \tau)]]/\sigma \rangle} \\
\text{[E-CALL]} \quad \frac{\begin{array}{c} \text{mbody}(C, m) = (\ell_m, \bar{x}, \bar{\tau}_a, pc_1 \gg pc_2, e, \tau) \\ \mathcal{M} = \mathcal{M}', \ell'_m \quad \ell'_m \Rightarrow pc_1 \quad \bigwedge_{\ell \in L} (pc_1 \Rightarrow pc_2 \vee \ell) \\ \Sigma_\sigma \vdash \bar{w} : \bar{\tau}_a \quad e' = e[\bar{x} \mapsto \bar{w}, \text{this} \mapsto \text{new } C(\bar{w})] \end{array}}{\langle \text{new } C(\bar{w}).m(\bar{w}) \mid \mathcal{C} \rangle \longrightarrow \langle \text{return}_\tau (e' \text{ at-pc } pc_2) \mid \mathcal{C}[\mathcal{M}, \ell_m/\mathcal{M}] \rangle} \\
\text{[E-RETURN]} \quad \frac{\Sigma_\sigma \vdash v : \tau \quad \mathcal{M} = \mathcal{M}', \ell_m}{\langle \text{return}_\tau v \mid \mathcal{C} \rangle \longrightarrow \langle v \mid \mathcal{C}[\mathcal{M}'/\mathcal{M}] \rangle} \\
\text{[E-LOCK]} \quad \frac{}{\langle \text{lock } \ell \text{ in } e \mid \mathcal{C} \rangle \longrightarrow \langle e \text{ with-lock } \ell \mid \mathcal{C}[L, \ell/L] \rangle} \\
\text{[E-UNLOCK]} \quad \frac{L = L', \ell}{\langle v \text{ with-lock } \ell \mid \mathcal{C} \rangle \longrightarrow \langle v \mid \mathcal{C}[L'/L] \rangle}
\end{array}$$

Figure 7. Selected small-step semantic rules for SeRIF.

5.2 Type System for SeRIF

The type system for SeRIF contains two different forms for typing judgments: one for values and one for expressions. The typing judgment for values is straightforward for a stateful language. It takes the form $\Sigma; \Gamma \vdash v : \tau$ where Σ is a heap type mapping references to types and Γ is a typing environment mapping variables to types. We write $\Sigma \vdash v : \tau$ when Γ is empty, as we did in Section 5.1.

Values specify no computation so they require no security reasoning. Typing judgments for expressions are more complex, including a standard pc label to track the integrity of the control flow. To secure reentrancy with static locks when possible, they also include a label λ representing locked integrity.

Allowing tail reentrancy while eliminating other forms of ℓ -reentrancy requires treating calls in tail position differently from calls in other positions. We accomplish this goal not by restricting when a given call can occur, but instead by restricting what can occur *after the call returns*. Instead of one lock label, this strategy uses two: an *input lock* λ_I that an expression must maintain to safely execute outside tail position, and an *output lock* λ_O specifying the locks the expression *actually* maintains. The typing judgment now takes the form $\Sigma; \Gamma; pc; \lambda_I \vdash e : \tau \dashv \lambda_O$.

For an expression e to type-check with input lock λ_I , each subexpression of e outside tail position must maintain λ_I . As non-value expressions only appear outside of tail position in let expressions, the following typing rule enforces this restriction.

$$\text{[LET]} \quad \frac{\begin{array}{c} \Sigma; \Gamma; pc; \lambda_I \vdash e_1 : \tau_1 \dashv \lambda'_O \quad \lambda'_O \Rightarrow \lambda_I \\ \Sigma; \Gamma, x : \tau_1; pc; \lambda_I \vdash e_2 : \tau_2 \dashv \lambda_O \end{array}}{\Sigma; \Gamma; pc; \lambda_I \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 \dashv \lambda_O}$$

$$\begin{array}{c}
\text{[IF]} \frac{\Sigma; \Gamma \vdash v : \text{bool}^\ell \quad \ell \Rightarrow pc \quad \ell \triangleleft \tau \quad \Sigma; \Gamma; pc; \lambda_I \vdash e_1 : \tau \dashv \lambda_0 \quad \Sigma; \Gamma; pc; \lambda_I \vdash e_2 : \tau \dashv \lambda_0}{\Sigma; \Gamma; pc; \lambda_I \vdash \text{if}\{pc\} v \text{ then } e_1 \text{ else } e_2 : \tau \dashv \lambda_0} \\
\text{[ASSIGN]} \frac{\Sigma; \Gamma \vdash v_1 : (\text{ref } \tau)^\ell \quad \Sigma; \Gamma \vdash v_2 : \tau \quad \ell \triangleleft \tau}{\Sigma; \Gamma; \ell; \lambda_I \vdash v_1 := v_2 : \text{unit}^{\ell'} \dashv \lambda_0} \\
\text{[CALL]} \frac{\text{mtype}(C, m) = \overline{\tau}_a \xrightarrow{pc_1 \gg pc_2; \lambda_0} \tau_0 \quad \Sigma; \Gamma \vdash v : C^\ell \quad \Sigma; \Gamma \vdash \overline{v}_a : \overline{\tau}_a \quad \ell \Rightarrow pc_1 \quad pc_1 \Rightarrow pc_2 \vee \lambda_I \quad \tau_0 <: \tau \quad pc_2 \vee \ell \triangleleft \tau}{\Sigma; \Gamma; pc_1; \lambda_I \vdash v.m(\overline{v}_a) : \tau \dashv \lambda_0 \vee pc_2} \\
\text{[LOCK]} \frac{\Sigma; \Gamma; pc; \lambda'_I \vdash e : \tau \dashv \lambda'_0 \quad \lambda'_I \wedge \ell \Rightarrow \lambda_I \quad \lambda'_0 \wedge \ell \Rightarrow \lambda_0}{\Sigma; \Gamma; pc; \lambda_I \vdash \text{lock } \ell \text{ in } e : \tau \dashv \lambda_0} \\
\text{[METHOD-OK]} \frac{\lambda_I \Rightarrow pc_2 \quad \ell_C \Rightarrow pc_2 \quad \lambda_I \vee \lambda'_0 \Rightarrow \lambda_0 \quad pc_1 \triangleleft \overline{\tau}_a \quad \Sigma; \overline{x} : \overline{\tau}_a, \text{this} : C^{pc_2}; pc_2; \lambda_I \vdash e : \tau \dashv \lambda'_0 \quad CT(C) = \text{class } C[\ell_C] \text{ extends } D \{ \dots \}}{(D, m) \in \text{dom}(\text{mtype}) \implies \text{mtype}(D, m) = \overline{\tau}_a \xrightarrow{pc_1 \gg pc_2; \lambda_0} \tau}{\Sigma \vdash \tau m\{pc_1 \gg pc_2; \lambda_0\}(\overline{x} : \overline{\tau}_a) \{e\} \text{ ok in } C}
\end{array}$$

Figure 8. Selected typing rules for SeRIF

This rule is standard except that it requires $\lambda'_0 \Rightarrow \lambda_I$, capturing the intuition above: e_1 must maintain at least lock λ_I , as it is outside tail position. Because e_2 is in tail position in this expression, there is no similar restriction on λ_0 .

Figure 8 contains selected typing rules for SeRIF. The notation $\ell \triangleleft \tau$ indicates that data of type τ is no more trusted than ℓ ; that is, $\ell \triangleleft t^{\ell'}$ if and only if $\ell \Rightarrow \ell'$. The rules also use auxiliary lookup functions *fields* and *mtype* and a subtyping relation $<:$ that includes both standard object subtyping and safe relabeling— $t^\ell <: t^{\ell'}$ if and only if $\ell \Rightarrow \ell'$. The complete type system is in Figures 10 and 11 (Appendix A).

Most typing rules (e.g., IF and ASSIGN) are standard for an information flow calculus [49]. The only non-standard rules are those that directly reference or constrain static locks: sequential composition (LET), method calls (CALL), and dynamic locking (LOCK).

Most premises of CALL are standard. They check that the object and arguments have appropriate types and ensure information security of the return type and control flow of the call. They also check that the call does not violate any static locks ($pc_1 \Rightarrow pc_2 \vee \lambda_I$) and that it attenuates trust in the output by the integrity of both the object and the method ($pc_2 \vee \ell \triangleleft \tau$).

This rule has two notable features. The first is not what it requires, but rather what it *does not* require. There is no relation between the static input locks λ_I of the surrounding environment and λ_0 , the locks maintained by the method itself. This lack of constraint is precisely what enables tail reentrancy. A call in tail position need not maintain any locks, so it may result in reentrancy. Outside tail position, however, the LET rule requires that the output locks of the call expression—bounded by the locks maintained by the method—must act for λ_I . CALL and LET therefore combine to enable safe tail reentrancy while ruling out other potentially dangerous reentrancy.

The second feature is that CALL does not maintain locks λ_0 —the locks maintained by the method—but instead only $\lambda_0 \vee pc_2$. This adjustment enables safe interaction with untrusted code that might not enforce the same guarantees as SeRIF. Such code may claim to maintain locks, but fail to do so. Our safeguard follows the principle of decentralized IFC [40]: you can only be hurt by an adversary you trust. We therefore attenuate the claimed lock label λ_0 by the integrity of the code.

Due to SeRIF’s inheritance structure, however, there is no way to determine the exact integrity of the code. The implementation of m may come from C or any of its superclasses or subclasses. We instead need a bound on the implementation’s integrity. The class typing rule METHOD-OK requires that the code’s integrity act for pc_2 to define or override a method with integrity pc_2 . As a result, pc_2 is the most precise

bound on the code’s integrity available to the type system.

To understand the LOCK rule, recall that the lock term is designed to convert static locks to dynamic ones. The type system must ensure that λ_I , the previous input locks, remain locked in some manner, but it can safely release the portion that is dynamically checked. In particular, LOCK splits λ_I into ℓ and some λ'_I such that $\lambda'_I \wedge \ell \Rightarrow \lambda_I$. Now λ_I will remain locked as long as e type-checks with static input lock λ'_I . Similarly, lock ℓ in e actually maintains locks on both λ'_O —the locks e maintains—and ℓ . It is thus safe to trust λ_O up to $\lambda'_O \wedge \ell \Rightarrow \lambda_O$. Notably, allowing these arbitrary label divisions is only secure because the label lattice is distributive. Otherwise, separately locking λ'_I and ℓ could be insufficient to lock λ_I , and similarly for λ'_O and λ_O .

Finally, METHOD-OK defines when a method is well-typed. This rule implements the idea that autoendorse methods statically lock integrity by default. Specifically, it requires $\lambda_I \Rightarrow pc_2$, so any expression outside tail position must respect locks on the new, higher integrity of control flow. The integrity of the code must also act for the integrity with which the function executes ($\ell_C \Rightarrow pc_2$), ensuring code cannot do anything its source is not trusted to do. Next, the locks the method claims to enforce (λ_O) must be maintained both initially (λ_I) and throughout (λ'_O). The last information-security check ($pc_1 \triangleleft \overline{\tau}_a$) guarantees that any code trusted to call the method is also trusted to provide its arguments.

5.3 Modeling Application Operation

We aim to model applications that, like smart contracts, service user requests and may persist state across requests. We represent the current state of the world by a set of class definitions in a class table CT and a state map σ . A single user interaction, which we term an *invocation* I , is a label specifying the user’s integrity and a call to a single method of an object stored in σ .

Execution of an invocation $I = (\iota, m(\overline{v}), \ell)$ with state σ starts from a semantic configuration with the expression, integrity ℓ , and no locks, and step it to completion. The notation $(I, CT, \sigma) \Downarrow \sigma'$ signifies that it terminates in updated state σ' . The following rule formalizes this idea, using $!\iota.m(\overline{v})$ as shorthand for let $o = !\iota$ in $o.m(\overline{v})$.

$$[\text{E-INVOKE}] \frac{\langle !\iota.m(\overline{v}) \mid (CT, \sigma, \ell, \cdot) \rangle \longrightarrow^* \langle w \mid (CT, \sigma', \ell, \cdot) \rangle}{(I, CT, \sigma) \Downarrow \sigma'}$$

The same notation denotes running a list of invocations \overline{I} in sequence, using the output state from one as the input state from the next. That is, if $\overline{I} = I_1, \dots, I_n$ and $(I_i, CT, \sigma_{i-1}) \Downarrow \sigma_i$ for each $1 \leq i \leq n$, then we write $(\overline{I}, CT, \sigma_0) \Downarrow \sigma_n$.

To type-check an invocation, the expression used in the evaluation must be well-typed in the evaluation environment:

$$[\text{INVOKE}] \frac{\Sigma; \cdot; \ell; \lambda_I \vdash !\iota.m(\overline{v}) : \tau \dashv \lambda_O}{\Sigma \vdash (\iota, m(\overline{v}), \ell)}$$

5.4 Examples Revisited

We now revisit the examples from Section 2 to see how SeRIF detects application vulnerabilities while permitting secure implementations.

Uniswap. The vulnerability (Section 2.1) stems from an unexpected interaction between an exchange, tokens, and a malicious user. While they may all have different integrity, for simplicity, we give the exchange and the tokens the same trusted label T and the user an untrusted label U with $U \not\approx T$.

Anyone can call `sellXForY`, but it computes how much of asset Y to move and transfers tokens, so it must have label $U \gg T; \lambda_0$ for some λ_0 . Similarly, the token’s `transferTo` method modifies high-integrity records, so it needs label $pc \gg T; \lambda'_0$ for some labels pc and λ'_0 .

The `METHOD-OK` rule requires `sellXForY` to type-check with some λ_I where $\lambda_I \Rightarrow T$. Because we sequence two calls to `transferTo`, `LET` requires either $\lambda'_0 \Rightarrow \lambda_I \Rightarrow T$, or a dynamic lock on label T around (at least) the first transfer. These options correspond precisely to the solutions suggested in Section 2.1. Requiring $\lambda'_0 \Rightarrow T$ is a statement that Uniswap expects the tokens not to call untrusted code. A dynamic lock, by contrast, secures the exchange without assuming any particular token behavior and correspondingly allows any value of λ'_0 .

Notably, `transferTo` can type-check with $\lambda'_0 \Rightarrow T$ in either of two ways: it can decline to call unknown code (i.e., remove lines 20 and 21 in Figure 1), or the token itself could acquire a dynamic lock while making the calls. The first option straightforwardly eliminates the vulnerability. By locking T , the second option dynamically prevents reentrant calls during a transfer to either the token or the exchange.

Key–value store. We use the same labeling scheme: the key–value store application gets a trusted label T while the user gets an untrusted label U . Because anyone can call `getOrCompute` but it modifies trusted data, it must have label $U \gg T; \lambda_0$ for some λ_0 . The user-provided computation function is not trusted, so it gets label $pc \gg U; \lambda'_0$ for some labels pc and λ'_0 .

As in the Uniswap example above, `METHOD-OK` requires `getOrCompute` to type-check with some $\lambda_I \Rightarrow T$. Because the user-provided fallback function executes in sequence before another trusted operation, `LET` and `CALL` combine to require either a dynamic lock or $\lambda'_0 \vee U \Rightarrow \lambda_I \Rightarrow T$. This second option, however, is impossible because $U \not\Rightarrow T$.

This forced reliance on a dynamic lock stems from the type system not trusting the user-provided callback to even type-check. In a modified type system that separated trust in the code’s execution from trust that it type-checks, it would be sufficient to require that it type-check with high-integrity and some $\lambda'_0 \Rightarrow T$. This solution would correspond to a static guarantee that the user-provided callback does not invoke `clear` or any other method modifying the store’s internal state.

Town Crier. As described in Section 2.3 and the original paper [64], Town Crier is secure despite using (object) reentrancy, and the type system can verify that. Using the same labels again, we label Town Crier and the trusted service address T and the user U . We can give the functions the following signatures.

```
int request{U ≫ T; T}(params:tU, callback:addressU)
void cancel{U ≫ T; U}(id:intU)
void deliver{T ≫ T; U}(id:intT, data:bytesT)
```

The `request` method—which just records the request parameters and updates a counter—type-checks simply. The `cancel` method type-checks with an endorsement on the condition on line 14 of Figure 3. Type-checking `deliver` relies on TC trusting `SERVICE_ADDR` not to call attackers when receiving money. However, `SERVICE_ADDR` is a hard-coded wallet address with no code that is already trusted to provide data to `deliver`, so the operation sending it money can safely have the signature $T \gg T; T$. These labels allow `deliver` to type-check as written.

6 Formalizing Security Guarantees

We now have the tools needed to formalize reentrancy and security from Section 4.

6.1 Attacker Model

Proving a security guarantee requires a well-defined attacker. As ℓ -reentrancy is parameterized on a label, we also parameterize attackers over what they compromise. We assume that an attacker \mathcal{A} controls some collection of system components, including anything that trusts any combination of those components. For simplicity, we require a label $\ell_{\mathcal{A}}$ representing the combined attacker power and a label ℓ_t representing the minimum honest integrity, where every label is either attacker-control or honest. That is, for all $\ell \in \mathcal{L}$, either $\ell_{\mathcal{A}} \Rightarrow \ell$ or $\ell \Rightarrow \ell_t$, but not both.² We prove that, for *any* such ℓ_t and $\ell_{\mathcal{A}}$, if all code trusted at ℓ_t abides by the static and dynamic locking requirements, the system is ℓ -reentrancy secure whenever $\ell \Rightarrow \ell_t$. This parameterization of the attacker ensures that only someone you trust can damage your security.

Notably, the requiring ℓ_t and $\ell_{\mathcal{A}}$ to exist means that, to guaranteeing security at $\ell_1 \wedge \ell_2$, one or both of ℓ_1 and ℓ_2 must act for ℓ_t , and therefore be honest. In other words, trusting the combined power of two labels is a statement that you believe at least one of those labels is honest, though you may not know which. Combined with trust in $\ell_1 \vee \ell_2$ expressing trust in both ℓ_1 and ℓ_2 , this idea supports modeling complex assumptions like “at least k of n nodes are honest.”

Because reentrancy attacks stem from attacker code performing unexpected operations, we grant attackers considerable power. Specifically, attackers can modify or replace any code that executes with low integrity—that is, any code where $\ell_{\mathcal{A}} \Rightarrow pc$. Allowing attackers to modify high-integrity code executing with a low-integrity pc may seem unrealistic, but experience has shown that code bases contain “gadgets” that attackers can combine to achieve arbitrary functionality [47, 52]. This expansive power conservatively models the ability to exploit such gadgets without modeling the gadgets explicitly.

To model the attacker’s ability to sidestep static security features, we introduce a new term to ignore static lock labels.

$$\begin{aligned}
 e & ::= \dots \mid \text{ignore-locks-in } e \\
 E & ::= \dots \mid \text{ignore-locks-in } E \\
 \text{[E-IGNORELOCKS]} & \frac{}{\langle \text{ignore-locks-in } v \mid \mathcal{C} \rangle \longrightarrow \langle v \mid \mathcal{C} \rangle} \\
 \text{[IGNORELOCKS]} & \frac{\Sigma; \Gamma; pc; \lambda'_I \vdash e : \tau \dashv \lambda'_O}{\Sigma; \Gamma; pc; \lambda_I \vdash \text{ignore-locks-in } e : \tau \dashv \lambda_O}
 \end{aligned}$$

Reasoning explicitly about ill-typed code is challenging, so the formal model requires all code to type-check, but allows low-integrity code to use this new term. Using `ignore-locks-in` may not appear to grant the full power of ignoring the type system. After all, the type system limits the location of method calls and state modifications based on the pc label, which attackers cannot modify. However, low-integrity code can only interact with high-integrity code in three ways: calling high-integrity methods, returning values to high-integrity contexts, or writing to memory that high-integrity code will later read. In each case, the operational semantics includes dynamic checks to ensure memory safety and to ensure that method calls and state modifications are only performed by sufficiently trusted code—exactly what the type system asks.

²Our results hold for any partition of \mathcal{L} into a downward-closed sublattice \mathcal{T} and an upward-closed sublattice \mathcal{A} , letting ℓ be “trusted” if $\ell \in \mathcal{T}$. If \mathcal{T} and \mathcal{A} are complete, this formulation is equivalent with $\ell_t = \bigvee \mathcal{T}$ and $\ell_{\mathcal{A}} = \bigwedge \mathcal{A}$.

Indeed, the only constraint the type system imposes that these dynamic checks do not enforce is the static locking that ignore-locks-in is designed to avoid. Modeling well-typed high-integrity code and unknown attacker code is therefore as simple as demanding that all code type-checks and high-integrity code does not use ignore-locks-in, formalized as follows.

Definition 3 (Lock Compliance). A class table CT complies with locks in ℓ_t -code if, whenever

$$CT(C) = \text{class } C[\ell_C] \text{ extends } D \{ \bar{f} : \bar{\tau}_f ; K ; \bar{M} \}$$

and $\ell_C \Rightarrow \ell_t$, then ignore-locks-in does not appear syntactically in the body of any method $m \in \bar{M}$.

Strong object-level memory isolation, like that in Ethereum, reduces the information security checks of the semantics to type-checking high-integrity code. Forcing dynamic lock checks, however, requires direct support in the system runtime. As such features are uncommon, we model a system where attackers can freely ignore dynamic locks. Specifically, we extend the operational semantics with a second rule for function calls, E-CALLATK, which enables calls to attacker-controlled code without checking dynamic label locks.

$$\text{[E-CALLATK]} \frac{\begin{array}{l} mbody(C, m) = (\ell_m, \bar{x}, \bar{\tau}_a, pc_1 \gg pc_2, e, \tau) \\ \mathcal{M} = \mathcal{M}', \ell'_m \quad \ell'_m \Rightarrow pc_1 \quad \ell_A \Rightarrow pc_2 \\ \Sigma_\sigma \vdash \bar{w} : \bar{\tau}_a \quad e' = e[\bar{x} \mapsto \bar{w}, \text{this} \mapsto \text{new } C(\bar{v})] \end{array}}{\langle \text{new } C(\bar{v}).m(\bar{w}) \mid \mathcal{C} \rangle \longrightarrow \langle \text{return}_\tau(e' \text{ at-pc } pc_2) \mid \mathcal{C}[\mathcal{M}, \ell_m/\mathcal{M}] \rangle}$$

This rule is identical to E-CALL, except instead of checking dynamic locks, it checks that pc_2 is untrusted ($\ell_A \Rightarrow pc_2$).

Interestingly, in systems that require even untrusted calls to check dynamic locks—admitting only E-CALL and not E-CALLATK—trust of $\ell_1 \wedge \ell_2$ can be safe even when neither ℓ_1 nor ℓ_2 is honest. Such systems enforce ℓ_t -reentrancy security whenever CT complies with locks in ℓ_t -code. There can even exist labels ℓ_1 and ℓ_2 where CT does not comply with locks in ℓ_1 -code or ℓ_2 -code, but $\ell_1 \wedge \ell_2 \Rightarrow \ell_t$, meaning ℓ_A cannot be a well-defined label. The proofs in Appendix E consider both system and attacker models.

Attacker-provided code. In addition to having ill-typed code, attackers can tailor their attacks to the specific application. We therefore define security with respect to *any* system with the same high-integrity code. Specifically, we employ a notion of ℓ_t -equivalent code that allows an attacker to add, remove, or replace code whenever $pc \not\Rightarrow \ell_t$.

We formalize the equivalence using erasure on the code in a class table CT . Let $CT|_{\ell_t}$ denote CT , but erasing any class C with low-integrity code ($\ell_C \not\Rightarrow \ell_t$), any method m that executes with low integrity ($pc_2 \not\Rightarrow \ell_t$), and the branches of if statements executing with low integrity ($pc \not\Rightarrow \ell_t$). Two class tables are then ℓ_t -equivalent if they erase to the same thing.

$$CT \approx_\ell CT' \iff CT|_{\ell_t} = CT'|_{\ell_t}$$

Attackers can also freely modify low-integrity locations in the heap, so we define ℓ_t -equivalent heaps using similar erasure. As a heap σ is a partial function from locations to value–type pairs, memory is erased to $\sigma|_{\ell_t}$ simply by erasing mappings with low-integrity types. Formally, $\sigma|_{\ell_t}(\iota) = \sigma(\iota)$ if $\sigma(\iota) = (v, t^\ell)$ with $\ell \Rightarrow \ell_t$, and it is undefined otherwise. As with code, the equivalence follows directly from this erasure:

$$\sigma \approx_\ell \sigma' \iff \sigma|_{\ell_t} = \sigma'|_{\ell_t}.$$

6.2 Noninterference

A typical goal for security in IFC systems, including our core calculus, is *noninterference* [23], which for integrity means untrusted data should not influence trusted data at all. As we argued in Section 3.2,

noninterference is too restrictive, and indeed, endorsement exists to violate it. However, explicit endorsement should be the *only* way to violate noninterference.

To state this, we first need a notion of a class table CT being *endorsement-free* for a label ℓ .

Definition 4 (Endorsement-Free). CT is ℓ -endorsement-free if, for all classes C and methods m such that

$$\begin{aligned} \text{class } C[\ell_C] \text{ extends } D \{ \overline{f} : \overline{\tau_f} ; K ; \overline{M} \} &\in CT \\ \tau m \{ pc_1 \gg pc_2 ; \lambda_o \} (\overline{x} : \overline{\tau_a}) \{ e \} &\in \overline{M} \end{aligned}$$

the following two properties hold. (1) Either $pc_1 \Rightarrow \ell$ or $pc_2 \not\Rightarrow \ell$, and (2) for any subexpression of e of the form `endorse v from ℓ_1 to ℓ_2` , similarly, either $\ell_1 \Rightarrow \ell$ or $\ell_2 \not\Rightarrow \ell$.

Intuitively, this definition says that CT is ℓ -endorsement-free if CT contains no means of endorsing either control flow or data from a label that ℓ does not trust to one that it does.

This condition is sufficient to prove a strong notion of noninterference at ℓ . Because the SeRIF semantics are nondeterministic with respect to selection of location names (E-REF), we use a modified equivalence \simeq_ℓ that allows renaming locations in addition to erasing low-integrity state. See Appendix B for the formal definition of this equivalence.

For partial functions f and f' , we write $f \subseteq f'$ to mean $\text{dom}(f) \subseteq \text{dom}(f')$ and $f(x) = f'(x)$ wherever f is defined.

Theorem 1 (Noninterference). *Let CT be a class table where $\Sigma \vdash CT \text{ ok}$ is ℓ -endorsement-free. For any well-typed heaps σ_1 and σ_2 such that $\Sigma \subseteq \Sigma_{\sigma_i}$ and any invocation I such that $\Sigma \vdash I$ and $(I, CT, \sigma_i) \Downarrow \sigma'_i$, if $\sigma_1 \simeq_\ell \sigma_2$, then $\sigma'_1 \simeq_\ell \sigma'_2$.*

Theorem 1 follows by a complicated induction on the operational semantics, erasing untrusted values in the heap. See Appendix D for details.

Note also that the theorem says nothing about lock compliance, only endorsement freedom. Indeed, reentrancy locks are unnecessary to enforce noninterference.

6.3 Formalizing Reentrancy

Definition 1 in Section 4.1 informally defines ℓ -reentrancy as a trusted computation calling an untrusted one, which then calls a trusted computation before returning. We also noted that the pc label specifies the integrity of the control flow and is therefore ideal for defining reentrancy.

Because SeRIF's semantics has no explicit call stack, it must insert at- pc tracking terms in the only places where the pc label of the currently-executing code can change: conditionals and method calls. The terms surround the body of the condition or method and remain until execution returns to the previous pc label. Nested tracking terms appear precisely when code in one conditional or method body calls a second before returning. We therefore formalize ℓ -reentrancy as three nested at- pc terms where ℓ trusts the label of the first and third, but not the second. As each condition or call may still have pending computation, we allow arbitrary evaluation contexts at each integrity level.

Definition 5 (ℓ -Reentrancy). A statement s is ℓ -reentrant if, for some evaluation contexts E_0, E_1, E_2 ,

$$s = E_0 \left[E_1 \left[E_2 [s' \text{ at-}pc \ pc_3] \text{ at-}pc \ pc_2 \right] \text{ at-}pc \ pc_1 \right]$$

where $pc_1, pc_3 \Rightarrow \ell$ but $pc_2 \not\Rightarrow \ell$.

We say an invocation $I = (\iota, m(\overline{v}), \ell')$ is ℓ -reentrant in σ if $\langle !\iota.m(\overline{v}) \mid (CT, \sigma, \ell', \cdot) \rangle \longrightarrow^* \langle s \mid \mathcal{C} \rangle$ where s is ℓ -reentrant.

With a definition of reentrancy and a formal attacker model, we can formalize the notion of security described in Section 4.2. Recall that “secure reentrancy” meant that any program behavior possible with reentrancy is also possible without reentrancy. Equivalently, state changes made by reentrant executions must be possible using non-reentrant ones.

We describe the properties a program *maintains* using a modified Hoare logic [28]. Because high-integrity code may interact with arbitrary attacker code, we consider all possible invocations with ℓ -equivalent code. Specifically, the high-integrity component of CT maintains a property defined by a predicate pair (P, Q) if, whenever P holds on the input state, Q must hold on the output state.

Definition 6 (Predicate Satisfaction). Given a class table CT , a heap type Σ , and state predicates P and Q , we say that CT *satisfies* (P, Q) at ℓ in Σ , denoted $\Sigma \models_{\ell} \{P\} CT \{Q\}$, if, for any CT' such that $CT \approx_{\ell} CT'$, any well-typed state σ_1 where $\Sigma \subseteq \Sigma_{\sigma_1}$, and any invocation sequence \bar{I} such that $\Sigma_{\sigma_1} \vdash \bar{I}$ and $(\bar{I}, CT', \sigma_1) \Downarrow \sigma_2$, then $P(\sigma_1)$ implies $Q(\sigma_2)$.

To simplify proofs, the definition requires invocations to be well-typed. The requirement does not, however, weaken the security guarantee. In a system like Ethereum without a strong type system, a high-integrity contract would need to examine its arguments to ensure they are well-typed. We assume this facility is built into the runtime.

The predicates P and Q can capture a variety of program properties. A simple example is program invariants—such as Uniswap’s invariant on the product of the token balances—in which case P and Q would be the same. Quantifying over a potentially infinite set of predicates, as the security definition does below, allows for arbitrarily complex properties. For example, requiring a specific high-integrity output state for each possible high-integrity input state would enforce noninterference. A demonstration of interference would demonstrate that one such predicate pair is not satisfied.

Our goal, however, is not to guarantee any specific properties, but to formalize the idea that reentrancy should not introduce new behavior. Definition 6 says nothing about reentrancy. It captures the *entire* set of possible behaviors, including the reentrant ones. Saying that a complete set of behaviors is equivalent to the non-reentrant behaviors requires a definition of non-reentrant behaviors. For that, we simply restrict our previous definition to executions that are not ℓ -reentrant.

Definition 7 (Single-Entry Predicate Satisfaction). Given a class table CT , a heap type Σ , and state predicates P and Q , we say that CT *single-entry satisfies* (P, Q) at ℓ in Σ , denoted $\Sigma \models_{\ell}^1 \{P\} CT \{Q\}$, if CT satisfies (P, Q) at ℓ in Σ when restricted to invocation sequences \bar{I} that are *not* ℓ -reentrant.

These two definitions combine to specify the difference between non-reentrant program behavior and all program behavior. To compare them, note that a program satisfies predicate pair (P, Q) precisely when no behavior violates it. Therefore, if reentrancy can exhibit new behaviors—the program is insecure—there should be a predicate pair that is single-entry satisfied, but not satisfied in general.

Because attackers can arbitrarily modify low-integrity state, *any* changes to low-integrity state are possible without ℓ -reentrancy. We correspondingly restrict our security notion to predicates that are unaffected by low-integrity state.

Definition 8 (ℓ -integrity Predicate). We say a predicate P is ℓ -integrity if, for all pairs of states σ_1 and σ_2 ,

$$\sigma_1 \approx_{\ell} \sigma_2 \implies P(\sigma_1) \Leftrightarrow P(\sigma_2).$$

We now define ℓ -reentrancy security formally.

Definition 9 (Reentrancy Security (formal)). We say a class table CT is ℓ -reentrancy secure in Σ if for all pairs (P, Q) of ℓ -integrity predicates, $\Sigma \models_{\ell}^1 \{P\} CT \{Q\}$ implies $\Sigma \models_{\ell} \{P\} CT \{Q\}$.

Definition 9 is the core security definition SeRIF enforces.

Theorem 2. *For any label ℓ , class table CT , and heap type Σ , if $\ell \Rightarrow \ell_t$ and $\Sigma \vdash CT$ ok complies with locks in ℓ_t -code, then CT is ℓ -reentrancy secure in Σ .*

Theorem 2 follows from two core results. First, all reentrancy allowed by SeRIF is tail reentrancy. That is, if an invocation passes through an ℓ -reentrant state, then the outer high-integrity call (E_1 at-pc pc_1 in Definition 5) must be in tail position.

Theorem 3. *For a label ℓ , class table CT , and well-typed heap σ_1 , if $\ell \Rightarrow \ell_t$ and $\Sigma_{\sigma_1} \vdash CT$ ok complies with locks in ℓ_t -code, then for any invocation I and heap σ_2 where $\Sigma_{\sigma_1} \vdash I$ and $(I, CT, \sigma_1) \Downarrow \sigma_2$, all ℓ -reentrant states in the execution are ℓ -tail-reentrant.*

Proof Sketch. The theorem follows from two facts. First, if a statement s steps to a call to a method that grants integrity ℓ , then s cannot maintain a lock on ℓ . Second, any statement executing with integrity ℓ must maintain a lock on ℓ (either statically or dynamically) unless it is in tail position. We provide a complete proof in Appendix E.1. \square

Once we know that all reentrant executions are tail-reentrant, we need only show that tail reentrancy is secure. The following theorem formalizes this idea by proving that, if all ℓ -reentrant states are ℓ -tail-reentrant, then single-entry predicate satisfaction translates to predicate satisfaction.

Theorem 4. *Let CT be a class table, σ_1 and σ_2 be well-typed heaps, and I be an invocation such that $(I, CT, \sigma_1) \Downarrow \sigma_2$ where all ℓ -reentrant states are ℓ -tail-reentrant. For any ℓ -integrity predicates P and Q , if $\Sigma_{\sigma_1} \models_{\ell}^1 \{P\} CT \{Q\}$ and $P(\sigma_1)$, then $Q(\sigma_2)$.*

Proof Sketch. Examine the execution of I and build a CT' and \bar{I} that produce a ℓ -equivalent final state with no reentrancy. Whenever a high-integrity environment transitions to a low-integrity one in CT , replace the low-integrity code in CT' with code that returns the same value as a hard-coded constant and makes no calls to high-integrity code. For each call from a low-integrity environment to a high-integrity method, add an invocation to \bar{I} that makes the same call with the same arguments. Add additional invocations between each high-integrity call to update the low-integrity state to match the low-integrity state in the original execution when the call occurred. The result is clearly a non-reentrant set of executions. Because all ℓ -reentrant states are ℓ -tail-reentrant in the original execution, placing a reentrant call sequentially after the call it was originally inside produces the same result.

Since the start and end states σ'_1 and σ'_2 of this new execution are ℓ -equivalent to σ_1 and σ_2 and $\Sigma_{\sigma_1} \models_{\ell}^1 \{P\} CT \{Q\}$,

$$P(\sigma_1) \iff P(\sigma'_1) \implies Q(\sigma'_2) \iff Q(\sigma_2).$$

See Appendix E.2 for details. \square

From here, we have enough to prove our desired result.

Proof of Theorem 2. For a class table CT' , invocation I , and heaps σ_1 and σ_2 such that $CT \approx_{\ell} CT'$ and $(I, CT', \sigma_1) \Downarrow \sigma_2$, Theorem 3 says all ℓ -reentrant states are ℓ -tail-reentrant. For ℓ -integrity predicates P and Q such that $\Sigma_{\sigma_1} \models_{\ell}^1 \{P\} CT \{Q\}$, Theorem 4 says that if $P(\sigma_1)$ then $Q(\sigma_2)$, which is precisely the definition of $\Sigma_{\sigma_1} \models_{\ell} \{P\} CT \{Q\}$. \square

Application	LoC	type-check time (s)	necessary annotations
Uniswap 1	57	4.1	11
Uniswap 2	49	4.0	9
Uniswap 3*	53	4.3	9
Town Crier 1	133	6.3	17
Town Crier 2*	133	6.5	17
Town Crier 3*	133	6.4	17
KV Store 1	38	2.1	10
KV Store 2*	35	2.0	9
Multi-DAO 1	38	3.5	8
Multi-DAO 2	36	3.3	7
Multi-DAO 3*	36	3.3	7

Table 1. Evaluation of SeRIF type checker. Asterisks indicate vulnerable implementations.

7 Implementation

We implemented a type checker for SeRIF in 4,200 lines of Java, using JFlex [31] and CUP [29]. We employ the SHErrLoc constraint solver [63] to analyze information flow constraints, infer missing integrity labels, and identify likely error locations.

We ran the type checker on four examples: the three from Section 2, but without simplifying Town Crier, and one we call Multi-DAO. Multi-DAO is a multi-contract version of the vulnerable portion of Ethereum’s DAO contract [46]. It is one application split across multiple contracts that synchronize on each transaction. This structure allows for the DAO’s original reentrancy vulnerability, as well as a second attack where the attacker reenters the application by leaving one contract and entering another before they synchronize. By definition, this attack is not object reentrancy, but as long as the Multi-DAO contracts trust each other, it is ℓ -reentrancy. As with the original DAO, the exploits can be patched either with dynamic locks or by performing local state changes *and inter-contract synchronization operations* before external calls.

For each example, the type checker correctly identified vulnerabilities in the initial versions presented in Section 2. It also accepted as secure patched implementations following the suggested fixes, both with and without dynamic locks.

Developer Overhead. Table 1 presents several metrics for developer overhead. As each example application is designed to distill complex security logic into minimal code, the examples are all relatively short—ranging from 35 to 133 lines of code. On these examples, the type checker is able to run in a few seconds on a consumer desktop from 2015 with an Intel i7-4790 CPU. Because the type system and the associated guarantees are compositional, modules can be checked independently, so running time should scale well as the code grows.

Another important practical concern is the annotation burden of adding information flow labels to the code. Labels on classes, fields, methods, and data endorsements are necessary to define the security of a program. Though SeRIF requires explicit labels elsewhere to ease formal reasoning, many of these—such as the pc labels on if statements—are simple to infer. Considering only the labels with no obvious inference mechanism, we found that 13% of the lines required explicit labels in Town Crier. The other examples required more annotations per line as their distilled nature led to more function declarations and explicit endorsements. As even Town Crier is a short application with complex security concerns, we expect many applications would have lower annotation burdens.

Finally, SHErrLoc is capable of localizing errors, helping guide development. To see its utility, we look

at the Uniswap example in more detail. As in Section 5.4, we use two labels: U and T . Recall that the exchange must either utilize a lock or state its assumption that the token will not call untrusted code. The following signature for the token’s `transferTo` method makes the assumption explicit, where H is a token holder class.

```
boolT transferTo{T≫T;T}(from:HT, to:HT, amount:intT)
```

To model the alert functions in H being unknown code from unknown sources, the interface can state the following entirely-untrusted signatures.

```
void alertSend{U≫U;U}(to:HU, amount:intU)
void alertReceive{U≫U;U}(from:HU, amount:intU)
```

With these signatures, the calls to the alert functions in `transferTo` on lines 20 and 21 of Figure 1 cannot type-check without a dynamic lock. `SHErrLoc` helpfully identifies line 21 as the most likely error. The type checker correctly identifies the program as secure if we either wrap both alerts in a dynamic lock or remove them entirely.

8 Related Work

We now discuss other work on reentrancy security, secure smart contracts, and information flow control.

Formal Reentrancy Security. Grossman et al. [26] define Effectively Callback-Free (ECF) executions, the only other formal definition of reentrancy security of which we are aware. An ECF execution is one where the operations can be reordered to produce the same result without callbacks (reentrancy). Their definition is object-based, which we have argued fails to separate the security specification from the program design, and they focus on dynamic analysis of individual executions.

Albert et al. [4] present a static analysis tool to check if code produces only ECF executions. The authors advertise the tool as providing modular guarantees, but define “modular” to mean that a contract remains secure against any possible outside code. Our approach provides the same guarantees when applied to a single program with no assumptions on others, but also enables developers to safely compose independently-checked modules by stating assumptions on each other’s behavior. Furthermore, Albert et al.’s analysis relies on an SMT solver, limiting its scalability. In comparison, `SeRIF` only relies on checking acts-for relationships of information flow labels.

We previously proposed the intuition of using information flow control with a mix of static and dynamic locks to enforce ℓ -reentrancy [12]. In this work we add a core calculus with static and dynamic semantics, formal definitions, proofs, and an evaluation.

Reentrancy-aware Languages. Several languages—all smart-contract oriented—attempt to guard against reentrancy using a variety of techniques.

Scilla [51] constrains programming style by removing the call-and-return model of contract interaction. Instead, it queues requests and executes them when the caller completes. While this structure makes object-level reentrancy difficult, it prevents contracts from using the return values from remote calls. Moreover, by allowing multiple unconstrained requests, it fails to detect or eliminate bugs like Uniswap (see Section 2.1).

Obsidian [14] and Flint [50] ease reasoning about contract behavior using typestate. Obsidian includes a dynamic check that prevents (object) reentrancy entirely, while Flint has no such check. Both languages and Move [9] have a notion of linear assets that cannot be created or destroyed. Asset linearity prevents attacks like the DAO, but fails to address the challenges of Uniswap. The errant send in Uniswap does not create or destroy tokens; it merely sends the wrong number because it the invariant it relies on is broken.

Nomos [17] enforces security using resource-aware session types. Since linearity of session types is insufficient to eliminate reentrancy, it uses the resources tracked by the session types to prevent attackers from acquiring permission to call an in-use contract—again, eliminating all (object) reentrancy.

Smart Contract Analysis Tools. There are many static analysis tools for blockchain smart contracts. Some tools operate as unsound best-effort bug finding tools. OYENTE [36] searches for anti-patterns in code, TEETHER [33] automatically generates exploits based on commonly-exploitable operations, and Ethainter [11] uses information flow taint analysis to attempt to locate a predefined set of security concerns, such as tainted owner variables and access to self-destruct.

Other tools use formal analysis techniques to soundly analyze contracts. Bhargavan et al. [7] prove functional correctness through translation to F^* . MAIAN [41] and ETHBMC [21] prove security against specific classes of vulnerabilities using symbolic execution and bounded model checking, respectively. EtherTrust [24] allows developers to specify program properties as Horn clauses and verify them using a formal semantics for EVM [25]. SOLYTHESIS [34] combines static and dynamic mechanisms. It statically determines what checks are necessary for correctness and compiles them into run-time checks.

These tools are valuable for securing smart contracts, but they all analyze individual contracts, and their analyses often fail to compose. As a result, they are unable to verify security of applications like Uniswap that span multiple contracts.

Information Flow Control. Several distributed and decentralized systems enforce security using IFC. Fabric [35] is a system and language for building distributed systems that allows secure data and code sharing between nodes despite mutual distrust. DStar [62] uses run-time tracking at the OS level to control information flow in a distributed system. These previous systems have the same limitation of information flow systems that is described in Section 1: they do not defend against reentrancy attacks. The IFC-based instruction set of Zagieboylo et al. [59] restricts endorsement of pc labels using a purely dynamic mechanism that appears to prevent all ℓ -reentrancy. However, this property is neither stated nor proved.

9 Conclusion

Despite decades of work on techniques for making software more secure and trustworthy, recent smart contract bugs have vividly shown that avoiding critical security vulnerabilities can be difficult even in very short programs. The essential challenge is composition of code with complex control flow across trust boundaries. Prior static information flow analyses provide compositional guarantees, but are missing a key ingredient: security against reentrant executions. Smart contracts have produced the most salient reentrancy vulnerabilities to date due to their structure of interacting service in different trust domains. As more applications adopt distributed service-oriented architectures mirroring this design, we expect reentrancy to become more of a concern elsewhere.

This paper provides a flexible general-purpose security definition that permits secure forms of reentrancy and a fine-grained static mechanism to reason about reentrancy security. We presented SeRIF, a core calculus that combines static and dynamic locking to provably enforce reentrancy security in addition to providing standard information flow assurances. We further showed that SeRIF is expressive enough to implement and analyze various challenging examples. SeRIF’s lightweight, inferable annotations support an independently-useful verification process while complementing other verification methods.

We hope these foundational results will aid the development of practical secure languages. To ensure usability, languages will need to infer labels wherever possible and use sensible defaults in many other areas. They might further require polymorphic, finer-grained locks that we believe can fit into the structure of a distributive lattice. Finally, while we focused entirely on single-threaded reentrancy, concurrency is common in real-world languages and applications. The relationship between reentrancy and concurrency controls/consistency models is unclear and, we believe, a promising area for future work.

Acknowledgments

We would first like to thank our anonymous reviewers for their thoughtful comments and suggestions. Additional thanks to Tom Magrino for help clarifying and explaining earlier versions of this work, and to Rachit Nigam, Rolph Recto, and Drew Zagieboylo for help editing.

This work was funded in part by a National Defense Science and Engineering Graduate (NDSEG) Fellowship, NSF grants 1704615 and 1704788, and a gift from Ripple. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and may not reflect those of these sponsors.

References

- [1] CVE-2014-1772. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1772>, 29 Jan. 2014. Accessed March 2021.
- [2] CVE-2018-8174. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-8174>, 14 Mar. 2018. Accessed March 2021.
- [3] CWE-1265: Unintended reentrant invocation of non-reentrant code via nested calls. <https://cwe.mitre.org/data/definitions/1265.html>, 20 Dec. 2018. Accessed March 2021.
- [4] E. Albert, S. Grossman, N. Rinetzky, C. Rodríguez-Núñez, A. Rubio, and M. Sagiv. Taming callbacks for smart contract modularity. *Proc. ACM on Programming Languages*, 4(OOPSLA), Nov. 2020. doi: 10.1145/3428277.
- [5] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing mobile code securely with information flow control. In *IEEE Symp. on Security and Privacy*, pages 191–205, May 2012. URL <http://www.cs.cornell.edu/andru/papers/mobile.html>.
- [6] O. Arden, J. Liu, and A. C. Myers. Flow-limited authorization. In *28th IEEE Computer Security Foundations Symp. (CSF)*, pages 569–583, July 2015. URL <http://www.cs.cornell.edu/andru/papers/flam>.
- [7] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, et al. Formal verification of smart contracts: Short paper. In *11th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 91–96, Oct. 2016. doi: 10.1145/2993600.2993611.
- [8] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, Apr. 1977. URL <https://ban.ai/multics/doc/a039324.pdf>. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).
- [9] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, Rain, D. Russi, S. Sezer, T. Zakian, and R. Zhou. Move: A language with programmable resources. <https://developers.diem.com/docs/technical-papers/move-paper/>, May 2020. Accessed March 2021.
- [10] L. Breidenbach, P. Daian, A. Juels, and E. G. Sirer. An in-depth look at the parity multisig bug. <https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/>, 22 July 2017. Accessed March 2021.

- [11] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *41st ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, page 454–469, June 2020. doi: 10.1145/3385412.3385990.
- [12] E. Cecchetti, S. Yao, H. Ni, and A. C. Myers. Securing smart contracts with information flow. In *3rd Int’l Symp. on Foundations and Applications of Blockchain (FAB)*, Apr. 2020.
- [13] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2009. URL <http://dl.acm.org/citation.cfm?id=1542476.1542483>.
- [14] M. Coblenz, R. Oei, T. Etzel, P. Koronkevich, M. Baker, Y. Bloem, B. A. Myers, J. Sunshine, and J. Aldrich. Obsidian: Typestate and assets for safer blockchain programming. *ACM Trans. on Programming Languages and Systems*, 42(3), Nov. 2020. doi: 10.1145/3417516. URL <https://doi.org/10.1145/3417516>.
- [15] ConsenSys Diligence. Uniswap audit. <https://github.com/ConsenSys/Uniswap-audit-report-2018-12#31-liq> Jan. 2019. Accessed March 2021.
- [16] P. Daian. Analysis of the DAO exploit. <https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/> 18 June 2016. Accessed March 2021.
- [17] A. Das, S. Balzer, J. Hoffmann, F. Pfenning, and I. Santurkar. Resource-aware session types for digital contracts. In *34th IEEE Computer Security Foundations Symp. (CSF)*. IEEE, 2019.
- [18] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.
- [19] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *20th ACM Symp. on Operating System Principles (SOSP)*, Oct. 2005. URL <http://dl.acm.org/citation.cfm?id=1095813>.
- [20] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhorkar, S. Han, P. Vines, and E. X. Wu. Collaborative verification of information flow for a high-assurance app store. In *21st ACM Conf. on Computer and Communications Security (CCS)*, pages 1092–1104, Nov. 2014.
- [21] J. Frank, C. Aschermann, and T. Holz. ETHBMC: A bounded model checker for smart contracts. In *29th USENIX Security Symp.*, Aug. 2020. URL <https://www.usenix.org/conference/usenixsecurity20/presentation/frank>.
- [22] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 47–60. USENIX Association, 2012. URL <http://dl.acm.org/citation.cfm?id=2387880.2387886>.
- [23] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982. URL <https://ieeexplore.ieee.org/document/6234468>.

- [24] I. Grishchenko, M. Maffei, and C. Schneidewind. Foundations and tools for the static analysis of Ethereum smart contracts. In *International Conference on Computer Aided Verification (CAV)*, pages 51–78. Springer, 2018.
- [25] I. Grishchenko, M. Maffei, and C. Schneidewind. A semantic framework for the security analysis of Ethereum smart contracts. In *Int’l Conf. on Principles of Security and Trust (POST)*, pages 243–269. Springer, 2018.
- [26] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM on Programming Languages*, 2(POPL):1–28, Dec. 2017. doi: 10.1145/3158136.
- [27] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *25th IEEE Computer Security Foundations Symp. (CSF)*, June 2012.
- [28] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.
- [29] S. Hudson, F. Flannery, C. S. Ananian, and M. Petter. CUP 0.11b: Construction of Useful Parsers. Software release, <http://www2.cs.tum.edu/projects/cup>, June 2014. URL <http://www2.cs.tum.edu/projects/cup>.
- [30] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. on Programming Languages and Systems*, 23(3):396–450, 2001.
- [31] G. Klein, S. Rowe, and R. Decamp. JFlex 1.8.2. Software release, <https://jflex.de>, May 2020. URL <https://jflex.de>.
- [32] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *21st ACM Symp. on Operating System Principles (SOSP)*, 2007. URL <http://dl.acm.org/citation.cfm?id=1294293>.
- [33] J. Krupp and C. Rossow. TEETHER: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symp.*, Aug. 2018.
- [34] A. Li, J. A. Choi, and F. Long. Securing smart contract with runtime validation. In *41st ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 438–453, June 2020. doi: 10.1145/3385412.3385982.
- [35] J. Liu, O. Arden, M. D. George, and A. C. Myers. Fabric: Building open distributed systems securely by construction. *J. Computer Security*, 25(4–5):319–321, May 2017. doi: 10.3233/JCS-0559. URL <http://www.cs.cornell.edu/andru/papers/jfabric>.
- [36] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *ACM Conf. on Computer and Communications Security (CCS)*, pages 254–269, 2016. doi: 10.1145/2976749.2978309. URL <http://doi.acm.org/10.1145/2976749.2978309>.
- [37] T. Magrino, J. Liu, O. Arden, C. Isradisaikul, and A. C. Myers. Jif 3.5: Java information flow. Software release, <https://www.cs.cornell.edu/jif>, June 2016. URL <https://www.cs.cornell.edu/jif>.
- [38] L. A. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symp. on Security and Privacy*, May 2010. URL <http://dl.acm.org/citation.cfm?id=1849992>.

- [39] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *IEEE Symp. on Security and Privacy*, pages 186–197, May 1998. URL <http://www.cs.cornell.edu/andru/papers/sp98/sp98.pdf>.
- [40] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, Oct. 2000. URL <http://www.cs.cornell.edu/andru/papers/iflow-tosem.pdf>.
- [41] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 653–663, Dec. 2018. doi: 10.1145/3274694.3274743.
- [42] Oracle Corporation. Java SE version 15 API specification. `java.util.Map#computeIfAbsent`. <https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Map.html#computeIfAbsent>. Sept. 2020. Accessed March 2021.
- [43] Parity Technologies. A postmortem on the parity multi-sig library self-destruct. <https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/>, 15 Nov. 2017. Accessed March 2021.
- [44] PeckShield. Uniswap/Lendf.Me hacks: Root cause and loss analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc0>. Apr. 2020. Accessed March 2021.
- [45] B. C. Pierce. *Types and programming languages*. MIT press, 2002.
- [46] N. Popper. A hacking of more than \$50 million dashes hopes in the world of virtual currency. *The New York Times*, 17 June 2016.
- [47] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur. (TISSEC)*, 15(1), Mar. 2012. URL <https://doi.org/10.1145/2133375.2133377>.
- [48] Rust 2020. The Rust standard library, version 1.48.0. `Enum std::collections::hash_map::Entry.or_insert_with`. https://doc.rust-lang.org/std/collections/hash_map/enum. Nov. 2020. Accessed March 2021.
- [49] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003. URL <http://www.cs.cornell.edu/andru/papers/jsac/sm-jsac03.pdf>.
- [50] F. Schrans, S. Eisenbach, and S. Drossopoulou. Writing safe smart contracts in Flint. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, pages 218–219, 2018.
- [51] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao. Safer smart contract programming with Scilla. *Proc. ACM on Programming Languages*, 3(OOPSLA):1–30, Oct. 2019.
- [52] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM Conf. on Computer and Communications Security (CCS)*, page 552–561, Oct. 2007. doi: 10.1145/1315245.1315313.

- [53] J. Siek and W. Taha. Gradual typing for objects. In *21st European Conf. on Object-Oriented Programming*, pages 2–27, July 2007.
- [54] Solidity. Solidity documentation. Release 0.7.5. <https://docs.soliditylang.org/en/v0.7.5/>, Nov. 18 2020. Accessed December 2020.
- [55] Solidity. Solidity security considerations. <https://solidity.readthedocs.io/en/latest/security-considerations.html>, 2021. Accessed March 2021.
- [56] The Open Group. SOA standards. <https://publications.opengroup.org/standards/soa>. Accessed December 2020.
- [57] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.
- [58] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *39th ACM Symp. on Principles of Programming Languages (POPL)*, pages 85–96, 2012.
- [59] D. Zagieboylo, G. E. Suh, and A. C. Myers. Using information flow to design an ISA that controls timing channels. In *32nd IEEE Computer Security Foundations Symp. (CSF)*, June 2019. URL <https://www.cs.cornell.edu/andru/papers/hyperisa>.
- [60] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Trans. on Computer Systems*, 20(3):283–328, Aug. 2002. ISSN 0734-2071. doi: 10.1145/566340.566343. URL <http://doi.acm.org/10.1145/566340.566343>.
- [61] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006. URL <http://dl.acm.org/citation.cfm?id=2018419>.
- [62] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *5th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 293–308, 2008. URL <http://dl.acm.org/citation.cfm?id=1387610>.
- [63] D. Zhang, A. C. Myers, D. Vytiniotis, and S. Peyton Jones. SHerrLoc: A static holistic error locator. *ACM Trans. on Programming Languages and Systems*, 39(4):18, Aug. 2017. URL <http://dl.acm.org/citation.cfm?id=3121137>.
- [64] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town Crier: An authenticated data feed for smart contracts. In *23rd ACM Conf. on Computer and Communications Security (CCS)*, CCS ’16, pages 270–282, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978326. URL <https://eprint.iacr.org/2016/168.pdf>.
- [65] L. Zheng and A. C. Myers. End-to-end availability policies and noninterference. In *18th IEEE Computer Security Foundations Workshop (CSFW)*, pages 272–286, June 2005. URL <http://www.cs.cornell.edu/andru/papers/avail.pdf>.
- [66] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *IEEE Symp. on Security and Privacy*, pages 236–250, May 2003. URL <http://www.cs.cornell.edu/andru/papers/sp03.pdf>.

A Full SeRIF Rules

The full operational semantics for SeRIF are given in Figure 9 and the full typing rules are given in Figures 10 and 11.

B Location–Name Isomorphism

The E-REF operational semantic rule allows for selection of any unmapped location name when creating a new location. This makes the SeRIF operational semantics nondeterministic in its choice of location names. However, this is the only source of nondeterminism in the semantics. That is, for any pair of statement-heap pairs that are equivalent up to location names, if one steps, then the other steps and the results are again equivalent up to location names.

To reason about these differences, we define an equivalence relation that relates statements and heaps that differ only in their location names. Formally, we define a location name permutation θ as an injective map from locations to locations. We extend it to values by permuting location names, recursively permuting constructor arguments of objects, and leaving other values unmodified. We further extend it to statements by recursively applying to each sub-statement and to heaps as follows.

$$\theta(\sigma)(\iota) \triangleq (\theta(v), \tau) \text{ where } \sigma(\theta^{-1}(\iota)) = (v, \tau)$$

This permutation supports the requisite equivalence relation.

Definition 10 (Location–name isomorphism). Statements s_1 and s_2 are *location–name isomorphic*, denoted $s_1 \simeq s_2$, if there exists some θ such that $s_1 = \theta(s_2)$. Similarly, for heaps σ_1 and σ_2 , $\sigma_1 \simeq \sigma_2 \iff \exists \theta. \sigma_1 = \theta(\sigma_2)$.

We write $(s_1, \sigma_1) \simeq (s_2, \sigma_2)$ to mean there is a θ such that $(s_1, \sigma_1) = (\theta(s_2), \theta(\sigma_2))$ and similarly for $(s_1, \mathcal{C}_1) \simeq (s_2, \mathcal{C}_2)$.

This definition is sufficient to state and prove the important property that the SeRIF semantics is deterministic up to location–name isomorphism.

Theorem 5. For any s_1, s'_1 , and s_2 and any $\mathcal{C}_1, \mathcal{C}'_1$ and \mathcal{C}_2 , if $(s_1, \mathcal{C}_1) \simeq (s_2, \mathcal{C}_2)$ and $\langle s_1 \mid \mathcal{C}_1 \rangle \longrightarrow \langle s'_1 \mid \mathcal{C}'_1 \rangle$, then there exists s'_2 and \mathcal{C}'_2 such that $\langle s_2 \mid \mathcal{C}_2 \rangle \longrightarrow \langle s'_2 \mid \mathcal{C}'_2 \rangle$, and for all such s'_2 and \mathcal{C}'_2 , $(s'_1, \mathcal{C}'_1) \simeq (s'_2, \mathcal{C}'_2)$.

Proof. By induction on the operational semantics. We take the permutation to be defined only mapping location names between σ_1 and σ_2 and extend it on uses of E-REF (or inductively with E-EVAL). \square

Finally, for use in the noninterference theorem (Theorem 1), we combine location–name isomorphism with ℓ_t -equivalence.

Definition 11 (Location–name ℓ_t -isomorphism). Two states σ_1 and σ_2 are *location–name ℓ_t -isomorphic*, denoted $\sigma_1 \simeq_{\ell_t} \sigma_2$, if there exists a θ such that $\sigma_1|_{\ell_t} = \theta(\sigma_2)|_{\ell_t}$.

C Preservation and Progress

We now prove preservation and progress theorems for SeRIF.

Because SeRIF is stateful, the type preservation theorem includes preservation of both the statement and the heap.

Theorem 6 (Type Preservation). *If*

[E-EVAL]	$\frac{\langle s \mid \mathcal{C} \rangle \longrightarrow \langle s' \mid \mathcal{C}' \rangle}{\langle E[s] \mid \mathcal{C} \rangle \longrightarrow \langle E[s'] \mid \mathcal{C}' \rangle}$
[E-LET]	$\overline{\langle \text{let } x = v \text{ in } e \mid \mathcal{C} \rangle \longrightarrow \langle e[x \mapsto v] \mid \mathcal{C} \rangle}$
[E-IFT]	$\overline{\langle \text{if}\{pc\} \text{ true then } e_1 \text{ else } e_2 \mid \mathcal{C} \rangle \longrightarrow \langle e_1 \text{ at-pc } pc \mid \mathcal{C} \rangle}$
[E-IFF]	$\overline{\langle \text{if}\{pc\} \text{ false then } e_1 \text{ else } e_2 \mid \mathcal{C} \rangle \longrightarrow \langle e_2 \text{ at-pc } pc \mid \mathcal{C} \rangle}$
[E-ATPC]	$\overline{\langle v \text{ at-pc } pc \mid \mathcal{C} \rangle \longrightarrow \langle v \mid \mathcal{C} \rangle}$
[E-REF]	$\frac{\iota \notin \text{dom}(\sigma) \quad \Sigma_\sigma \vdash v : \tau \quad \mathcal{M} = \mathcal{M}', \ell_m \quad \ell_m \triangleleft \tau}{\langle \text{ref } v \ \tau \mid \mathcal{C} \rangle \longrightarrow \langle \iota \mid \mathcal{C}[\sigma[\iota \mapsto (v, \tau)]]/\sigma \rangle}$
[E-DEREF]	$\frac{\sigma(\iota) = (v, \tau)}{\langle !\iota \mid \mathcal{C} \rangle \longrightarrow \langle v \mid \mathcal{C} \rangle}$
[E-ASSIGN]	$\frac{\Sigma_\sigma(\iota) = \tau \quad \Sigma_\sigma \vdash v : \tau \quad \mathcal{M} = \mathcal{M}', \ell_m \quad \ell_m \triangleleft \tau}{\langle \iota := v \mid \mathcal{C} \rangle \longrightarrow \langle () \mid \mathcal{C}[\sigma[\iota \mapsto (v, \tau)]]/\sigma \rangle}$
[E-CAST]	$\frac{D <: C}{\langle (C)(\text{new } D(\bar{v})) \mid \mathcal{C} \rangle \longrightarrow \langle \text{new } D(\bar{v}) \mid \mathcal{C} \rangle}$
[E-FIELD]	$\overline{\langle \text{new } C(\bar{v}).f_i \mid \mathcal{C} \rangle \longrightarrow \langle v_i \mid \mathcal{C} \rangle}$
[E-CALL]	$\frac{\begin{array}{l} \text{mbody}(C, m) = (\ell_m, \bar{x}, \bar{\tau}_a, pc_1 \gg pc_2, e, \tau) \\ \mathcal{M} = \mathcal{M}', \ell'_m \quad \ell'_m \Rightarrow pc_1 \quad \bigwedge_{\ell \in L} (pc_1 \Rightarrow pc_2 \vee \ell) \\ \Sigma_\sigma \vdash \bar{w} : \bar{\tau}_a \quad e' = e[\bar{x} \mapsto \bar{w}, \text{this} \mapsto \text{new } C(\bar{v})] \end{array}}{\langle \text{new } C(\bar{v}).m(\bar{w}) \mid \mathcal{C} \rangle \longrightarrow \langle \text{return}_\tau (e' \text{ at-pc } pc_2) \mid \mathcal{C}[\mathcal{M}, \ell_m/\mathcal{M}] \rangle}$
[E-CALLATK]	$\frac{\begin{array}{l} \text{mbody}(C, m) = (\ell_m, \bar{x}, \bar{\tau}_a, pc_1 \gg pc_2, e, \tau) \\ \mathcal{M} = \mathcal{M}', \ell'_m \quad \ell'_m \Rightarrow pc_1 \quad \ell_A \Rightarrow pc_2 \\ \Sigma_\sigma \vdash \bar{w} : \bar{\tau}_a \quad e' = e[\bar{x} \mapsto \bar{w}, \text{this} \mapsto \text{new } C(\bar{v})] \end{array}}{\langle \text{new } C(\bar{v}).m(\bar{w}) \mid \mathcal{C} \rangle \longrightarrow \langle \text{return}_\tau (e' \text{ at-pc } pc_2) \mid \mathcal{C}[\mathcal{M}, \ell_m/\mathcal{M}] \rangle}$
[E-RETURN]	$\frac{\Sigma_\sigma \vdash v : \tau \quad \mathcal{M} = \mathcal{M}', \ell_m}{\langle \text{return}_\tau v \mid \mathcal{C} \rangle \longrightarrow \langle v \mid \mathcal{C}[\mathcal{M}'/\mathcal{M}] \rangle}$
[E-LOCK]	$\overline{\langle \text{lock } \ell \text{ in } e \mid \mathcal{C} \rangle \longrightarrow \langle e \text{ with-lock } \ell \mid \mathcal{C}[L, \ell/L] \rangle}$
[E-UNLOCK]	$\frac{L = L', \ell}{\langle v \text{ with-lock } \ell \mid \mathcal{C} \rangle \longrightarrow \langle v \mid \mathcal{C}[L'/L] \rangle}$
[E-ENDORSE]	$\overline{\langle \text{endorse } v \text{ from } \ell' \text{ to } \ell \mid \mathcal{C} \rangle \longrightarrow \langle v \mid \mathcal{C} \rangle}$
[E-IGNORELOCKS]	$\overline{\langle \text{ignore-locks-in } v \mid \mathcal{C} \rangle \longrightarrow \langle v \mid \mathcal{C} \rangle}$

Figure 9. Full small-step operational semantics for SeRIF.

Value Typing			
$[\text{VAR}] \frac{\Gamma(x) = \tau}{\Sigma; \Gamma \vdash x : \tau}$	$[\text{UNIT}] \frac{}{\Sigma; \Gamma \vdash () : \text{unit}^\ell}$	$[\text{TRUE}] \frac{}{\Sigma; \Gamma \vdash \text{true} : \text{bool}^\ell}$	$[\text{FALSE}] \frac{}{\Sigma; \Gamma \vdash \text{false} : \text{bool}^\ell}$
$[\text{NEW}] \frac{\text{fields}(C) = \bar{f} : \bar{\tau} \quad \Sigma; \Gamma \vdash \bar{v} : \bar{\tau}}{\Sigma; \Gamma \vdash \text{new } C(\bar{v}) : C^\ell}$	$[\text{LOC}] \frac{\Sigma(\iota) = \tau}{\Sigma; \Gamma \vdash \iota : (\text{ref } \tau)^\ell}$	$[\text{NULL}] \frac{}{\Sigma; \Gamma \vdash \text{null} : (\text{ref } \tau)^\ell}$	
$[\text{SUBTYPEV}] \frac{\Sigma; \Gamma \vdash v : \tau' \quad \tau' <: \tau}{\Sigma; \Gamma \vdash v : \tau}$			
Core Expression Typing			
$[\text{VAL}] \frac{\Sigma; \Gamma \vdash v : \tau}{\Sigma; \Gamma; pc; \lambda_1 \vdash v : \tau \dashv \lambda_0}$	$[\text{ENDORSE}] \frac{\Sigma; \Gamma \vdash v : t^{\ell'}}{\Sigma; \Gamma; \ell; \lambda_1 \vdash \text{endorse } v \text{ from } \ell' \text{ to } \ell : t^\ell \dashv \lambda_0}$		
$[\text{CAST}] \frac{\Sigma; \Gamma \vdash v : D^\ell}{\Sigma; \Gamma; pc; \lambda_1 \vdash (C)v : C^\ell \dashv \lambda_0}$	$[\text{FIELD}] \frac{\Sigma; \Gamma \vdash v : C^\ell \quad \text{fields}(C) = \bar{f} : \bar{\tau} \quad \tau_i <: \tau \quad \ell \triangleleft \tau}{\Sigma; \Gamma; pc; \lambda_1 \vdash v.f_i : \tau \dashv \lambda_0}$	$[\text{CALL}] \frac{\text{mtype}(C, m) = \bar{\tau}_a \xrightarrow{pc_1 \gg pc_2; \lambda_0} \tau_0 \quad \Sigma; \Gamma \vdash v : C^\ell \quad \Sigma; \Gamma \vdash \bar{v}_a : \bar{\tau}_a \quad \ell \Rightarrow pc_1 \quad pc_1 \Rightarrow pc_2 \vee \lambda_1 \quad \tau_0 <: \tau \quad pc_2 \vee \ell \triangleleft \tau}{\Sigma; \Gamma; pc_1; \lambda_1 \vdash v.m(\bar{v}_a) : \tau \dashv \lambda_0 \vee pc_2}$	
$[\text{IF}] \frac{\Sigma; \Gamma \vdash v : \text{bool}^\ell \quad \ell \Rightarrow pc \quad \ell \triangleleft \tau \quad \Sigma; \Gamma; pc; \lambda_1 \vdash e_1 : \tau \dashv \lambda_0 \quad \Sigma; \Gamma; pc; \lambda_1 \vdash e_2 : \tau \dashv \lambda_0}{\Sigma; \Gamma; pc; \lambda_1 \vdash \text{if}\{pc\} v \text{ then } e_1 \text{ else } e_2 : \tau \dashv \lambda_0}$		$[\text{REF}] \frac{\Sigma; \Gamma \vdash v : \tau \quad pc \triangleleft \tau}{\Sigma; \Gamma; pc; \lambda_1 \vdash \text{ref } v \tau : (\text{ref } \tau)^\ell \dashv \lambda_0}$	
$[\text{DEREF}] \frac{\Sigma; \Gamma \vdash v : (\text{ref } \tau')^\ell \quad \tau' <: \tau \quad \ell \triangleleft \tau}{\Sigma; \Gamma; pc; \lambda_1 \vdash !v : \tau \dashv \lambda_0}$	$[\text{ASSIGN}] \frac{\Sigma; \Gamma \vdash v_1 : (\text{ref } \tau')^\ell \quad \Sigma; \Gamma \vdash v_2 : \tau \quad \ell \triangleleft \tau}{\Sigma; \Gamma; \ell; \lambda_1 \vdash v_1 := v_2 : \text{unit}^{\ell'} \dashv \lambda_0}$	$[\text{LOCK}] \frac{\Sigma; \Gamma; pc; \lambda'_1 \vdash e : \tau \dashv \lambda'_0 \quad \lambda'_1 \wedge \ell \Rightarrow \lambda_1 \quad \lambda'_0 \wedge \ell \Rightarrow \lambda_0}{\Sigma; \Gamma; pc; \lambda_1 \vdash \text{lock } \ell \text{ in } e : \tau \dashv \lambda_0}$	
$[\text{LET}] \frac{\Sigma; \Gamma; pc; \lambda_1 \vdash e_1 : \tau_1 \dashv \lambda'_0 \quad \lambda'_0 \Rightarrow \lambda_1 \quad \Sigma; \Gamma, x : \tau_1; pc; \lambda_1 \vdash e_2 : \tau_2 \dashv \lambda_0}{\Sigma; \Gamma; pc; \lambda_1 \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 \dashv \lambda_0}$		$[\text{VARIANCE}] \frac{\Sigma; \Gamma; pc'; \lambda'_1 \vdash e : \tau' \dashv \lambda'_0 \quad \tau' <: \tau \quad pc \Rightarrow pc' \quad \lambda'_1 \Rightarrow \lambda_1 \quad \lambda'_0 \Rightarrow \lambda_0}{\Sigma; \Gamma; pc; \lambda_1 \vdash e : \tau \dashv \lambda_0}$	
Tracking Statement Typing			
$[\text{ATPC}] \frac{\Sigma; \Gamma; pc; \lambda_1 \vdash s : \tau \dashv \lambda_0}{\Sigma; \Gamma; pc'; \lambda_1 \vdash s \text{ at-}pc \text{ } pc : \tau \dashv \lambda_0}$	$[\text{WITHLOCK}] \frac{\Sigma; \Gamma; pc; \lambda'_1 \vdash s : \tau \dashv \lambda'_0 \quad \lambda'_1 \wedge \ell \Rightarrow \lambda_1 \quad \lambda'_0 \wedge \ell \Rightarrow \lambda_0}{\Sigma; \Gamma; pc; \lambda_1 \vdash s \text{ with-lock } \ell : \tau \dashv \lambda_0}$		
$[\text{RETURN}] \frac{\Sigma; \cdot; pc; \lambda'_1 \vdash s : \tau \dashv \lambda'_0 \quad \lambda'_1 \vee \lambda'_0 \Rightarrow \lambda_0}{\Sigma; \Gamma; pc; \lambda_1 \vdash \text{return } s : \tau \dashv \lambda_0}$			
Attacker-Model Expression Typing			
$[\text{IGNORELOCKS}] \frac{\Sigma; \Gamma; pc; \lambda'_1 \vdash e : \tau \dashv \lambda'_0}{\Sigma; \Gamma; pc; \lambda_1 \vdash \text{ignore-locks-in } e : \tau \dashv \lambda_0}$			

Figure 10. Full typing rules for SeRIF values, expressions, and statements.

Class Typing		
$\lambda_1 \Rightarrow pc_2 \quad \ell_C \Rightarrow pc_2 \quad \lambda_1 \vee \lambda'_0 \Rightarrow \lambda_0 \quad pc_1 \triangleleft \overline{\tau_a}$ $\Sigma; \overline{x}; \overline{\tau_a}, \text{this}: C^{pc_2}; pc_2; \lambda_1 \vdash e: \tau \dashv \lambda'_0$		
[METHOD-OK]	$\frac{CT(C) = \text{class } C[\ell_C] \text{ extends } D \{ \dots \} \quad \text{can-override}(D, m, \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \lambda_0} \tau)}{\Sigma \vdash \tau m \{ pc_1 \gg pc_2; \lambda_0 \} (\overline{x}; \overline{\tau_a}) \{ e \} \text{ ok in } C}$	
[CLASS-OK]	$\frac{\begin{array}{l} \text{fields}(D) = \overline{g}; \overline{\tau_g} \\ K = C(\overline{g}; \overline{\tau_g}; \overline{f}; \overline{\tau_f}) \{ \text{super}(\overline{g}); \text{this}.\overline{f} = \overline{f} \} \\ \Sigma \vdash \overline{M} \text{ ok in } C \end{array}}{\Sigma \vdash \text{class } C[\ell_C] \text{ extends } D \{ \overline{f}; \overline{\tau_f}; K; \overline{M} \} \text{ ok}}$	
[CT-OK]	$\frac{\begin{array}{l} C \text{ referenced in any type} \implies C \in \text{dom}(CT) \\ \forall C \in \text{dom}(CT). \Sigma \vdash CT(C) \text{ ok} \end{array}}{\Sigma \vdash CT \text{ ok}}$	
Lookup Functions		
$\frac{CT(C) = \text{class } C[\ell_C] \text{ extends } D \{ \overline{f}; \overline{\tau_f}; K; \overline{M} \} \quad \text{fields}(D) = \overline{g}; \overline{\tau_g}}{\text{fields}(C) = \overline{g}; \overline{\tau_g}; \overline{f}; \overline{\tau_f}}$	$\frac{CT(C) = \text{class } C[\ell_C] \text{ extends } D \{ \overline{f}; \overline{\tau_f}; K; \overline{M} \} \quad \tau m \{ pc_1 \gg pc_2; \lambda_0 \} (\overline{x}; \overline{\tau_a}) \{ e \} \in \overline{M}}{\begin{array}{l} \text{mtype}(C, m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \lambda_0} \tau \\ \text{mbody}(C, m) = (\ell_C, \overline{x}, \overline{\tau_a}, pc_1 \gg pc_2, e, \tau) \end{array}}$	
$\frac{CT(C) = \text{class } C[\ell_C] \text{ extends } D \{ \overline{f}; \overline{\tau_f}; K; \overline{M} \} \quad m \text{ not defined in } \overline{M}}{\begin{array}{l} \text{mtype}(C, m) = \text{mtype}(D, m) \\ \text{mbody}(C, m) = \text{mbody}(D, m) \end{array}}$	$\frac{(D, m) \in \text{dom}(\text{mtype}) \implies \text{mtype}(D, m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \lambda_0} \tau}{\text{can-override}(D, m, \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \lambda_0} \tau)}$	
Subtyping		
$\frac{\ell \Rightarrow \ell'}{t^\ell <: t^{\ell'}}$	$\frac{CT(C) = \text{class } C[\ell_C] \text{ extends } D \{ \dots \}}{C^\ell <: D^\ell}$	$\frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$
Protection	Heap Typing	
$\frac{\ell \Rightarrow \ell'}{\ell \triangleleft t^{\ell'}}$	$\frac{\sigma(\iota) = (v, \tau) \implies \Sigma_\sigma \vdash v: \tau}{\vdash \sigma \text{ wt}}$	

Figure 11. Typing rules for SeRIF classes, auxiliary lookup functions, and relations.

- $\langle s \mid (CT, \sigma, \mathcal{M}, L) \rangle \longrightarrow \langle s' \mid (CT, \sigma', \mathcal{M}', L') \rangle$,
- $\Sigma_\sigma \vdash CT \text{ ok}$,
- $\Sigma_\sigma; \Gamma; pc; \lambda_I \vdash s : \tau \dashv \lambda_O$, and
- $\vdash \sigma \text{ wt}$,

then

- $\Sigma_\sigma \subseteq \Sigma_{\sigma'}$,
- $\vdash \sigma' \text{ wt}$, and
- $\Sigma_{\sigma'}; \Gamma; pc; \lambda_I \vdash s' : \tau \dashv \lambda_O$.

The proof of Theorem 6 makes use of several simple lemmas.

Lemma 1 (Closed Value Typing). *If $v \neq x$ and $\Sigma; \Gamma \vdash v : t^\ell$, then $\Sigma; \Gamma' \vdash v : t^\ell$ for any Γ' and ℓ .*

Proof. By inspection on the value typing rules. □

Lemma 2 (Value Substitution). *The following rule is admissible*

$$\frac{\Sigma; \Gamma, x : \tau'; pc; \lambda_I \vdash s : \tau \dashv \lambda_O \quad \Sigma; \Gamma \vdash v : \tau'}{\Sigma; \Gamma; pc; \lambda_I \vdash s[x \mapsto v] : \tau \dashv \lambda_O}$$

Proof. By simple structural induction on the proof that $\Sigma; \Gamma, x : \tau'; pc; \lambda_I \vdash s : \tau \dashv \lambda_O$. □

Lemma 3 (Heap-type Extension). *The following rules are admissible*

$$\frac{\Sigma; \Gamma \vdash v : \tau \quad \Sigma \subseteq \Sigma'}{\Sigma'; \Gamma \vdash v : \tau} \quad \frac{\Sigma; \Gamma; pc; \lambda_I \vdash s : \tau \dashv \lambda_O \quad \Sigma \subseteq \Sigma'}{\Sigma'; \Gamma; pc; \lambda_I \vdash s : \tau \dashv \lambda_O}$$

Proof. By simple induction on the proofs of $\Sigma; \Gamma \vdash v : \tau$ and $\Sigma; \Gamma; pc; \lambda_I \vdash s : \tau \dashv \lambda_O$. □

Lemma 4 (Heap Extension). *The following rule is admissible*

$$\frac{\vdash \sigma \text{ wt} \quad \Sigma_\sigma \vdash v : \tau \quad \iota \notin \text{dom}(\sigma)}{\vdash \sigma[\iota \mapsto (v, \tau)] \text{ wt}}$$

Proof. For notational ease, let $\sigma[\iota \mapsto (v, \tau)] = \sigma'$. First we note that because $\iota \notin \text{dom}(\sigma)$, we have that $\Sigma_{\sigma'} = \Sigma_\sigma \cup \{\iota \mapsto \tau\}$ with $\iota \notin \text{dom}(\Sigma_\sigma) = \text{dom}(\sigma)$. Now assume $\sigma'(\iota') = (v', \tau')$. If $\iota' = \iota$, then the premise of the rule gives us $\Sigma_\sigma \vdash v' : \tau'$, otherwise inversion on $\vdash \sigma \text{ wt}$ gives us the same property. By Lemma 3, $\Sigma_{\sigma'} \vdash v' : \tau'$, thereby proving $\vdash \sigma' \text{ wt}$. □

Lemma 5 (Statement Substitution). *If $\Sigma; \Gamma; pc; \lambda_I \vdash E[s_1] : \tau \dashv \lambda_O$ then there is some $\Gamma', pc', \lambda'_I, \tau'$, and λ'_O such that $\Sigma; \Gamma'; pc'; \lambda'_I \vdash s_1 : \tau' \dashv \lambda'_O$ and for any statement s_2 and heap-type $\Sigma' \supseteq \Sigma$, such that $\Sigma'; \Gamma'; pc'; \lambda'_I \vdash s_2 : \tau' \dashv \lambda'_O$, then $\Sigma'; \Gamma; pc; \lambda_I \vdash E[s_2] : \tau \dashv \lambda_O$.*

Proof. By simple induction on the proof of $\Sigma; \Gamma; pc; \lambda_I \vdash E[s_1] : \tau \dashv \lambda_O$. □

These lemmas are sufficient to prove type preservation.

Proof of Theorem 6. This will be a proof by induction on the typing rules and inversion on the operational semantics.

Case VAL: Values cannot step, so this is impossible.

Case ENDORSE: Because v must be a closed value, it type-checks with any label, so VAL proves the result.

- Case CAST:** Inversion on the operational semantics requires that $v = \text{new } C'(\bar{v})$ and $C' <: C$. Therefore **NEW**, **SUBTYPEV**, and **VAL** prove the case.
- Case FIELD:** Inversion on the operational semantics says $v = \text{new } D(\bar{v})$ and the premise of **FIELD** requires $\Sigma; \Gamma \vdash \text{new } D(\bar{v}) : C^\ell$. By inversion on the value typing rules, $D^\ell <: C^\ell$ and $\Sigma; \Gamma \vdash v_i : \tau_i$. Therefore, **SUBTYPEV** is sufficient to prove $\Sigma; \Gamma \vdash v_i : \tau$, and **VAL** completes the case.
- Case CALL:** Inversion on the operational semantics says $v = \text{new } D(\bar{v})$, and the premise of **CALL** requires $\Sigma; \Gamma \vdash \text{new } D(\bar{v}) : C^\ell$. By inversion on the value typing rules, $D^\ell <: C^\ell$. By the restrictions on overriding and the fact that $mtype(C, m) = \bar{\tau}_a \xrightarrow{pc_1 \gg pc_2; \lambda_0} \tau$, we know that $mbody(D, m) = (\ell_m, \bar{x}, \bar{\tau}_a, pc_1 \gg pc_2, e, \tau)$. **METHOD-OK** further requires $\Sigma; \bar{x} : \bar{\tau}_a, \text{this} : \tilde{D}^{pc_2}; pc_2; \lambda'_1 \vdash e : \tau \dashv \lambda'_0$ where $\lambda'_1 \vee \lambda'_0 \Rightarrow \lambda_0$ and $D <: \tilde{D}$. Therefore, using the premise that $\Sigma; \Gamma \vdash \bar{w} : \bar{\tau}_a$ and Lemma 2, it must be the case that $\Sigma; \cdot; pc_2; \lambda'_1 \vdash e[\bar{x} \mapsto \bar{w}, \text{this} \mapsto \text{new } D(\bar{v})] : \tau \dashv \lambda'_0$. This premise coupled with **RETURN** and **ATPC** prove the desired result.
- Case IF:** Inversion on the operational semantics requires that the step must be **E-IFT** or **E-IFF**. The appropriate premise of **IF** requiring the branches to type-check in the same environment and **ATPC** prove the case.
- Case REF:** By construction $\Sigma_{\sigma'}(\iota) = \tau$, so **LOC** and **VAL** prove the well-typed condition. Lemma 4 ensures $\vdash \sigma'$ wt, and $\sigma \subset \sigma'$, so $\Sigma_\sigma \subset \Sigma_{\sigma'}$.
- Case DEREf:** Inversion on the operational semantics shows the step uses **E-DEREf**, meaning $v = \iota$ and $\sigma(\iota) = (v', \tau)$. The assumption that $\vdash \sigma$ wt means $\Sigma_\sigma \vdash v' : \tau$, so that coupled with **SUBTYPEV** and **VAL** proves the case.
- Case ASSIGN:** Inversion on the operational semantics shows the step must be **E-ASSIGN**, which means $v_1 = \iota$, so inversion on the premise $\Sigma; \Gamma \vdash v_1 : (\text{ref } \tau)^\ell$ shows $\sigma(\iota) = (v, \tau)$, so **E-ASSIGN** requires $\Sigma_\sigma \vdash v_2 : \tau$. Therefore, $\Sigma_\sigma = \Sigma_{\sigma'}$ and $\sigma' = \sigma[\iota \mapsto (v_2, \tau)]$ remains well-typed. Finally, **UNIT** and **VAL** prove s' properly type-checks.
- Case LOCK:** The semantic rule must be **E-LOCK**, so $s' = e$ with-lock ℓ , and the premises of **WITHLOCK** are identical to **LOCK**, so **WITHLOCK** proves the case.
- Case LET:** Here we see $s = \text{let } x = s_1 \text{ in } e_2$. We consider two sub-cases, if $s_1 = v$ is a value, and if it is not. In the first sub-case, the operational semantic rule must be **LET**, and inversion on the typing rules proves that $\Sigma_\sigma; \Gamma \vdash v : \tau_1$, so Lemma 2 proves the sub-case. In the second sub-case, inversion on the operational semantics proves that the step must be **E-EVAL**. The **LET** rule's first premise is that $\Sigma_\sigma; \Gamma; pc; \lambda_1 \vdash s_1 : \tau_1 \dashv \lambda'_0$. Coupled with the inductive step in **E-EVAL** that $\langle s_1 \mid (CT, \sigma, \mathcal{M}, L) \rangle \longrightarrow \langle s'_1 \mid (CT, \sigma', \mathcal{M}', L') \rangle$, the inductive hypothesis proves that $\Sigma_{\sigma'}; \Gamma; pc; \lambda_1 \vdash s'_1 : \tau_1 \dashv \lambda'_0$ with $\Sigma_\sigma \subseteq \Sigma_{\sigma'}$ and $\vdash \sigma'$ wt. Lemma 3 therefore shows that $\Sigma_{\sigma'}; \Gamma, x : \tau_1; pc; \lambda_1 \vdash e_2 : \tau_2 \dashv \lambda_0$, so **LET** is sufficient to show $\Sigma_{\sigma'}; \Gamma; pc; \lambda_1 \vdash \text{let } x = s'_1 \text{ in } e_2 : \tau_2 \dashv \lambda_0$, finishing the case.
- Case VARIANCE:** By induction on the typing rules.
- Cases ATPC, WITHLOCK, and RETURN:** Each of these cases has two sub-cases: where the sub-statement is a value and where it is not. If the sub-statement is a value, the step must be **E-ATPC**, **E-UNLOCK**, or **E-RETURN**, respectively. In each case **VAL** allows values to type-check with any pc and lock labels, proving the case. If the sub-statement is not a value, the only step possible is **E-EVAL**. Here the proof follows by induction on the typing rules in the same manner as the **LET** case above. \square

Several semantic steps (**E-REF**, **E-ASSIGN**, and **E-CALL**) include information-security checks to guarantee that the code performing the operation is sufficiently trusted. The type system guarantees that these labels remain at least as trusted as the pc label of code executing. We formally define this property as a relation between a label stack and a statement, denoted by $\mathcal{M} \rightsquigarrow s$, and then prove that the semantics

maintains this relation. The relation is formally defined on evaluation contexts and extended to statements $s = E[e]$ if $\mathcal{M} \rightsquigarrow E$.

$$\frac{}{\ell_m \rightsquigarrow [\cdot]} \quad \frac{\mathcal{M} \rightsquigarrow E}{\mathcal{M} \rightsquigarrow \text{let } x = E \text{ in } e} \quad \frac{\mathcal{M} \rightsquigarrow E}{\mathcal{M} \rightsquigarrow E \text{ with-lock } \ell} \quad \frac{\mathcal{M} \rightsquigarrow E}{\mathcal{M} \rightsquigarrow \text{ignore-locks-in } E}$$

$$\frac{\mathcal{M} \rightsquigarrow E}{\ell, \mathcal{M} \rightsquigarrow \text{return}_\tau E} \quad \frac{\ell, \mathcal{M} \rightsquigarrow E \quad \ell \Rightarrow pc}{\ell, \mathcal{M} \rightsquigarrow E \text{ at-pc } pc}$$

Proposition 1. *For any statements s and s' and configurations $\mathcal{C} = (CT, \sigma, (\ell_m, \mathcal{M}), L)$ and $\mathcal{C}' = (CT, \sigma', \mathcal{M}', L')$, if $\vdash CT \text{ ok}$ and $(\ell_m, \mathcal{M}) \rightsquigarrow s$ and $\Sigma_\sigma; \Gamma; \ell_m; \lambda_1 \vdash s : \tau \dashv \lambda_0$ and $\langle s \mid \mathcal{C} \rangle \longrightarrow \langle s' \mid \mathcal{C}' \rangle$, then $\mathcal{M}' \rightsquigarrow s'$.*

The proof of Proposition 1 relies on two lemmas.

Lemma 6. *For any label list \mathcal{M} and evaluation contexts E_1 and E_2 , $\mathcal{M} \rightsquigarrow E_1[E_2]$ if and only if there exist $\mathcal{M}_1, \mathcal{M}_2$, and ℓ_m such that (1) $\mathcal{M}_1, \ell_m, \mathcal{M}_2 = \mathcal{M}$, (2) $\mathcal{M}_1, \ell_m \rightsquigarrow E_1$, and (3) $\ell_m, \mathcal{M}_2 \rightsquigarrow E_2$.*

Proof. This is a proof by induction on E_1 .

Case $E_1 = [\cdot]$:

(\Rightarrow) Let \mathcal{M}_1 be empty and note that \mathcal{M} cannot be empty, so $\mathcal{M} = \ell_m, \mathcal{M}_2$.

(\Leftarrow) By inversion on the rules, \mathcal{M}_1 must be empty, so $\mathcal{M} = \ell_m, \mathcal{M}_2$, proving the result.

Case $E_1 = (\text{let } x = E'_1 \text{ in } e), E'_1 \text{ with-lock } \ell, \text{ or ignore-locks-in } E'_1$:

(\Rightarrow) By induction, there exist $\mathcal{M}_1, \mathcal{M}_2$, and ℓ_m such that $\mathcal{M} = \mathcal{M}_1, \ell_m, \mathcal{M}_2$, $\mathcal{M}_1, \ell_m \rightsquigarrow E'_1$, and $\ell_m, \mathcal{M}_2 \rightsquigarrow E_2$. Therefore, by the appropriate rule, $\mathcal{M}_1, \ell_m \rightsquigarrow E_1$.

(\Leftarrow) By induction, $\mathcal{M}_1, \ell_m, \mathcal{M}_2 \rightsquigarrow E'_1[E_2]$, so by the appropriate rule, $\mathcal{M}_1, \ell_m, \mathcal{M}_2 \rightsquigarrow E_1[E_2]$.

Case $E_1 = \text{return}_\tau E'_1$:

(\Rightarrow) Inversion on the correspondence proves $\mathcal{M} = \ell, \mathcal{M}'$ and $\mathcal{M}' \rightsquigarrow E'_1[E_2]$. By induction, there is some $\mathcal{M}'_1, \ell_m, \mathcal{M}_2 = \mathcal{M}'$ such that $\mathcal{M}'_1, \ell_m \rightsquigarrow E'_1$ and $\ell_m, \mathcal{M}_2 \rightsquigarrow E_2$. Letting $\mathcal{M}_1 = \ell, \mathcal{M}'_1$ completes the case.

(\Leftarrow) By inversion on the correspondence rules, if $\mathcal{M}_1, \ell_m \rightsquigarrow E_1$, then $\mathcal{M}_1 = \ell, \mathcal{M}'_1$ for some ℓ and \mathcal{M}'_1 and $\mathcal{M}'_1, \ell_m \rightsquigarrow E'_1$. By induction, $\mathcal{M}'_1, \ell_m, \mathcal{M}_2 \rightsquigarrow E'_1[E_2]$, so therefore

$$\mathcal{M}_1, \ell_m, \mathcal{M}_2 = \ell, \mathcal{M}'_1, \ell_m, \mathcal{M}_2 \rightsquigarrow \text{return}_\tau E'_1[E_2] = E_1[E_2].$$

Case $E_1 = E'_1 \text{ at-pc } pc$:

(\Rightarrow) By inversion on the rules, $\mathcal{M} \rightsquigarrow E'_1[E_2]$, so by induction $\mathcal{M} = \mathcal{M}_1, \ell_m, \mathcal{M}_2$ with the desired properties. Moreover, $\mathcal{M} = \ell, \mathcal{M}'$ and $\ell \Rightarrow pc$. Because \mathcal{M}_1, ℓ_m is a non-empty prefix of \mathcal{M} , it must be the case that $\mathcal{M}_1, \ell_m = \ell, \mathcal{M}'_1$, so therefore $\mathcal{M}_1, \ell_m \rightsquigarrow E'_1 \text{ at-pc } pc = E_1$, as desired.

(\Leftarrow) By inversion on the correspondence rules, $\mathcal{M}_1, \ell_m \rightsquigarrow E'_1$, so by induction, $\mathcal{M} = \mathcal{M}_1, \ell_m, \mathcal{M}_2 \rightsquigarrow E'_1[E_2]$. Moreover, $\mathcal{M}_1, \ell_m = \ell, \mathcal{M}'_1$ and $\ell \Rightarrow pc$. Therefore $\mathcal{M} = \ell, \mathcal{M}'_1, \mathcal{M}_2$, satisfying the requirements to prove $\mathcal{M} \rightsquigarrow E'_1[E_2] \text{ at-pc } pc = E_1[E_2]$. \square

Lemma 7. *For statements s and s' , configurations $\mathcal{C} = (CT, \sigma, \mathcal{M}, L)$ and $\mathcal{C}' = (CT, \sigma', \mathcal{M}', L')$, and label lists \mathcal{M}_1 and \mathcal{M}_2 , if $\mathcal{M} = \mathcal{M}_1, \mathcal{M}_2$ and \mathcal{M}_2 is not empty, then $\langle s \mid \mathcal{C} \rangle \longrightarrow \langle s' \mid \mathcal{C}' \rangle$ if and only if $\langle s \mid \mathcal{C}[\mathcal{M}_2/\mathcal{M}] \rangle \longrightarrow \langle s' \mid \mathcal{C}'[\mathcal{M}'_2/\mathcal{M}] \rangle$ for some \mathcal{M}'_2 where $\mathcal{M}' = \mathcal{M}_1, \mathcal{M}'_2$.*

Proof. By simple induction on the operational semantics. \square

Proof of Proposition 1. This will be a proof by induction on the operational semantics.

Case E-EVAL: In this case $s = E[\tilde{s}]$, and by definition, $\tilde{s} = \tilde{E}[e]$. By Lemma 6, there exist $\mathcal{M}_1, \mathcal{M}_2$, and ℓ such that $\ell_m, \mathcal{M} = \mathcal{M}_1, \ell, \mathcal{M}_2$ where $\mathcal{M}_1, \ell \rightsquigarrow E$ and $\ell, \mathcal{M}_2 \rightsquigarrow \tilde{E}$. Therefore E-EVAL gives $\langle \tilde{s} \mid \mathcal{C} \rangle \longrightarrow \langle \tilde{s}' \mid \mathcal{C}' \rangle$, and because ℓ, \mathcal{M}_2 is non-empty, Lemma 7 proves $\langle \tilde{s} \mid \mathcal{C}[(\ell, \mathcal{M}_2)/\mathcal{M}] \rangle \longrightarrow \langle \tilde{s}' \mid \mathcal{C}'[\mathcal{M}'_2/\mathcal{M}] \rangle$, and moreover $\mathcal{M}' = \mathcal{M}_1, \mathcal{M}'_2$. Induction on this step ensures that $\mathcal{M}'_2 \rightsquigarrow \tilde{s}'$, so therefore \mathcal{M}'_2 must be non-empty. As a single step can only add or remove one element from \mathcal{M} , that means $\mathcal{M}'_2 = \ell, \mathcal{M}''_2$, so by Lemma 6, $\mathcal{M}' = \mathcal{M}_1, \ell, \mathcal{M}''_2 \rightsquigarrow E[\tilde{s}'] = s'$.

Case E-IFT and E-IFF: Here $s = \text{if}\{pc\} v$ then e_1 else e_2 . By inversion on the correspondence rules, $\mathcal{M} = \cdot$, and by inversion on the typing rules $\ell_m \Rightarrow pc$. Therefore $\ell_m \rightsquigarrow [\cdot]$ at-pc pc , so by definition $\mathcal{M}' = \ell_m \rightsquigarrow (e_i \text{ at-pc } pc) = s'$.

Case E-ATPC: Here $s = v$ at-pc pc and $s' = v$. By inversion on the correspondence rules, $\mathcal{M} = \cdot$ and $\mathcal{M}' = \ell_m$. Because $\ell_m \rightsquigarrow v$ for any v , this completes the case.

Cases E-CALL and E-CALLATK: Here $s = \text{new } C(\bar{v}).m(\bar{w})$ and $mbody(C, m) = (\ell'_m, \bar{x}, \bar{\tau}_a, pc_1 \gg pc_2, e, \tau)$. By inversion on the correspondence rules, $\mathcal{M} = \cdot$ and $\mathcal{M}' = \ell_m, \ell'_m$. By METHOD-OK, $\ell'_m \Rightarrow pc_2$. Therefore, letting $e' = e[\bar{x} \mapsto \bar{w}]$, this $\mapsto \text{new } C(\bar{v})$,

$$\frac{\frac{\ell'_m \rightsquigarrow e' \quad \ell'_m \Rightarrow pc_2}{\ell'_m \rightsquigarrow e' \text{ at-pc } pc_2}}{\ell_m, \ell'_m \rightsquigarrow \text{return}_\tau (e' \text{ at-pc } pc_2)} .$$

Case E-RETURN: Here $s = \text{return}_\tau v$, so inversion on the correspondence rules proves $\mathcal{M} = \ell$. Therefore $\mathcal{M}' = \ell_m \rightsquigarrow v = s'$ proves the case.

No other operational semantic rules modify \mathcal{M} or add or remove return or at-pc terms. Therefore the same proofs apply before and after the step. \square

The progress theorem is not without caveats. SeRIF's type system intentionally leaves checking of explicit casts, null dereferences, and dynamic reentrancy locks to run time. As a result, the progress theorem states that these three are the *only* ways a well-typed program can get stuck.

Theorem 7 (Progress). *For any statement s and configuration $\mathcal{C} = (CT, \sigma, (\ell_m, \mathcal{M}), L)$, if*

- $\Sigma_\sigma; \cdot; pc; \lambda_I \vdash s : \tau \dashv \lambda_O$,
- $\ell_m \Rightarrow pc$, and
- $(\ell_m, \mathcal{M}) \rightsquigarrow s$,

then one of the following holds:

1. s is a closed value,
2. $\langle s \mid \mathcal{C} \rangle \longrightarrow \langle s' \mid \mathcal{C}' \rangle$ for some s' and \mathcal{C}' ,
3. $s = E[(C)(\text{new } D(\bar{v}))]$ where $D \not<: C$,
4. $s = E[!\text{null}]$ or $s = E[\text{null} := v]$, or
5. $s = E[\text{new } C(\bar{v}).m(\bar{w})]$ for a C and m such that $mtype(C, m) = \bar{\tau}_a \xrightarrow{pc_1 \gg pc_2; \lambda_O} \tau$ and there is some $\ell_m \in L$ such that $pc_1 \not\neq pc_2 \vee \ell_m$.

Proof. This is a proof by induction on the derivation that $\Sigma_\sigma; \cdot; pc; \lambda_I \vdash s : \tau \dashv \lambda_O$.

Case VAL: Because $\Gamma = \cdot$, s is a closed value.

Case ENDORSE: Here $s = \text{endorse } v$ from ℓ to ℓ' . Since $\Gamma = \cdot$, v is a closed value, so E-ENDORSE applies.

Case CAST: Here $s = (C)v$. Inversion on the value typing rules coupled with the fact that $\Gamma = \cdot$ proves that $v = \text{new } D(\bar{v})$. If $D <: C$, then E-CAST applies with $\mathcal{C}' = \mathcal{C}$. Otherwise this is a bad cast.

Case FIELD: Here $s = v.f_i$. Again, inversion on the value typing rules with $\Gamma = \cdot$ proves $v = \text{new } C(\bar{v})$. Moreover FIELD requires reference to a valid fields, so E-FIELD steps s with $\mathcal{C}' = \mathcal{C}$.

Case CALL: Here $s = v.m(\bar{v})$. If a step can be taken, it must use E-CALL or E-CALLATK. Because $\Gamma = \cdot$, inversion on the premise that $\Sigma_\sigma; \Gamma \vdash v : C^\ell$ proves $v = \text{new } C(\bar{w})$. The premise $\Sigma; \Gamma \vdash \bar{v} : \bar{\tau}_a$ also directly proves the corresponding premise of E-CALL/E-CALLATK. Inversion on the proof that $(\ell_m, \mathcal{M}) \rightsquigarrow s$ proves that \mathcal{M} is empty, so therefore the premise of E-CALL/E-CALLATK requiring the caller's integrity to act for pc_1 is satisfied by $\ell_m \Rightarrow pc \Rightarrow pc_1$. At this point, E-CALLATK applies if $\ell_A \Rightarrow pc_2$ and E-CALL applies if $\bigwedge_{\ell \in L} (pc_1 \Rightarrow pc_2 \vee \ell)$. Therefore, if the statement is stuck, neither is satisfied, and the second is precisely the condition of a dynamic reentrancy lock blocking a call.

Case IF: Here $s = \text{if}\{pc'\} v$ then e_1 else e_2 . Inversion on the value typing rules using $\Gamma = \cdot$ means $v = \text{true}$ or $v = \text{false}$. Therefore E-IFT or E-IFF apply.

Case REF: Here $s = \text{ref } v \tau$. This step will be with E-REF. Since $\Gamma = \cdot$, the requirement that $\Sigma_\sigma \vdash v : \tau$ comes directly from REF. Moreover, inversion on the rules proving $(\ell_m, \mathcal{M}) \rightsquigarrow s$ shows that $\mathcal{M} = \cdot$, so the protection requirement of E-REF is $\ell_m \triangleleft \tau$ and $\ell_m \Rightarrow pc \triangleleft \tau$, meaning the step applies with some fresh $\iota \notin \text{dom}(\sigma)$.

Case Deref: Here $s = !v$. Since $\Gamma = \cdot$, inversion on the Deref premise that $\Sigma \vdash v : (\text{ref } \tau')^\ell$ means $v = \iota$ with $\Sigma_\sigma(\iota) = \tau'$ or $v = \text{null}$. In the first case, by definition this means $\sigma(\iota) = (v', \tau')$ for some v' , so E-Deref applies. In this second case, this is a null dereference.

Case ASSIGN: Here $s = (v_1 := v_2)$. Again, $\Gamma = \cdot$ and inversion on the typing rules using the premise $\Sigma; \Gamma \vdash v_1 : (\text{ref } \tau)^\ell$ proves that $v_1 = \iota$ or $v_1 = \text{null}$. If $v_1 = \text{null}$, then this is a null dereference. If $v_1 = \iota$, then the step must be E-ASSIGN. The requirement that $\Sigma_\sigma(\iota) = \tau$ and $\Sigma \vdash v_2 : \tau$ stem from inversion on the typing derivation of v_1 and the second premise of ASSIGN. Finally, inversion on the rules proving $(\ell_m, \mathcal{M}) \rightsquigarrow s$ shows that $\mathcal{M} = \cdot$, and ASSIGN requires $pc \vee \ell \triangleleft \tau$, so the transitivity of \Rightarrow proves $\ell_m \triangleleft \tau$, as needed.

Case LOCK: E-LOCK always applies.

Case LET: Here $s = \text{let } x = \tilde{s} \text{ in } e$. The first hypothesis of LET proves $\Sigma_\sigma; \cdot; pc; \lambda_1 \vdash \tilde{s} : \tau_1 \dashv \lambda'_0$, and $(\ell_m, \mathcal{M}) \rightsquigarrow \tilde{s}$. Therefore, our inductive hypothesis applies to \tilde{s} . If \tilde{s} is a closed value, then E-LET applies to s , stepping to $s' = e[x \mapsto \tilde{s}]$ letting $C' = C$. If $\langle \tilde{s} \mid C \rangle \longrightarrow \langle \tilde{s}' \mid C' \rangle$, then by E-EVAL, $\langle s \mid C \rangle \longrightarrow \langle \text{let } x = \tilde{s}' \text{ in } e \mid C' \rangle$. For the other three cases where $\tilde{s} = E[e']$ where e' is a failure condition, we note that $\text{let } x = E \text{ in } e$ is an evaluation context, so s falls into the same failure case.

Case VARIANCE: Because $\ell_m \Rightarrow pc \Rightarrow pc'$, this case follows directly by induction.

Case ATPC: Here $s = \tilde{s} \text{ at-pc } pc'$. Inversion on the proof that $(\ell_m, \mathcal{M}) \rightsquigarrow s$ proves $\ell_m \Rightarrow pc'$. The hypothesis of ATPC is $\Sigma_\sigma; \cdot; pc'; \lambda_1 \vdash \tilde{s} : \tau \dashv \lambda_0$, so the inductive hypothesis applies to \tilde{s} .

If \tilde{s} is a closed value, then E-ATPC applies letting $C' = C$. If \tilde{s} steps to \tilde{s}' , then E-EVAL proves $\langle s \mid C \rangle \longrightarrow \langle \tilde{s}' \text{ at-pc } pc' \mid C' \rangle$. For the other three cases, as with LET, $\tilde{s} = E[e]$ where e is a failure condition, so $E \text{ at-pc } pc'$ is an evaluation context proving that s falls into the same failure case as \tilde{s} .

Cases WITHLOCK and IGNORELOCKS: The logic of these cases is the same as the logic of the ATPC case, but using pc instead of pc' .

Case RETURN: Here $s = \text{return}_\tau \tilde{s}$. Inversion on the proof that $(\ell_m, \mathcal{M}) \rightsquigarrow s$ shows that \mathcal{M} is not empty and $\mathcal{M} \rightsquigarrow \tilde{s}$. Additionally, a premise of RETURN is $\Sigma_\sigma; \cdot; pc; \lambda'_1 \vdash \tilde{s} : \tau \dashv \lambda'_0$. Therefore, the inductive hypothesis applies using Lemma 7 to replace (ℓ_m, \mathcal{M}) with \mathcal{M} in C .

If \tilde{s} is a closed value, the well-typed premise of RETURN proves $\Sigma_\sigma \vdash v : \tau$, and since (ℓ_m, \mathcal{M}) is non-empty, E-RETURN applies. If \tilde{s} steps to \tilde{s}' , then E-EVAL allows s to step as well. Again, for the three failure cases where $\tilde{s} = E[e]$, simply replacing E with $\text{return}_\tau E$ creates the expected form. \square

Note that, for any invocation $I = (\ell, \iota, m(\bar{v}))$, $\ell \rightsquigarrow !\iota.m(\bar{v})$. Therefore, if the invocation and class table are well-typed in Σ_σ for a well-typed heap σ , Theorems 6 and 7 combine with Proposition 1 to prove that the invocation either steps to a closed value with a well-typed heap or gets stuck on one of the three run-time

error checks.

D Proof of Noninterference

We now provide a proof of Theorem 1 presented in Section 6. We prove Theorem 1 using an erasure-based construction. Specifically, we will erase low-integrity values in the heap and then execute the same program using a modified semantics that continues to omit low-integrity values from the state and uses a special value, \bullet , when one would be read. We prove that, if the original execution terminated and the code is endorsement-free, this modified execution must terminate and, critically, the high-integrity components of the state must match. The theorem then follows by noting that if $\sigma_1 \approx_{\ell_t} \sigma_2$, then both executions must produce heaps that's high-integrity components are the same as the modified execution on a partially-erased heap.

Formally, we introduce a new value to denote erased data.

$$v ::= \dots \mid \bullet$$

The typing and semantic rules that handle \bullet are parameterized on a label ℓ_t defining high-integrity values. For notational ease, we omit that label in our syntax. However, as our theorems are all parameterized over ℓ_t , they remain true for any possible choice of ℓ_t .

The type system allows \bullet to be any type, as long as that type is low-integrity. To simplify notation, we define $\text{label}(t^\ell) = \ell$.

$$\text{[BULLET]} \frac{\text{label}(\tau) \not\Rightarrow \ell_t}{\Sigma; \Gamma \vdash \bullet : \tau}$$

We introduce an expanded operational semantics to deal with these terms. To separate executions with and without bullets, we define a new step function denoted \rightarrow when working with erased terms. We also define a context B defining syntactic forms that normally require a decision based on the value. The B-BULLETCtx rule simply erases the entire expression when the given value is \bullet .

$$B ::= \text{if}\{pc\} [\cdot] \text{ then } e \text{ else } e \mid ![\cdot] \mid (C)[\cdot] \mid [\cdot].f \mid [\cdot].m(\bar{v})$$

$$\text{[B-PURESTEP]} \frac{\langle s \mid (CT, \sigma, \mathcal{M}, L) \rangle \longrightarrow \langle s' \mid (CT, \sigma, \mathcal{M}', L') \rangle}{\langle s \mid (CT, \sigma, \mathcal{M}, L) \rangle \rightarrow \langle s' \mid (CT, \sigma, \mathcal{M}', L') \rangle} \quad \text{[B-EVAL]} \frac{\langle s \mid C \rangle \rightarrow \langle s' \mid C' \rangle}{\langle E[s] \mid C \rangle \rightarrow \langle E[s'] \mid C' \rangle}$$

$$\text{[B-BULLETCtx]} \frac{}{\langle B[\bullet] \mid C \rangle \rightarrow \langle \bullet \mid C \rangle}$$

$$\text{[B-TREF]} \frac{\iota \notin \text{dom}(\sigma) \quad \Sigma_\sigma \vdash v : \tau \quad \mathcal{M} = \mathcal{M}', \ell_m \quad \ell_m \triangleleft \tau \quad \text{label}(\tau) \Rightarrow \ell_t}{\langle \text{ref } v \ \tau \mid C \rangle \rightarrow \langle \iota \mid C[\sigma[\iota \mapsto (v, \tau)]]/\sigma \rangle}$$

$$\text{[B-UREF]} \frac{\iota \notin \text{dom}(\sigma) \quad \Sigma_\sigma \vdash v : \tau \quad \mathcal{M} = \mathcal{M}', \ell_m \quad \ell_m \triangleleft \tau \quad \text{label}(\tau) \not\Rightarrow \ell_t}{\langle \text{ref } v \ \tau \mid C \rangle \rightarrow \langle \iota \mid C[\sigma[\iota \mapsto (\bullet, \tau)]]/\sigma \rangle} \quad \text{[B-BASSIGN]} \frac{}{\langle \bullet := v \mid C \rangle \rightarrow \langle () \mid C \rangle}$$

$$\text{[B-TASSIGN]} \frac{\Sigma_\sigma(\iota) = \tau \quad \Sigma_\sigma \vdash v : \tau \quad \mathcal{M} = \mathcal{M}', \ell_m \quad \ell_m \triangleleft \tau \quad \text{label}(\tau) \Rightarrow \ell_t}{\langle \iota := v \mid C \rangle \rightarrow \langle () \mid C[\sigma[\iota \mapsto (v, \tau)]]/\sigma \rangle}$$

$$\text{[B-UASSIGN]} \frac{\Sigma_\sigma(\iota) = \tau \quad \Sigma_\sigma \vdash v : \tau \quad \mathcal{M} = \mathcal{M}', \ell_m \quad \ell_m \triangleleft \tau \quad \text{label}(\tau) \not\Rightarrow \ell_t}{\langle \iota := v \mid C \rangle \rightarrow \langle () \mid C[\sigma[\iota \mapsto (\bullet, \tau)]]/\sigma \rangle}$$

These semantics inherit from our original operation semantics whenever the step does not modify the heap. When modifying the heap, however, \rightarrow omits any values that are in low-integrity memory locations, while treating high-integrity memory locations normally. When reading from the heap, it produces \bullet whenever it tries to read from a location that has a type but not a value. In our construction for our proof, these will be precisely the low-integrity locations.

We now claim that, if CT is endorsement-free at ℓ_t , then any invocation with input state σ_1 will, under normal semantics, produce a state σ_2 that is ℓ_t -equivalent to executing the same invocation under bullet semantics with input state $\sigma_1|_{\ell_t}$.

Lemma 8 (Label Stack Maintenance). *For any expression e , if*

$$\langle e \mid (CT, \sigma, \mathcal{M}, L) \rangle \longrightarrow^* \langle v \mid (CT, \sigma', \mathcal{M}', L') \rangle,$$

then $\mathcal{M}' = \mathcal{M}$ and $L' = L$.

Proof. This will be a proof by induction on the number of steps from e to v and on the operational semantics. In the base case, there are zero steps, so the result trivially holds.

We now assume $\langle e \mid (CT, \sigma, \mathcal{M}, L) \rangle \longrightarrow^* \langle v \mid (CT, \sigma', \mathcal{M}', L') \rangle$ takes $n \geq 1$ steps and the result holds for all executions of $k < n$ steps. We consider the following cases.

Case E-EVAL: If $E = [\cdot]$, we can replace this step with another, so without loss of generality, we assume $E \neq [\cdot]$. Since $e = E[\tilde{e}]$ is an expression, \tilde{e} is also an expression and $E = \text{let } x = x \text{ in } E' e''$. By inversion on the operational semantics, E-EVAL is the only rule that can apply until \tilde{e} reaches some value \tilde{v} . Moreover, $E[\tilde{v}]$ is not a value, so $\langle \tilde{e} \mid \mathcal{C} \rangle \longrightarrow^* \langle \tilde{v} \mid \tilde{\mathcal{C}} \rangle$ in fewer steps. By induction, we therefore have that $\tilde{\mathcal{M}} = \mathcal{M}$ and $\tilde{L} = L$. Moreover, since $E[\tilde{e}]$ was surface syntax, $E[\tilde{v}]$ must be as well. Therefore, another application of our inductive hypothesis proves

$$\langle E[\tilde{e}] \mid (CT, \sigma, \mathcal{M}, L) \rangle \longrightarrow^* \langle E[\tilde{v}] \mid (CT, \tilde{\sigma}, \mathcal{M}, L) \rangle \longrightarrow^* \langle v \mid (CT, \sigma', \mathcal{M}, L) \rangle$$

Cases E-IFT and E-IFF: Because $e = \text{if}\{pc\} v'$ then e_1 else e_2 was surface-syntax, e_i must also be surface syntax. Inspection on the semantic rules says that any expression of the form \tilde{e} at-pc pc can only step using E-EVAL if \tilde{e} steps or E-ATPC if \tilde{e} is a value. Therefore, we know that $\langle \tilde{e} \mid \mathcal{C} \rangle \longrightarrow^* \langle v \mid \mathcal{C}' \rangle$, and then v at-pc pc steps once using E-ATPC. By induction on the number of steps, we therefore have that $\mathcal{M}' = \mathcal{M}$ and $L' = L$.

Case E-LOCK: This case is similar to the previous case. Again, we know that $e = \text{lock } \tilde{e} \text{ in } \ell$ and \tilde{e} is surface-syntax. We also know that $\langle e \mid \mathcal{C} \rangle \longrightarrow \langle \tilde{e} \text{ with-lock } \ell \mid \mathcal{C}[L, \ell/L] \rangle$. By the same argument as above, \tilde{e} must step to a value in fewer steps, so by induction

$$\langle \tilde{e} \mid (CT, \sigma, \mathcal{M}, (L, \ell)) \rangle \longrightarrow^* \langle v \mid (CT, \sigma', \mathcal{M}, (L, \ell)) \rangle$$

A single application of E-UNLOCK then gives us the desired result.

Cases E-CALL and E-CALLATK: These cases are identical to the previous one, but modifying \mathcal{M} instead of L and using E-RETURN instead of E-UNLOCK.

Cases E-UNLOCK and E-RETURN: These are impossible because e is surface-syntax.

In all other cases, stepping e once continues to be surface syntax and leaves \mathcal{M} and L unmodified. We can therefore remove a single step and apply our inductive hypothesis. \square

Lemma 9 (Step Confinement). *For a state σ_1 where $\Sigma \subseteq \Sigma_{\sigma_1}$ and a statement s_1 , if*

1. $\Sigma \vdash CT \text{ ok}$ is endorsement-free at ℓ_t ,
2. $\vdash \sigma_1 \text{ wt}$,

3. $\Sigma; \Gamma; pc; \lambda_1 \vdash s_1 : \tau \dashv \lambda_0$,
4. $pc \not\# \ell_t$,
5. for all sub-statements s at-pc pc' of s_1 , $pc' \not\# \ell_t$, and
6. $\langle s_1 \mid (CT, \sigma_1, \mathcal{M}_1, L_1) \rangle \longrightarrow \langle s_2 \mid (CT, \sigma_2, \mathcal{M}_2, L_2) \rangle$,

then

- $\sigma_1 \approx_{\ell_t} \sigma_2$ and
- for all sub-statements s at-pc pc' of s_2 , $pc' \not\# \ell_t$.

Proof. This will be a proof by induction on the semantic rule used to take a step. The following are the nontrivial cases.

Case E-EVAL: Here we have $s_1 = E[\tilde{s}_1]$. We claim by induction on E that $\Sigma; \Gamma; pc'; \lambda'_1 \vdash \tilde{s}_1 : \tau' \dashv \lambda'_0$ for some pc' , λ'_1 , τ' , and λ'_0 where $pc' \not\# \ell_t$. If $E = [\cdot]$, this follows directly from our assumptions. If $E = \text{let } x = x \text{ in } E' s'$, $\text{return}_\tau E'$, or E' with-lock ℓ , we note that $\Sigma; \Gamma; pc; \lambda'_1 \vdash E'[\tilde{s}_1] : \tau' \dashv \lambda'_0$ for some λ'_1 , τ' , and λ'_0 , so by induction on E , we have the desired result. If $E = E' \text{ at-pc } pc''$, we note that $\Sigma; \Gamma; pc''; \lambda'_1 \vdash E'[\tilde{s}_1] : \tau' \dashv \lambda_0$ and, by assumption, $pc'' \not\# \ell_t$. Thus induction on E again gets us the desired typing judgment.

E-EVAL tells us $\langle \tilde{s}_1 \mid (CT, \sigma_1, \mathcal{M}_1, L_1) \rangle \longrightarrow \langle \tilde{s}_2 \mid (CT, \sigma_2, \mathcal{M}_2, L_2) \rangle$ and $s_2 = E[\tilde{s}_2]$. Since \tilde{s}_1 is a sub-statement of s_1 , it must satisfy hypothesis 5, and the typing judgment above gives us hypotheses 3 and 4. Induction on the operational semantics therefore gives us $\sigma_1 \approx_{\ell_t} \sigma_2$ and, for all sub-statements e at-pc pc' of \tilde{s}_2 , $pc' \not\# \ell_t$. By hypothesis 5, the same must be true of E , so therefore $E[\tilde{s}_2] = s_2$ satisfies the required condition.

Cases E-IFT and E-IFF: Here $s_1 = \text{if}\{pc'\} v$ then s'_1 else s'_2 . We know that $\sigma_1 = \sigma_2$, so that condition is trivially true. Both s'_1 and s'_2 are surface-syntax, so they contain no sub-statement of the form e at-pc pc'' , meaning the only such sub-statement in $s_2 = s'_i$ at-pc pc' is the outer one. By inversion on the typing rules, we know that $pc \Rightarrow pc'$, so by transitivity, $pc' \not\# \ell_t$.

Case E-REF: Here $s_1 = \text{ref } v \tau'$. By inversion on the typing rules, we know that $pc \triangleleft \tau'$, and by assumption, $pc \not\# \ell_t$. Therefore, since $\iota \notin \text{dom}(\sigma_1)$ and $\sigma_2 = \sigma_1[\iota \mapsto (v, \tau')]$, we know that $\sigma_1|_{\ell_t} = \sigma_2|_{\ell_t}$, which is exactly the definition of $\sigma_1 \approx_{\ell_t} \sigma_2$. There are no sub-statements of the form e at-pc pc' , so that result is trivially true.

Case E-ASSIGN: Here $s_1 = \iota := v$. By inversion on the typing rules, we know $\Sigma(\iota) = \tau'$ and $pc \triangleleft \tau'$. By assumption, $pc \not\# \ell_t$, so therefore $\iota \notin \text{dom}(\sigma_1)$. Given this and the fact that $\Sigma_{\sigma_2} = \Sigma_{\sigma_1}$, again $\sigma_1 \approx_{\ell_t} \sigma_2$, as desired. As in the previous case, there are no sub-statements of the form e at-pc pc' .

Cases E-CALL and E-CALLATK: Here $s_1 = \text{new } C(\bar{v}).m(\bar{w})$ with $mbody(C, m) = (\ell_m, \bar{x}, \bar{\tau}_a, pc_1 \gg pc_2, e, \tau)$. Inversion on the typing rules proves that $pc \Rightarrow pc_1$, and by assumption, $pc \not\# \ell_t$. Therefore, by transitivity, $pc_1 \not\# \ell_t$, so, by the definition of CT being endorsement-free at ℓ_t , it must be the case that $pc_2 \not\# \ell_t$. Moreover, $e[\bar{x} \mapsto \bar{w}, \text{this} \mapsto \text{new } C(\bar{v})]$ is surface-syntax, so the only sub-statement of the form s' at-pc pc' on s_2 is the outer one where $pc' = pc_2$, and we just proved $pc_2 \not\# \ell_t$. Finally, the step leaves the heap and heap type unmodified, finishing the case.

All other cases leave the heap and heap type unmodified and do not add sub-statement of the form e at-pc pc' , making the result trivial in those cases. \square

Corollary 1 (Confinement). *Given a class table CT and an expression (not statement) e , if*

- CT and e are both endorsement-free at ℓ_t ,
- $\Sigma_\sigma \vdash CT$ ok,
- $\Sigma_\sigma; \Gamma; pc; \lambda_1 \vdash e : \tau \dashv \lambda_0$ for some $pc \not\# \ell_t$, and
- $\langle e \mid (CT, \sigma, \mathcal{M}, L) \rangle \longrightarrow^* \langle v \mid (CT, \sigma', \mathcal{M}', L') \rangle$,

then $\sigma \approx_{\ell_t} \sigma'$, $\mathcal{M} = \mathcal{M}'$, and $L = L'$.

Proof. We apply Lemma 8 and inductively apply Lemma 9 using the fact that expressions cannot contain any subexpressions of the form e' at-pc pc' . \square

We aim to prove something about execution in our regular semantics through execution in our semantics with bullets, so we need a way to relate terms with and without bullets. We do this using a syntactic relation denoted $e_1 \geq_{\bullet} e_2$ to indicate that e_2 is just e_1 but possibly with some information erased. On values, the relation is defined as follows.

$$\frac{v \neq \bullet}{v \geq_{\bullet} v} \qquad \frac{v \notin \{x, \bullet\}}{v \geq_{\bullet} \bullet} \qquad \frac{\bar{v} \geq_{\bullet} \bar{w}}{\text{new } C(\bar{v}) \geq_{\bullet} \text{new } C(\bar{w})}$$

We extend this relation to typing proofs. We first relate typing proofs of closed values (so proofs that do not use VAR) to value typing proofs using BULLET. Note that we do not mandate that the heap types be the same at every location so long as they are the same at the locations used in the typing proof. That is

$$[\text{LOC}] \frac{\Sigma_1(\iota) = \tau}{\Sigma_1; \Gamma \vdash \iota : (\text{ref } \tau)^\ell} \geq_{\bullet} \frac{\Sigma_2(\iota) = \tau}{\Sigma_2; \Gamma \vdash \iota : (\text{ref } \tau)^\ell} [\text{LOC}]$$

whenever $\Sigma_1(\iota) = \Sigma_2(\iota)$, even if Σ_1 and Σ_2 differ on other locations. Notably, if $\Sigma_1(\iota) = \tau \neq \Sigma_2(\iota)$ (possibly because $\iota \notin \text{dom}(\Sigma_2)$) and $\ell \not\equiv \ell_t$, then,

$$[\text{LOC}] \frac{\Sigma_1(\iota) = \tau}{\Sigma_1; \Gamma \vdash \iota : (\text{ref } \tau)^\ell} \geq_{\bullet} \frac{\ell \not\equiv \ell_t}{\Sigma_2; \Gamma \vdash \bullet : (\text{ref } \tau)^\ell} [\text{BULLET}].$$

We finally extend the relation to typing proofs of expressions and statements by extending it structurally. That is, if the typing proofs of each sub-statement is related, then the typing proof of the whole statement is related. For example,

$$\frac{\frac{\frac{\pi_1}{\Sigma_1; \Gamma \vdash v_1 : \tau} \geq_{\bullet} \frac{\pi_2}{\Sigma_2; \Gamma \vdash v_2 : \tau}}{\Sigma_1; \Gamma; pc; \lambda_I \vdash \text{ref } v \tau : (\text{ref } \tau)^\ell \dashv \lambda_0} \geq_{\bullet} \frac{\frac{\pi_2}{\Sigma_2; \Gamma \vdash v_2 : \tau} \quad pc \triangleleft \tau}{\Sigma_2; \Gamma; pc; \lambda_I \vdash \text{ref } v \tau : (\text{ref } \tau)^\ell \dashv \lambda_0}}$$

We usually denote this relation $\Sigma_1; \Gamma; pc; \lambda_I \vdash s_1 : \tau \dashv \lambda_0 \geq_{\bullet} \Sigma_2; \Gamma; pc; \lambda_I \vdash s_2 : \tau \dashv \lambda_0$.

We now use this relation to relate executions in the regular semantics and the erasure semantics. For this we use a slightly modified erasure procedure on heaps, $\sigma|_{\ell_t}^{\bullet}$. Instead of simply removing all low-integrity mappings, it instead replaces the values with \bullet .

$$\sigma|_{\ell_t}^{\bullet}(\iota) \triangleq \begin{cases} (v, t^\ell) & \text{if } \sigma(\iota) = (v, t^\ell) \text{ and } \ell \Rightarrow \ell_t \\ (\bullet, t^\ell) & \text{if } \sigma(\iota) = (v, t^\ell) \text{ and } \ell \not\equiv \ell_t \end{cases}$$

Lemma 10 (Bullet Semantics Completeness). *Let $\mathcal{C}_i = (CT, \sigma_i, \mathcal{M}, L)$. If*

- $\Sigma_{\sigma_1}; \Gamma; pc; \lambda_I \vdash s_1 : \tau \dashv \lambda_0 \geq_{\bullet} \Sigma_{\sigma_2}; \Gamma; pc; \lambda_I \vdash s_2 : \tau \dashv \lambda_0$,
- $\langle s_1 \mid \mathcal{C}_1 \rangle \longrightarrow \langle s'_1 \mid \mathcal{C}'_1 \rangle$,

then $\langle s_2 \mid \mathcal{C}_2 \rangle \dashv \bullet \longrightarrow \langle s'_2 \mid \mathcal{C}'_2 \rangle$.

Proof. This is a proof by induction on the operational semantics of $s_1 \longrightarrow s'_1$.

Case E-EVAL: By induction on E , if $s_1 = E[\tilde{s}_1]$, then $s_2 = E'[\tilde{s}_2]$ where $\tilde{s}_1 \geq_{\bullet} \tilde{s}_2$. By induction on the operational semantics, $\langle \tilde{s}_2 \mid \mathcal{C}_2 \rangle \dashv \bullet \longrightarrow \langle \tilde{s}'_2 \mid \mathcal{C}'_2 \rangle$, and B-EVAL applies to complete the case.

Cases E-IFT and E-IFF: Here we have $s_1 = \text{if}\{pc\} \tilde{v}$ then \tilde{e}_1 else \tilde{e}_2 , and $s_2 = \text{if}\{pc\} v^\bullet$ then e_1^\bullet else e_2^\bullet . We consider two sub-cases. First, if $v^\bullet = \bullet$, we see that $\langle s_2 \mid \mathcal{C}_2 \rangle \dashrightarrow \langle \bullet \mid \mathcal{C}_2 \rangle$ by B-BULLETCtx. If $v^\bullet \neq \bullet$, then $v^\bullet = \tilde{v}$, and therefore B-PURESTEP allows s_2 to step, as desired.

Cases E-CAST, E-FIELD, E-CALL, and E-CALLATK: These cases follow the same logic as the previous case, with their corresponding syntax.

Case E-REF: Here we have $s_1 = \text{ref } v_1 \tau$ so therefore $s_2 = \text{ref } v_2 \tau$ where $v_1 \geq_\bullet v_2$. Inversion on E-REF proves $\mathcal{M} = \mathcal{M}'$, ℓ_m where $\ell_m \triangleleft \tau$. Since \mathcal{M} is the same in \mathcal{C}_1 and \mathcal{C}_2 , if $\text{label}(\tau) \Rightarrow \ell_t$, then B-TREF applies, and if not, B-UREF applies.

Case E-DEREF: Here $s_1 = !\iota$ and $s_2 = !v$ where either $v = \bullet$ or $v = \iota$. If $v = \bullet$, then B-BULLETCtx applies. Otherwise, we know that $\Sigma_{\sigma_2}; \Gamma; pc, \lambda_1 \vdash !\iota : \tau \dashv \lambda_0$. By inversion on the expression typing rules, we know that $\Sigma_{\sigma_2}; \Gamma \vdash \iota : (\text{ref } \tau)^\ell$, and by inversion on the value typing rules, we therefore have $\Sigma_{\sigma_2}(\iota) = \tau$. In other words, $\iota \in \text{dom}(\sigma_2)$, so B-PURESTEP applies with E-DEREF.

Case E-ASSIGN: In this case $s_1 = \iota := v$ and $s_2 = v_1 := v_2$ where $v_1 = \bullet$ or $v_1 = \iota$. If $v_1 = \bullet$, then B-BASSIGN applies. Otherwise, because s_1 is well-typed with Σ_{σ_1} , inversion on the typing rules proves $\Sigma_{\sigma_1}(\iota) = \tau'$. Because s_1 steps with E-ASSIGN, inversion on E-ASSIGN proves $\mathcal{M} = \mathcal{M}'$, ℓ_m where $\ell_m \triangleleft \tau'$. By inversion on the \geq_\bullet relation, it must be the case that $\Sigma_{\sigma_2}(\iota) = \tau'$ and $\Sigma_{\sigma_2}; \Gamma \vdash v_2 : \tau'$. Since \mathcal{M} is the same in \mathcal{C}_1 and \mathcal{C}_2 , this is sufficient to apply one of B-TASSIGN or B-UASSIGN, depending on $\text{label}(\tau')$.

For all other cases, the heap remains unmodified and no decisions are made based on a value that may be \bullet , so B-PURESTEP applies to s_2 using the same step that applied to s_1 . \square

Lemma 11 (Bullet Step Correspondence). *For any class table CT , statements s_1 and s_2 , heaps σ_1 and σ_2 , and heap-type Σ , if*

- $\Sigma \vdash CT \text{ ok}$ is endorsement-free at ℓ_t ,
- s_1 and s_2 are endorsement-free at ℓ_t ,
- $\vdash \sigma_i \text{ wt}$ for both $i = 1, 2$,
- $\sigma_1 \approx_{\ell_t} \sigma_2$ with $\sigma_2 \subseteq \sigma_1|_{\ell_t}^\bullet$,
- $\Sigma \subseteq \Sigma_{\sigma_2}$,
- $\Sigma_{\sigma_1}; \Gamma; pc; \lambda_1 \vdash s_1 : \tau \dashv \lambda_0 \geq_\bullet \Sigma_{\sigma_2}; \Gamma; pc; \lambda_1 \vdash s_2 : \tau \dashv \lambda_0$, and
- $\langle s_1 \mid (CT, \sigma_1, \mathcal{M}, L) \rangle \dashrightarrow^+ \langle v \mid \mathcal{C} \rangle$,

then there exists statements s'_1 and s'_2 , heaps σ'_1 and σ'_2 , and label stacks \mathcal{M}' and L' such that

- $\langle s_1 \mid (CT, \sigma_1, \mathcal{M}, L) \rangle \dashrightarrow^+ \langle s'_1 \mid (CT, \sigma'_1, \mathcal{M}', L') \rangle$,
- $\langle s_2 \mid (CT, \sigma_2, \mathcal{M}, L) \rangle \dashrightarrow \langle s'_2 \mid (CT, \sigma'_2, \mathcal{M}', L') \rangle$,
- s'_1 and s'_2 are endorsement-free at ℓ_t ,
- $\vdash \sigma'_i \text{ wt}$ for both $i = 1, 2$,
- $\sigma'_1 \approx_{\ell_t} \sigma'_2$ with $\sigma'_2 \subseteq \sigma'_1|_{\ell_t}^\bullet$, and
- $\Sigma_{\sigma'_1}; \Gamma; pc; \lambda_1 \vdash s'_1 : \tau \dashv \lambda_0 \geq_\bullet \Sigma_{\sigma'_2}; \Gamma; pc; \lambda_1 \vdash s'_2 : \tau \dashv \lambda_0$.

Proof. By Lemma 10, the fact that $s_1 \dashrightarrow^+ v$ means that $s_2 \dashrightarrow s'_2$. This will be a proof by induction on the rule used to prove $s_2 \dashrightarrow s'_2$, though in the case of B-TREF, we may need to construct a new, different s'_2 .

Case B-PURESTEP: We have that $s_1 \dashrightarrow s'_1$ by whatever step was used in the hypothesis of B-PURESTEP. To prove the typing proofs correspond, we note that, for most possible steps, both s'_1 and s'_2 type-check by the same logic as in the proof of Theorem 6, meaning the typing proofs transform in the same way. The exception is E-ENDORSE. Here let $s_2 = \text{endorse } v_2$ from ℓ' to ℓ and consider two cases: if $v_2 = \bullet$ and if $v_2 \neq \bullet$. When $v_2 \neq \bullet$, the same argument as in Theorem 6 applies, and $s_1 = \text{endorse } v_1$ from ℓ' to ℓ follows the same step by the same argument. When $v_2 = \bullet$, inversion

on the typing rules gives us that $\ell' \not\approx \ell_t$. Because we know s_2 is endorsement-free at ℓ_t , this means $\ell \not\approx \ell_t$, so therefore $\Sigma; \Gamma \vdash \bullet : t^\ell$. Again, s_1 follows E-ENDORSE and the typing proofs correspond. For the heap correspondence and well-typed conditions, we note that the heaps and their types remain unchanged for both executions. To maintain endorsement-freedom at ℓ_t , most possible steps cannot add new terms, so they cannot add new endorse terms. E-CALL and E-CALLATK, however, can introduce new terms into s'_1 and s'_2 that may not have been present in s_1 and s_2 . Because CT is endorsement-free at ℓ_t , any new sub-statements of the form endorse v from ℓ' to ℓ must have the required property.

Case B-EVAL: In this case $s_2 = E_2[\tilde{s}_2]$ and $\tilde{s}_2 \rightarrow \tilde{s}'_2$. By inversion on $s_1 \geq_\bullet s_2$, it must be the case that $s_1 = E_1[\tilde{s}_1]$ where $\tilde{s}_1 \geq_\bullet \tilde{s}_2$. By inversion on the set of evaluation contexts, s_1 can only step through E-EVAL and no other steps. Therefore, by induction, on the \rightarrow relation, $\tilde{s}_1 \rightarrow \tilde{s}'_1$ with the required properties, so E-EVAL gives us everything except correspondence of the typing proof. We get that by noting that we can apply Lemma 5 in exactly the same way to both proofs.

Case B-BULLETCtx with $B = ![\cdot], (C)[\cdot],$ or $[\cdot].f$: Here we have that $s_2 = B[\bullet]$, so by inversion on $s_1 \geq_\bullet s_2$, we know that $s_1 = B[v_1]$ for some non-bullet value v_1 . By the fact that $s_1 \rightarrow^+ v$, we know that s_1 must step, so by inspection on the operational semantics, it must step with E-DEREF, E-CAST, or E-FIELD, depending on the syntactic form. In each case the result is a non-variable value v'_1 , so therefore $s'_1 = v'_1 \geq_\bullet \bullet = v'_2$ with typing proofs using VAL to get to a value typing judgment that allows them to differ on \bullet . The heap does not change.

Case B-BULLETCtx with $B = \text{if}\{pc'\}[\cdot]$ then e_1^2 else e_2^2 : First we note that $s'_2 = \bullet$ and $\sigma'_2 = \sigma_2$. We also note that inversion on the typing rules shows $\Sigma_{\sigma_2}; \Gamma \vdash \bullet : \text{bool}^\ell$ for some $\ell \not\approx \ell_t$ and $\ell \triangleleft \tau$, meaning BULLET gives us $\Sigma_{\sigma_2}; \Gamma \vdash \bullet : \tau$. We now examine s_1 and the corresponding steps. Because $\Sigma_{\sigma_1}; \Gamma; pc; \lambda_1 \vdash s_1 : \tau \dashv \lambda_0 \geq_\bullet \Sigma_{\sigma_2}; \Gamma; pc; \lambda_1 \vdash s_2 : \tau \dashv \lambda_0$, we know $s_1 = \text{if}\{pc'\} v_1$ then e_1^1 else e_2^1 . This syntactic structure means s_1 must step with one of E-IFT or E-IFF. Because we have assumed that $\langle s_1 \mid (CT, \sigma_1, \mathcal{M}, L) \rangle \rightarrow^+ \langle v \mid \mathcal{C} \rangle$, we further know that

$$\langle s_1 \mid (CT, \sigma_1, \mathcal{M}, L) \rangle \rightarrow \langle e_i^1 \text{ at-pc } pc' \mid (CT, \sigma_1, \mathcal{M}, L) \rangle \rightarrow^* \langle v \text{ at-pc } pc' \mid \mathcal{C} \rangle \rightarrow \langle v \mid \mathcal{C} \rangle.$$

By inspection on the semantic rules, we know that E-EVAL must apply in each of the steps in the middle segment, meaning $\langle e_i^1 \mid (CT, \sigma_1, \mathcal{M}, L) \rangle \rightarrow^* \langle v \mid \mathcal{C} \rangle$.

The correspondence of the typing proof with s_1 proves that $\Sigma_{\sigma_1}; \Gamma \vdash v_1 : \text{bool}^\ell$ for some $\ell \not\approx \ell_t$. Inversion on that typing rules therefore tells us $\ell \Rightarrow pc'$ and $\Sigma_{\sigma_1}; \Gamma; pc'; \lambda_1 \vdash e_i^1 : \tau \dashv \lambda_0$. By Corollary 1, $\mathcal{M}' = \mathcal{M}$ and $L' = L$, and $\sigma'_1 \approx_{\ell_t} \sigma_1 \approx_{\ell_t} \sigma_2 = \sigma'_2$. By inductively applying Theorem 6, we get $\sigma'_1 \supseteq \sigma_1$, so

$$\sigma'_2 = \sigma_2 \subseteq \sigma_1|_{\ell_t} \subseteq \sigma'_1|_{\ell_t}.$$

By letting $s'_1 = v$ and noting that all values are endorsement-free at ℓ_t , we complete the case.

Case B-BULLETCtx with $[\cdot].m(\bar{v})$: This case is very similar to the previous case. Again, $s'_2 = \bullet$ and $\sigma'_2 = \sigma_2$. Also, inversion on the typing rules gives us $\Sigma_{\sigma_2}; \Gamma \vdash \bullet : C^\ell$ for some $\ell \not\approx \ell_t$, and $\ell \triangleleft \tau$, again allowing BULLET to prove $\Sigma_{\sigma_2}; \Gamma \vdash \bullet : \tau$. We again turn to s_1 .

The typing correspondence now means $s_1 = v_1.m(\bar{w})$, so it must step using E-CALL or E-CALLATK. Therefore $v_1 = \text{new } C(\bar{w}')$ and $\text{mbody}(C, m) = (\ell_m, \bar{x}, \bar{\tau}_a, pc_1 \gg pc_2, e, \tau)$. Again, we know that it steps to a value, so now

$$\begin{aligned} \langle s_1 \mid (CT, \sigma_1, \mathcal{M}, L) \rangle &\rightarrow \langle (\text{return}_\tau e') \text{ at-pc } pc_2 \mid (CT, \sigma_1, (\mathcal{M}, \ell_m), L) \rangle \\ &\rightarrow^* \langle (\text{return}_\tau v) \text{ at-pc } pc_2 \mid (CT, \sigma'_1, \mathcal{M}'_1, L'_1) \rangle \end{aligned}$$

where $e' = e[\bar{x} \mapsto \bar{w}, \text{this} \mapsto \text{new } C(\bar{w}')] is an expression. Additionally, $\Sigma_{\sigma_1}; \Gamma; pc_2; \lambda'_1 \vdash e' : \tau \dashv \lambda'_0$. By the correspondence of the typing proofs of s_1 and s_2 , we know that $\Sigma_{\sigma_2}; \Gamma; pc; \lambda_1 \vdash \bullet.m(\bar{v}) : \tau \dashv$$

λ_0 interpreting \bullet as $\Sigma_{\sigma_2}; \Gamma \vdash \bullet : C^\ell$. Inversion on the typing rules therefore gives us that $\ell \not\Rightarrow \ell_t$ and $\ell \Rightarrow pc_1$. By transitivity, we know that $pc_1 \not\Rightarrow \ell_t$, so by the fact that CT is endorsement-free at ℓ_t , we have that $pc_2 \not\Rightarrow \ell_t$. Therefore, we can apply Corollary 1 to our above semantic steps, giving:

- $\sigma'_1 \approx_{\ell_t} \sigma_1 \approx_{\ell_t} \sigma_2 = \sigma'_2$,
- $\mathcal{M}'_1 = \mathcal{M}, \ell_m$, and
- $L'_1 = L$.

Again, Theorem 6's result tells us $\sigma'_1 \supseteq \sigma_1$, meaning $\sigma'_2 \subseteq \sigma'_1|_{\ell_t}^\bullet$. Applying E-ATPC and E-RETURN while letting $s'_1 = v$ completes the case.

Case B-TREF: Here we take the \tilde{s}'_2 from Lemma 10 as a candidate, which we may modify. In particular, we note that $s_2 = \text{ref } v_2 \tau'$, so by the typing correspondence, $s_1 = \text{ref } v_1 \tau'$. Therefore, s_1 must step using E-REF, giving $s'_1 = \iota$ for some $\iota \notin \text{dom}(\sigma_1)$ and $\sigma'_1 = \sigma_1[\iota \mapsto (v_1, \tau')]$.

For s'_2 , we know that $\text{dom}(\sigma_2) \subseteq \text{dom}(\sigma_1)$, so $\iota \notin \text{dom}(\sigma_2)$. We also know that s_2 could step using B-TREF, so we can use the same step, setting the location to ι . Therefore, $\sigma'_2 = \sigma_2[\iota \mapsto (v_2, \tau')]$. The fact that s'_2 is well-typed in $\Sigma_{\sigma'_2}$ follows directly from this extension.

Endorsement-freedom of s'_1 and s'_2 , typing correspondence, and that $\sigma'_1 \approx_{\ell_t} \sigma'_2$ are now straightforward. To show that $\sigma'_2 \subseteq \sigma'_1|_{\ell_t}^\bullet$, we note that the typing correspondence between v_1 and v_2 means that either $v_2 = v_1$ or $v_2 = \bullet$. The second case is impossible because $\text{label}(\tau') \Rightarrow \ell_t$, so inversion on the typing rules demonstrates $\Sigma_{\sigma_2}; \Gamma \not\vdash \bullet : \tau'$. With $v_2 = v_1$, the relation between σ'_1 and σ'_2 follows directly from their definitions and the corresponding relation between σ_1 and σ_2 .

Case B-UREF: Using the same logic as the previous case, s_1 must step using E-REF, and we can make $s'_2 = s'_1 = \iota$ for some $\iota \notin \text{dom}(\sigma_1) \supseteq \text{dom}(\sigma_2)$. We again have that s'_1 and s'_2 correspond and are well-typed and that $\sigma'_1 \approx_{\ell_t} \sigma'_2$. Finally, we note that, by assumption from B-UREF, $s_2 = \text{ref } v_2 \tau'$ where $\text{label}(\tau') \not\Rightarrow \ell_t$. Therefore, $\sigma'_1|_{\ell_t}^\bullet = \sigma_1[\iota \mapsto \tau']|_{\ell_t}^\bullet = \sigma_1|_{\ell_t}^\bullet[\iota \mapsto (\bullet, \tau')]$, and correspondingly, $\sigma'_2 = \sigma_2[\iota \mapsto (\bullet, \tau')]$. The correspondence follows from the correspondence between σ_1 and σ_2 .

Case B-BASSIGN: Here $s_2 = \bullet := v_2$, so $s_1 = \iota := v_1$ where $v_1 \geq_\bullet v_2$. By inversion on the typing rules and the \geq_\bullet relation, we know that $\Sigma_{\sigma_1}; \Gamma \vdash \iota : \text{ref } \tau'^\ell$ and $\Sigma_{\sigma_2}; \Gamma \vdash \bullet : \text{ref } \tau'^\ell$. Moreover, we know that $\ell \not\Rightarrow \ell_t$ and $\ell \triangleleft \tau$. Since s_1 steps, it must step with E-ASSIGN, meaning $\sigma'_1 = \sigma_1[\iota \mapsto (v_1, \tau')]$. Therefore $\sigma'_1|_{\ell_t}^\bullet = \sigma_1|_{\ell_t}^\bullet \supseteq \sigma_2 = \sigma'_2$. Letting $s'_1 = s'_2 = ()$ completes the case.

Case B-TASSIGN: This is similar to the B-TREF case, but we do not need to construct a new location, as $s_2 = \iota := v$. We also know by the same logic as in that case that $v \neq \bullet$, so $s_1 = s_2$. B-TASSIGN and E-ASSIGN produce precisely the same output on the same input, proving the case.

Case B-UASSIGN: Here we note that $s_2 = \iota := v_2$, meaning $s_1 = \iota := v_1$ where $v_1 \geq_\bullet v_2$ and $i \in \text{dom}(\sigma_2)$. By inversion on the typing rules, we know that $\Sigma_{\sigma_i}(\iota) = \tau'$ for both $i = 1, 2$ and some τ' where $\text{label}(\tau') \not\Rightarrow \ell_t$. Therefore, s_1 steps using E-ASSIGN, so $\sigma'_1 = \sigma_1[\iota \mapsto (v_1, \tau')]$ where $\sigma_1(\iota) = (v, \tau')$ for some v . As a result, $\sigma'_2 = \sigma_2 \subseteq \sigma_1|_{\ell_t}^\bullet = \sigma'_1|_{\ell_t}^\bullet$. The two steps result in $s'_1 = s'_2 = ()$, so the statements type-check with corresponding rules. \square

Corollary 2. For any class table CT , heap type Σ , and expression (not statement) e , if

- $\Sigma \vdash CT$ ok is endorsement-free at ℓ_t ,
- e is endorsement-free at ℓ_t ,
- $\Sigma \subseteq \Sigma_{\sigma_1}$,
- $\Sigma; \Gamma; pc; \lambda_I \vdash e : \tau \dashv \lambda_0$,
- $\vdash \sigma_1$ wt, and
- $\langle e \mid (CT, \sigma_1, \mathcal{M}, L) \rangle \longrightarrow^* \langle v \mid (CT, \sigma'_1, \mathcal{M}', L') \rangle$,

then there is some value v' , heap σ'_2 , and heap type Σ'_2 such that

- $\langle e \mid (CT, \sigma_1|_{\ell_t}^\bullet, \mathcal{M}, L) \rangle \dashrightarrow^* \langle v' \mid (CT, \sigma'_2, \mathcal{M}', L') \rangle$ and
- $\sigma'_1 \simeq_{\ell_t} \sigma'_2$.

Proof. This proof follows from Lemma 11 and induction on the number of steps, letting $s_1 = s_2 = e$ and $\sigma_2 = \sigma_1|_{\ell_t}^\bullet$ to start. If there are zero steps—that is $e = s_1 = v$ —then we are done. Otherwise Lemma 11 allows us to step s_2 once using \dashrightarrow and provides a corresponding set of steps using \longrightarrow for s_1 . The result may have differently-named locations from the original, but Theorem 5 allows us to continue stepping a location-name isomorphic expression. The steps therefore maintain all requirements to apply Lemma 11 again until s_1 reaches a value. At that point, we are assured $\sigma'_2 \subseteq \sigma''_1|_{\ell_t}^\bullet$ and $\sigma'_1 \approx_{\ell_t} \sigma''_1$ for some $\sigma''_1 \simeq \sigma'_1$. Therefore $\sigma'_1 \simeq_{\ell_t} \sigma'_2$. \square

Theorem 1 (Noninterference). *Let CT be a class table where $\Sigma \vdash CT \text{ ok}$ is endorsement-free at ℓ_t . For any well-typed heaps σ_1 and σ_2 such that $\Sigma \subseteq \Sigma_{\sigma_i}$ and any invocation I such that $\Sigma \vdash I$ and $(I, CT, \sigma_i) \Downarrow \sigma'_i$, if $\sigma_1 \simeq_{\ell_t} \sigma_2$, then $\sigma'_1 \simeq_{\ell_t} \sigma'_2$.*

Proof. First we note that since $\Sigma \subseteq \Sigma_{\sigma_i}$, Lemma 4 means $\Sigma_{\sigma_i} \vdash CT \text{ ok}$ for both $i = 1, 2$, meaning our various lemmas apply in both cases. Without loss of generality, we assume $\sigma_1 \approx_{\ell_t} \sigma_2$, since we can permute the location names in one to match the other and permute the results back later. There exists a unique

$$\tilde{\sigma} = \sigma_1|_{\ell_t}^\bullet = \sigma_2|_{\ell_t}^\bullet$$

Let $I = (\iota, m(\bar{v}), \ell)$. Note that $! \iota.m(\bar{v})$ is an expression with no endorse statements and $\Sigma \vdash ! \iota.m(\bar{v}) : \tau$ for some τ . Therefore, by Corollary 2,

$$\langle ! \iota.m(\bar{v}) \mid (CT, \tilde{\sigma}, \ell, \cdot) \rangle \dashrightarrow^* \langle v \mid (CT, \tilde{\sigma}', \ell, \cdot) \rangle$$

where $\tilde{\sigma}' \simeq_{\ell_t} \sigma'_i$ for both $i = 1, 2$. Transitivity of \simeq_{ℓ_t} then proves $\sigma'_1 \simeq_{\ell_t} \sigma'_2$. \square

E Proof of Reentrancy Security

We now prove Theorem 2. As discussed in Section 6.3, we do this by first proving Theorem 3 saying all reentrancy is tail-reentrancy and Theorem 4 that says tail reentrancy is secure according on Definition 9.

E.1 SeRIF Allows Only Tail Reentrancy

We start by proving Theorem 3. We prove this theorem using the general formulation of “trusted” and “untrusted” labels. In particular, we partition \mathcal{L} into a downward-closed sublattice \mathcal{T} and the attacker-controlled labels $\mathcal{A} = \overline{\mathcal{T}}$. Notationally, we will use ℓ_t to denote some trusted label ($\ell_t \in \mathcal{T}$), rather than a distinguished one. We refer to code complying with locks in \mathcal{T} -code, to mean it complies with locks in ℓ_t -code for all $\ell_t \in \mathcal{T}$.

Finally, we will prove the result for two adversarial models: one where E-CALLATK is admissible and \mathcal{A} is a sublattice, and the other where E-CALLATK is *not* admissible, but \mathcal{A} has no restrictions beyond $\mathcal{A} = \overline{\mathcal{T}}$. These two proofs are extremely similar. Indeed, they differ only in a single case of Lemma 15 and Lemma 16 on which it relies. We will specifically call out the differences when they arise.

The proof follows the following general structure. First we show that high-integrity code maintains all of the input locks λ_I it claims to and the operational semantics maintain all dynamic locks. Second, we will show that, if a statement that complies with locks steps to an auto-endorse call, it cannot comply with a lock on any label that call endorses through (i.e., one that does not trust pc_1 but does trust pc_2). Finally, we connect these to show that, for low-integrity call from a high-integrity context that proceeds to make a reentrant call, the original low-integrity call must have been in tail position from the original high-integrity execution.

To discuss the security of an invocation mid-evaluation, we need to discuss the security of a statement s with respect to locks. We do this using several different tools. First, we extend our notion of lock compliance in \mathcal{T} -code to statements. We do this with a judgment $pc \vdash_{\mathcal{T}} s \text{ cwl}$. The nontrivial rules are as follows.

$$\begin{array}{c}
\frac{pc \vdash_{\mathcal{T}} e_1 \text{ cwl} \quad pc \vdash_{\mathcal{T}} e_2 \text{ cwl}}{pc \vdash_{\mathcal{T}} (\text{if}\{pc'\} v \text{ then } e_1 \text{ else } e_2) \text{ cwl}} \quad \frac{pc \vdash_{\mathcal{T}} e \text{ cwl}}{pc \vdash_{\mathcal{T}} (\text{lock } \ell \text{ in } e) \text{ cwl}} \quad \frac{pc \vdash_{\mathcal{T}} s \text{ cwl}}{pc \vdash_{\mathcal{T}} (s \text{ with-lock } \ell) \text{ cwl}} \\
\\
\frac{pc \vdash_{\mathcal{T}} s \text{ cwl} \quad pc \vdash_{\mathcal{T}} e \text{ cwl}}{pc \vdash_{\mathcal{T}} (\text{let } x = s \text{ in } e) \text{ cwl}} \quad \frac{pc \vdash_{\mathcal{T}} s \text{ cwl}}{pc \vdash_{\mathcal{T}} (\text{return}_{\tau} s) \text{ cwl}} \\
\\
\frac{pc' \vdash_{\mathcal{T}} s \text{ cwl}}{pc \vdash_{\mathcal{T}} (s \text{ at-pc } pc') \text{ cwl}} \quad \frac{pc \vdash_{\mathcal{T}} s \text{ cwl} \quad pc \notin \mathcal{T}}{pc \vdash_{\mathcal{T}} (\text{ignore-locks-in } s) \text{ cwl}}
\end{array}$$

If s has none of the syntactic forms in the rules defined above, then $pc \vdash_{\mathcal{T}} s \text{ cwl}$ for any pc and \mathcal{T} . Note that, because at-pc terms are statements but not expressions, for any expression e , $pc \vdash_{\mathcal{T}} e \text{ cwl}$ if one of two conditions holds: either $pc \notin \mathcal{T}$ or e has no subexpressions of the form ignore-locks-in e' . As a result, we can also specify our definition of lock compliance from class tables using this judgment. Specifically,

$$CT \text{ complies with locks in } \mathcal{T}\text{-code} \iff \frac{CT(C) = \text{class } C[\ell_C] \text{ extends } D \{ \bar{f}; \bar{\tau}_f; K; \bar{M} \} \quad \tau m\{pc_1 \gg pc_2; \lambda_0\}(\bar{x}; \bar{\tau}_a) \{e\} \in \bar{M}}{\ell_C \vdash_{\mathcal{T}} e \text{ cwl}} \text{ is admissible for } CT$$

Lemma 12. *If $pc \vdash_{\mathcal{T}} s \text{ cwl}$ and $pc \Rightarrow pc'$, then $pc' \vdash_{\mathcal{T}} s \text{ cwl}$.*

Proof. By simple induction on the definition of $pc \vdash_{\mathcal{T}} s \text{ cwl}$. □

We will also be considering statements in the middle of evaluation, so we need a way to extract the pc label that we expect sub-statements to type-check with, and similarly we need to extract the list of dynamic locks that will be present when a sub-statement completes executing. We do that using the following two recursive functions defined on evaluation contexts.

$$\begin{array}{l}
getLocks(L, E) = \begin{cases} L & \text{if } E = [\cdot] \\ getLocks((L, \ell), E') & \text{if } E = E' \text{ with-lock } \ell \\ getLocks(L, E') & \text{if } E = \text{let } x = E' \text{ in } e, \text{return}_{\tau} E', \text{ or } E' \text{ at-pc } pc \end{cases} \\
innerPc(pc, E) = \begin{cases} pc & \text{if } E = [\cdot] \\ innerPc(pc', E') & \text{if } E = E' \text{ at-pc } pc' \\ innerPc(pc, E') & \text{if } E = \text{let } x = E' \text{ in } e, \text{return}_{\tau} E', \text{ or } E' \text{ with-lock } \ell \end{cases}
\end{array}$$

We extend both of these to statements by $getLocks(L, E[e]) = getLocks(L, E)$, and similarly for $innerPc$.

Definition 12 (Configuration Safety). A statement-configuration pair $\langle s \mid (CT, \sigma, \mathcal{M}, L) \rangle$ is \mathcal{T} -safe with pc and \hat{L} if

1. $\Sigma_{\sigma} \vdash CT$ ok complies with locks in \mathcal{T} -code,
2. $\vdash \sigma$ wt,
3. $\Sigma_{\sigma}; \Gamma; pc; \lambda_I \vdash s : \tau \dashv \lambda_O$,
4. $pc \vdash_{\mathcal{T}} s \text{ cwl}$,
5. $L = getLocks(\hat{L}, s)$, and
6. for any E and s' where $E \neq E'[\text{return}_{\tau} [\cdot]]$ and $pc' = innerPc(pc, E) \in \mathcal{T}$, if $s = E[s']$ then there is some λ'_I such that $\Sigma_{\sigma}; \Gamma'; pc'; \lambda'_I \vdash s' : \tau' \dashv \lambda'_O$ and $(\bigwedge getLocks(\hat{L}, E)) \wedge \lambda'_I \Rightarrow pc'$.

Lemma 13. *If $\langle E[s] \mid \mathcal{C} \rangle$ is \mathcal{T} -safe at pc and L , then $\langle s \mid \mathcal{C} \rangle$ is \mathcal{T} -safe at $innerPc(pc, E)$ and $getLocks(L, E)$.*

Proof. By induction on E and the definitions of $innerPc$ and $getLocks$. \square

Lemma 14 (Preservation of \mathcal{T} -Safety). *If $\langle s \mid \mathcal{C} \rangle$ is \mathcal{T} -safe with pc and \hat{L} , and $\langle s \mid \mathcal{C} \rangle \longrightarrow \langle s' \mid \mathcal{C}' \rangle$, then $\langle s' \mid \mathcal{C}' \rangle$ is \mathcal{T} -safe with pc and \hat{L} .*

Proof. Condition 1 follows from Lemma 3 and the fact that CT must remain unchanged. Conditions 2 and 3 follow directly from Theorem 6. We prove the other three conditions by induction on the operational semantics. Notationally, we let $\mathcal{C} = (CT, \sigma, \mathcal{M}, L)$ and $\mathcal{C}' = (CT, \sigma', \mathcal{M}', L')$. Also, by assumption, there is some λ'_1 such that $\Sigma_\sigma; \Gamma; pc; \lambda'_1 \vdash s : \tau \dashv \lambda_0$ and $(\bigwedge \hat{L}) \wedge \lambda'_1 \Rightarrow pc$. We assume without loss of generality that λ_1 has this property.

Case E-EVAL: In this case $s = E[\tilde{s}]$, $\langle \tilde{s} \mid \mathcal{C} \rangle \longrightarrow \langle \tilde{s}' \mid \mathcal{C}' \rangle$, and $s' = E[\tilde{s}']$. Let $pc' = innerPc(pc, E)$ and $\hat{L}' = getLocks(\hat{L}, E)$. By Lemma 13, we know that \tilde{s} is \mathcal{T} -safe at pc' and \hat{L}' , so by induction on the operational semantics, $\langle \tilde{s}' \mid \mathcal{C}' \rangle$ is as well. We also note that $\Sigma_\sigma; \Gamma'; pc'; \lambda'_1 \vdash \tilde{s} : \tau' \dashv \lambda'_0$.

By the safety of $\langle s \mid \mathcal{C} \rangle$, for every pair of sub-contexts E_1 and E_2 such that $E = E_1[E_2]$, either $\tilde{pc} = innerPc(pc, E_1) \notin T$, $E_1 = E'_1[\text{return}_\tau[\cdot]]$, or $\Sigma_\sigma; \tilde{\Gamma}; \tilde{pc}; \tilde{\lambda}_1 \vdash E_2[\tilde{s}] : \tau_1 \dashv \tilde{\lambda}_0$ for some $\tilde{\lambda}_1$ where $(\bigwedge getLocks(\hat{L}, E_1)) \wedge \tilde{\lambda}_1 \Rightarrow \tilde{pc}$. By Theorem 6, $\Sigma_\sigma; \Gamma'; pc'; \lambda'_1 \vdash \tilde{s}' : \tau' \dashv \lambda'_0$, so by Lemma 5, we also have that $\Sigma_\sigma; \tilde{\Gamma}; \tilde{pc}; \tilde{\lambda}_1 \vdash E_2[\tilde{s}'] : \tau_1 \dashv \tilde{\lambda}_0$. As this holds for every choice of E_1 and E_2 , this proves the case.

Cases E-IFT and E-IFF: In both cases we have $s = \text{if}\{pc'\} v$ then e_1 else e_2 and $s' = e_i$ at- pc pc' for either $i = 1$ or 2 . By inversion on the typing rules, $pc \Rightarrow pc'$ and $\Sigma_\sigma; \Gamma; pc'; \lambda_1 \vdash e_i : \tau \dashv \lambda_0$ for both $i = 1, 2$. Moreover, by Lemma 12, $pc' \vdash_{\mathcal{T}} e_i$ cwl, so Condition 4 holds for s' . Because e_1 and e_2 are expressions, we know that if $e_i = E[\tilde{s}]$, then E consists entirely of let and ignore-locks-in statements and \tilde{s} is an expression. Therefore, if $pc' \notin T$, then Condition 6 is trivial.

If $pc' \in T$, then because \mathcal{T} is downward-closed, $pc \in T$. Because $pc \vdash_{\mathcal{T}} e_i$ cwl, ignore-locks-in cannot appear in e in this sub-case, so E consists entirely of let statements. As a result, $\Sigma_\sigma; \Gamma'; pc'; \lambda_1 \vdash \tilde{s} : \tau' \dashv \lambda_0$ for some $\Gamma' \supseteq \Gamma$ and τ' . Because $(\bigwedge \hat{L}) \wedge \lambda_1 \Rightarrow pc \Rightarrow pc'$, this proves that $\langle e_i \mid \mathcal{C}' \rangle$ is \mathcal{T} -safe at pc' and \hat{L} . Since $s' = e_i$ at- pc pc' , the \mathcal{T} -safety transfers to $\langle s' \mid \mathcal{C}' \rangle$.

Case E-LET: Here $s = (\text{let } x = v \text{ in } e)$ and $s' = e[x \mapsto v]$. Theorem 6 proves $\Sigma_\sigma; \Gamma; pc; \lambda_1 \vdash s' : \tau \dashv \lambda_0$. Moreover, because s' is an expression, by the same logic as in the previous case, $\langle s' \mid \mathcal{C}' \rangle$ must be \mathcal{T} -safe at pc and \hat{L} .

Case E-LOCK: Here $s = \text{lock } \ell \text{ in } e$. First we note that $L' = (L, \ell) = (\hat{L}, \ell) = getLocks(\hat{L}, e \text{ with-lock } \ell)$, as is required by Condition 5.

Next, inversion on the typing rules tells that $\Sigma_\sigma; \Gamma; pc; \lambda'_1 \vdash e : \tau \dashv \lambda'_0$ where $\lambda'_1 \wedge \ell \Rightarrow \lambda_1$ and $\lambda'_0 \wedge \ell \Rightarrow \lambda_0$. Further, we know that $s' = e \text{ with-lock } \ell$, $L = \hat{L}$, and $L' = (L, \ell)$. By Condition 4 on s , first Condition 4 holds trivially on s' , and second, either $pc \in T$ or e contains no ignore-locks-in terms, as e is an expression. Therefore, by the same logic as in the previous two cases, it suffices to show Condition 6 holds when $pc \in T$ and $E = [\cdot]$ with-lock ℓ . Here we know that $\Sigma_\sigma; \Gamma; pc; \lambda'_1 \vdash e : \tau \dashv \lambda'_0$ with λ'_1 defined as above. As a result,

$$\begin{aligned} \left(\bigwedge getLocks(\hat{L}, E) \right) \wedge \lambda'_1 &= \left(\bigwedge (\hat{L}, \ell) \right) \wedge \lambda'_1 \\ &= \left(\bigwedge \hat{L} \right) \wedge \ell \wedge \lambda'_1 \\ &\Rightarrow \left(\bigwedge \hat{L} \right) \wedge \lambda_1 \\ &\Rightarrow pc. \end{aligned}$$

Case E-UNLOCK: Here $s = v$ with-lock ℓ and $s' = v$, so Condition 4 is trivial. Condition 5 follows from the semantic rule that requires $L = (L', \ell)$, so if $getLocks(\hat{L}, s) = L$, then $\hat{L} = L' = getLocks(\hat{L}, v)$. Condition 6 follows from the fact that values type-check with any λ_I , including pc .

Cases E-CALL and E-CALLATK: Here $s = new\ C(\bar{v}).m(\bar{w})$. If we let $mbody(C, m) = (\ell_m, \bar{x}, \bar{\tau}_a, pc_1 \gg pc_2, e, \tau)$, then $s' = (return_\tau\ e')$ at- $pc\ pc_2$ where $e' = e[\bar{x} \mapsto \bar{w}, this \mapsto new\ C(\bar{v})]$. By Condition 1 on s , we know that $\ell_C \vdash_{\mathcal{T}} e'$ cwl. The METHOD-OK rule requires that $\ell_C \Rightarrow pc_2$, so therefore by Lemma 12 proves $pc_2 \vdash_{\mathcal{T}} e'$ cwl, proving Condition 4.

Since the body of the method is an expression and $L = L'$, Condition 5 holds trivially.

For Condition 6, we consider multiple possible evaluation contexts E . If $E = [\cdot]$, note that $return_\tau\ s''$ type-checks with any λ_I . If $E = return_\tau\ [\cdot]$, then this is precisely the caveat that Condition 6 does not restrict. If $E = return_\tau\ [\cdot]$ at- $pc\ pc_2$, METHOD-OK ensures that $\Sigma_\sigma; \cdot; pc_2; \lambda'_I \vdash e' : \tau \dashv \lambda'_O$ for some $\lambda'_I \Rightarrow pc_2$. In particular, this means $(\bigwedge \hat{L}) \wedge \lambda'_I \Rightarrow pc_2$ regardless of the contents of \hat{L} . Moreover, because we know that $pc_2 \vdash_{\mathcal{T}} e'$ cwl and e' is an expression, either $pc_2 \notin \mathcal{T}$, in which case Condition 6 is trivial in e' , or $pc_2 \in \mathcal{T}$ and e' does not contain ignore-locks-in terms. In the second case, the same logic as in several previous cases completes the proof that Condition 6 holds, and thus the case.

In all other cases the step leaves L unchanged and produces a value. All well-typed value type check with any λ_I and $pc \vdash_{\mathcal{T}} v$ cwl for any label pc and value v , so all conditions hold. \square

Lemma 15. For any label $\ell_t \in \mathcal{T}$, statement s , configuration $\mathcal{C} = (CT, \sigma, \mathcal{M}, L)$, lock list \hat{L} , if

1. $\langle s \mid \mathcal{C} \rangle$ is \mathcal{T} -safe with pc and \hat{L} for some label pc ,
2. s contains no sub-statements of the form ignore-locks-in s' ,
3. $\langle s \mid \mathcal{C} \rangle \longrightarrow^* \langle E[new\ C(\bar{v}).m(\bar{w})] \mid \mathcal{C}' \rangle \longrightarrow \langle s' \mid \mathcal{C}'' \rangle$, and

4. $mtype(C, m) = \bar{\tau}_a \xrightarrow{pc_1 \gg pc_2; \lambda'_O} \hat{\tau}$ with $pc_1 \not\Rightarrow \ell_t$ and $pc_2 \Rightarrow \ell_t$,

then for any λ_I and λ_O such that $\Sigma_\sigma; \Gamma; pc; \lambda_I \vdash s : \tau \dashv \lambda_O$, then $(\bigwedge \hat{L}) \wedge (\lambda_I \vee \lambda_O) \not\Rightarrow \ell_t$.

Proof. This is a proof by induction on the number of steps in premise 3. For the base case of zero steps, $s = E[new\ C(\bar{v}).m(\bar{w})]$. We prove this case by induction on E . For these cases, we will use the notational short-hand $s' = E'[new\ C(\bar{v}).m(\bar{w})]$ where E' will be defined in each inductive case.

Case $E = [\cdot]$: The \mathcal{T} -safety of $\langle s \mid \mathcal{C} \rangle$ with pc and \hat{L} ensures $\Sigma_\sigma; \Gamma; pc; \lambda_I \vdash new\ C(\bar{v}).m(\bar{w}) : \tau \dashv \lambda_O$. By inversion on the typing rules, we know that $pc_1 \Rightarrow pc_2 \vee \lambda_I$. We also know that this expression steps again, so by inversion on the operational semantics, it must step using E-CALL, meaning $\bigwedge_{\ell \in L} (pc_1 \Rightarrow pc_2 \vee \ell)$. Therefore, by the fact that \wedge produces the greatest lower bound and the distributive property of the lattice,

$$pc_1 \Rightarrow \bigwedge_{\ell \in L} (pc_2 \vee \ell) \wedge (pc_2 \vee \lambda_I) = pc_2 \vee \left(\left(\bigwedge L \right) \wedge \lambda_I \right).$$

Moreover, because $pc_1 \not\Rightarrow \ell_t$, transitivity of \Rightarrow tells us that this label does not act for ℓ_t . Yet $pc_2 \Rightarrow \ell_t$, so by the definition of join, it must be the case that $((\bigwedge L) \wedge \lambda_I) \not\Rightarrow \ell_t$. Because $\lambda_I \Rightarrow \lambda_I \vee \lambda_O$, and $L = \hat{L}$ in this case, transitivity of \Rightarrow and equality substitution proves $(\bigwedge \hat{L}) \wedge (\lambda_I \vee \lambda_O) \not\Rightarrow \ell_t$, as desired.

Case $E = let\ x = E'$ in e : By inversion on the typing rules, we note that $\Sigma_\sigma; \Gamma; pc; \lambda_I \vdash s' : \tau' \dashv \lambda'_O$ where $\lambda'_O \Rightarrow \lambda_I$. Premises 1 and 2 are clearly true for $\langle s' \mid \mathcal{C} \rangle$, so by induction on E , $(\bigwedge \hat{L}) \wedge (\lambda_I \vee \lambda'_O) \not\Rightarrow \ell_t$. Since $\lambda'_O \Rightarrow \lambda_I$, we know that $\lambda_I \vee \lambda'_O = \lambda_I \Rightarrow \lambda_I \vee \lambda_O$. Transitivity of \Rightarrow then proves the desired result.

Case $E = return_\tau\ E'$: Here inversion on the typing rules tells us $\Sigma_\sigma; \cdot; pc; \lambda'_I \vdash s' : \tau \dashv \lambda'_O$ for some λ'_I and λ'_O where $\lambda'_I \vee \lambda'_O \Rightarrow \lambda_O$. As with the previous case, our inductive hypothesis on E applies, giving us $(\bigwedge \hat{L}) \wedge (\lambda'_I \vee \lambda'_O) \not\Rightarrow \ell_t$. Since $\lambda'_I \vee \lambda'_O \Rightarrow \lambda_O \Rightarrow \lambda_I \vee \lambda_O$, transitivity of \Rightarrow again gives us the desired result.

Case $E = E'$ with-lock ℓ : Here inversion on the typing rules tells us $\Sigma_\sigma; \Gamma; pc; \lambda'_1 \vdash s' : \tau \dashv \lambda'_0$ where $\lambda'_1 \wedge \ell \Rightarrow \lambda_1$ and $\lambda'_0 \wedge \ell \Rightarrow \lambda_0$. By the definition of *getLocks*, we know that $\langle s' \mid \mathcal{C} \rangle$ must be \mathcal{T} -safe with pc and (\hat{L}, ℓ) . Premise 2 is clearly true of s' since we have not added new syntax, so induction on E tells us $(\bigwedge(\hat{L}, \ell)) \wedge (\lambda'_1 \vee \lambda'_0) \not\Rightarrow \ell_t$. Using the above facts and the distributive property of the lattice,

$$\begin{aligned} \left(\bigwedge(\hat{L}, \ell) \right) \wedge (\lambda'_1 \vee \lambda'_0) &= \left(\bigwedge \hat{L} \right) \wedge \ell \wedge (\lambda'_1 \vee \lambda'_0) \\ &= \left(\bigwedge \hat{L} \right) \wedge ((\lambda'_1 \wedge \ell) \vee (\lambda'_0 \wedge \ell)) \\ &\Rightarrow \left(\bigwedge \hat{L} \right) \wedge (\lambda_1 \vee \lambda_0). \end{aligned}$$

Transitivity of \Rightarrow finishes the case.

Case $E = E'$ at-pc pc' : Here $\langle s' \mid \mathcal{C} \rangle$ is \mathcal{T} -safe at pc' and \hat{L} and premise 2 clearly holds, so induction on E proves the case.

Case $E = \text{ignore-locks-in } E'$: This case is impossible by assumption 2.

We now move on to the inductive case on the number of steps. For all cases, Lemma 14 ensures that premise 1 remains true after a single step. By inspection on the operational semantics, we can introduce ignore-locks-in terms in only two ways: directly through E-CALL and E-CALLATK and indirectly through E-EVAL. Thus premise 2 inductively holds for all other steps. Similarly, premises 3 and 4 remain true by assumption at top-level. We can therefore directly apply our inductive hypothesis for all steps except E-EVAL, E-CALL, and E-CALLATK. We handle those cases explicitly.

For the case of E-EVAL where $s = \tilde{E}[\tilde{s}]$, we induct on \tilde{E} and the operational semantics.

Case $\tilde{E} = [\cdot]$: Here induction on the operational semantic rule proves the case.

Case $\tilde{E} = \text{let } x = \tilde{E}' \text{ in } e$: We now consider two sub-cases: if $\langle \tilde{E}'[\tilde{s}] \mid \mathcal{C} \rangle \longrightarrow^* \langle \tilde{E}''[\text{new } C(\bar{v}).m(\bar{w})] \mid \mathcal{C}' \rangle$ or not. If there is such an evaluation, then all of the inductive hypotheses hold for $\tilde{E}'[\tilde{s}]$, so induction on \tilde{E} prove the case. If there is no such evaluation, inspection on the operational semantics tells us that we can only step s using E-EVAL stepping $\tilde{E}'[\tilde{s}]$ until it steps to a value. Therefore, premise 3, ensures that there is some value v and context \mathcal{C}_v such that $\langle \tilde{E}'[\tilde{s}] \mid \mathcal{C} \rangle \longrightarrow^+ \langle v \mid \mathcal{C}_v \rangle$. Using E-EVAL on each step gives us

$$\langle s \mid \mathcal{C} \rangle \longrightarrow^+ \langle \text{let } x = v \text{ in } e \mid \mathcal{C}_v \rangle \longrightarrow^* \langle E[\text{new } C(\bar{v}).m(\bar{w})] \mid \mathcal{C}' \rangle \longrightarrow \langle \tilde{s} \mid \mathcal{C}'' \rangle.$$

Therefore, $\langle \text{let } x = v \text{ in } e \mid \mathcal{C}_v \rangle$ satisfies our inductive hypothesis, so induction completes the case.

For the other three possible cases of \tilde{E} , the same logic as in the base-case proof above applies.

We now turn to when the step is E-CALL or E-CALLATK. In both cases $s = \text{new } D(\bar{v}').m'(\bar{w}')$ and $mbody(D, m') = (\ell_m, \bar{x}, \bar{\tau}'_a, pc'_1 \gg pc'_2, e, \tau)$. If we let $e' = e[\bar{x} \mapsto \bar{w}', \text{this} \mapsto \text{new } D(\bar{v}')]$, this steps to $(\text{return}_\tau e')$ at-pc pc'_2 . We handle this in two sub-cases: if $pc'_2 \in \mathcal{T}$ and if $pc'_2 \notin \mathcal{T}$.

If $pc'_2 \in \mathcal{T}$, then METHOD-OK proves that $\ell_m \Rightarrow pc'_2$ and therefore $\ell_m \in \mathcal{T}$. Because $\langle s \mid \mathcal{C} \rangle$ is \mathcal{T} -safe, the method body e , and hence e' , cannot have any subexpressions of the form ignore-locks-in e'' . Therefore the new statement satisfies premise 2 of this lemma, allowing us to apply the inductive hypothesis.

If $pc'_2 \notin \mathcal{T}$, we claim that $(\bigwedge \hat{L}) \wedge pc'_2 \not\Rightarrow \ell_t$, which we prove differently based on the security assumptions of the system: either $\mathcal{A} = \overline{\mathcal{T}}$ is a sublattice, or E-CALLATK is not admissible. In both cases we will apply Lemma 16 to the configuration after taking this step. To meet the requirement of hte lemma that there is no sub-statement of the form s' at-pc pc' , we use $\langle e' \mid (CT, \sigma, (\mathcal{M}, \ell_m), L) \rangle$, noting that this configuration is \mathcal{T} -safe with pc'_2 and \hat{L} .

Because $pc'_2 \notin \mathcal{T}$, Lemma 16 proves that $(\bigwedge \hat{L}) \in \mathcal{A}$. When \mathcal{A} is a sublattice, it is closed under join, so $(\bigwedge \hat{L}) \wedge pc'_2 \in \mathcal{A}$. By the downward-closed property of \mathcal{T} , that means $(\bigwedge \hat{L}) \wedge pc'_2 \not\Rightarrow \ell_t$.

If E-CALLATK is not admissible, Lemma 16 proves that $pc'_2 \Rightarrow pc_2 \vee (\bigwedge \hat{L})$. By the definition of meet and the distributive property of the lattice,

$$pc'_2 = \left(pc_2 \vee \left(\bigwedge \hat{L} \right) \right) \wedge pc'_2 = (pc_2 \wedge pc'_2) \vee \left(\left(\bigwedge \hat{L} \right) \wedge pc'_2 \right).$$

By assumption on this sub-case, $pc'_2 \notin \mathcal{T}$ and therefore $pc'_2 \not\Rightarrow \ell_t$, so at least one of the two sides of the join cannot act for ℓ_t . However, the definition of meet gives $pc_2 \wedge pc'_2 \Rightarrow pc_2 \Rightarrow \ell_t$. Therefore $(\bigwedge \hat{L}) \wedge pc'_2 \not\Rightarrow \ell_t$.

By inversion on the typing rules, if $\Sigma_\sigma; \Gamma; pc; \lambda_I \vdash s : \tau \dashv \lambda_o$, then $\lambda'_o \vee pc'_2 \Rightarrow \lambda_o$ where λ'_o is the lock label on $D.m$. In particular, $pc'_2 \Rightarrow \lambda_o \Rightarrow \lambda_I \vee \lambda_o$. As a result, $(\bigwedge \hat{L}) \wedge pc'_2 \Rightarrow (\bigwedge \hat{L}) \wedge (\lambda_I \vee \lambda_o)$, so transitivity of \Rightarrow proves $(\bigwedge \hat{L}) \wedge (\lambda_I \vee \lambda_o) \not\Rightarrow \ell_t$. \square

Lemma 16. For any statement s , configuration $\mathcal{C} = (CT, \sigma, \mathcal{M}, L)$, label pc , and lock list \hat{L} , if

- $\langle s \mid \mathcal{C} \rangle$ is \mathcal{T} -safe with pc and \hat{L} ,
- s contains no sub-statements of the form s' at- pc pc' ,
- $\langle s \mid \mathcal{C} \rangle \longrightarrow^* \langle E[\text{new } C(\bar{v}).m(\bar{w})] \mid \mathcal{C}' \rangle \longrightarrow \langle s' \mid \mathcal{C}'' \rangle$, and
- $mtype(C, m) = \bar{\tau}_a \xrightarrow{pc_1 \gg pc_2; \lambda_o} \tau$ with $pc_2 \in \mathcal{T}$,

then $pc \notin \mathcal{T}$ implies $(\bigwedge \hat{L}) \notin \mathcal{T}$, and $pc \Rightarrow pc_2 \vee (\bigwedge \hat{L})$ if no step uses E-CALLATK.

Proof. This proof follows by induction on the number of steps. For the base case where $s = E[\text{new } C(\bar{v}).m(\bar{w})]$, we induct on E to prove $pc \Rightarrow pc_2 \vee (\bigwedge \hat{L})$.

Case $E = [\cdot]$: Inversion on the typing rules tells us $pc \Rightarrow pc_1$ and inversion on the operational semantics tell us $pc_1 \Rightarrow pc_2 \vee (\bigwedge L)$. By the definition of $getLocks$, $\hat{L} = L$, so transitivity proves the case.

Case $E = E$ at- pc pc' : This case is impossible by assumption.

Case $E = E'$ with-lock ℓ : In this case we note that if $s = E[\tilde{s}]$, then $\langle E'[\tilde{s}] \mid \mathcal{C} \rangle$ must be \mathcal{T} -safe with pc and (\hat{L}, ℓ) . Therefore, by induction on E , $pc \Rightarrow pc_2 \vee (\bigwedge (\hat{L}, \ell))$. However,

$$\bigwedge (\hat{L}, \ell) = \left(\bigwedge \hat{L} \right) \wedge \ell \Rightarrow \bigwedge \hat{L}.$$

Therefore, by transitivity, $pc \Rightarrow pc_2 \vee (\bigwedge \hat{L})$.

All other cases: The pc remains unmodified and $getLocks(\hat{L}, E'[\tilde{s}]) = getLocks(\hat{L}, E[\tilde{s}]) = L$, so a simple inductive application completes the case.

This directly proves the second conclusion in this case. When $pc \notin \mathcal{T}$, the fact that \mathcal{T} is downward-closed means $pc_2 \vee (\bigwedge \hat{L}) \notin \mathcal{T}$. However, $pc_2 \in \mathcal{T}$ and \mathcal{T} is a sublattice, so therefore it must be the case that $(\bigwedge \hat{L}) \notin \mathcal{T}$.

We now move to the inductive step. Lemma 14 ensures that \mathcal{T} -safety is retained. By inspection on the operational semantic rules, we can introduce new syntax only with E-EVAL, E-IFT, E-IFF, E-CALL, and E-CALLATK. For all other steps, a direct application of the inductive hypothesis proves the lemma. We now prove those cases.

Case E-EVAL: This case is by induction on \tilde{E} where $s = \tilde{E}[\tilde{s}]$. If $\tilde{E} = [\cdot]$, induction on the operational semantics completes the case. When $\tilde{E} = \text{let } x = \tilde{E}' \text{ in } e$, we must consider whether $\tilde{E}'[\tilde{s}]$ steps to the relevant method call or not. If it does, a direct inductive application proves the case. If it does not, we note that $\langle \tilde{E}'[\tilde{s}] \mid \mathcal{C} \rangle \longrightarrow^+ \langle v \mid \mathcal{C}_v \rangle$ for some value v and configuration \mathcal{C}_v . This new expression satisfies the premises of our top-level inductive hypothesis, so we can apply that.

By assumption, $\tilde{E} \neq \tilde{E}'$ at- pc pc' , and the other possible options are the same as in the base case.

Cases E-IFT and E-IFF: In this case we note that $s = \text{if}\{pc'\} v$ then e_1 else e_2 . Inversion on the typing rules proves that $\Sigma_\sigma; \Gamma; pc'; \lambda_I \vdash e_i : \tau \dashv \lambda_o$ for both $i = 1, 2$ and $pc \Rightarrow pc'$. Therefore, $\langle e_i \mid \mathcal{C} \rangle$ is \mathcal{T} -safe with pc' and \hat{L} . Moreover, e_1 and e_2 are expressions, so they contain no sub-statements of

the form s'' at-pc pc'' , allowing us to apply our inductive hypothesis. If $pc \notin \mathcal{T}$, then because \mathcal{T} is downward-closed, $pc' \notin \mathcal{T}$, so induction proves that $(\bigwedge \hat{L}) \notin \mathcal{T}$. If E-CALLATK is not admissible, induction proves $pc' \Rightarrow pc_2 \vee (\bigwedge \hat{L})$, so transitivity gets us the desired result.

Case E-CALL: In this case $s = \text{new } D(\bar{v}').m'(\bar{w}')$ with $\text{mbody}(D, m') = (\ell_{m'}, \bar{x}, \bar{\tau}'_a, pc'_1 \gg pc'_2, e, \tau')$, and $pc'_1 \Rightarrow pc'_2 \vee (\bigwedge \hat{L})$. Inversion on the typing rules proves that $pc \Rightarrow pc'_1$. Additionally, the statement after the step is $\text{return}_{\tau'}(e' \text{ at-pc } pc'_2)$ for some expression e' .

By Lemma 14 the new configuration is \mathcal{T} -safe at pc and \hat{L} , so inductively, replacing the statement with e' is \mathcal{T} -safe at pc'_2 and \hat{L} . If $pc'_2 \notin \mathcal{T}$, then by induction $(\bigwedge \hat{L}) \notin \mathcal{T}$. If $pc \notin \mathcal{T}$ but $pc'_2 \in \mathcal{T}$, then the same logic as in the base case proves $(\bigwedge \hat{L}) \notin \mathcal{T}$.

If E-CALLATK is never used, induction on the number of steps proves $pc'_2 \Rightarrow pc_2 \vee (\bigwedge \hat{L})$. Combining this with the flow above, we get

$$pc \Rightarrow pc'_2 \vee (\bigwedge \hat{L}) \Rightarrow (pc_2 \vee (\bigwedge \hat{L})) \vee (\bigwedge \hat{L}) = pc_2 \vee (\bigwedge \hat{L}). \quad \square$$

Case E-CALLATK: Inversion on the semantic rules proves $pc'_2 \in \mathcal{A} = \overline{\mathcal{T}}$. Using the same argument as in the E-CALL case to apply the inductive hypothesis, induction proves that $(\bigwedge \hat{L}) \notin \mathcal{T}$ regardless of the value of pc . This case is impossible by assumption when E-CALLATK is not taken.

We formalize the concept of a tail call, which is a call initiated in a tail position of some expression, by defining a *tail context* T which, by construction, does nothing after the call returns.

Definition 13 (Tail Context).

$$T ::= [\cdot] \mid \text{return}_{\tau} T \mid T \text{ with-lock } \ell \mid T \text{ at-pc } pc$$

The following lemma captures our intuition that a tail context “does nothing”.

Lemma 17. *If $\langle T[v] \mid (CT, \sigma, \mathcal{M}, L) \rangle \longrightarrow \langle s \mid (CT, \sigma', \mathcal{M}', L') \rangle$, then for some tail context T' , $s = T'[v]$ and $\sigma = \sigma'$.*

Proof. By simple induction on the operational semantics, noting for E-EVAL that, if $T[v] = E[s']$, then $E = T_1$ and $s' = T_2[v]$ for some tail contexts T_1 and T_2 . \square

Definition 14 (Tail Reentrancy). We say a statement s is in an ℓ_t -tail-reentrant state if s is ℓ_t -reentrant—that is, $s = E_0[E_1[E_2[s' \text{ at-pc } pc_3] \text{ at-pc } pc_2] \text{ at-pc } pc_1]$ where $pc_1, pc_3 \Rightarrow \ell_t$ and $pc_2 \not\Rightarrow \ell_t$ —and there is some tail context T , evaluation context \tilde{E}_2 , and label pc'_2 such that $pc'_2 \not\Rightarrow \ell_t$ and

$$E_1[[\cdot] \text{ at-pc } pc_2] = T[\tilde{E}_2 \text{ at-pc } pc'_2]$$

Theorem 3. *For any label $\ell_t \in \mathcal{T}$, class table CT , and well-typed heap σ_1 , if $\Sigma_{\sigma_1} \vdash CT$ ok complies with locks in \mathcal{T} -code, then for any invocation I and heap σ_2 where $\Sigma_{\sigma_1} \vdash I$ and $(I, CT, \sigma_1) \Downarrow \sigma_2$, all ℓ_t -reentrant states in the execution are ℓ_t -tail-reentrant.*

Proof. By Definition 5, if $I = (\iota, m(\bar{v}), \ell)$ is an ℓ_t -reentrant invocation in σ_1 , there must exist a statement s such that

$$s = E_0 \left[E_1 \left[E_2[s' \text{ at-pc } pc_3] \text{ at-pc } pc_2 \right] \text{ at-pc } pc_1 \right]$$

where $pc_1, pc_3 \Rightarrow \ell_t$ but $pc_2 \not\Rightarrow \ell_t$, and $\langle \iota.m(\bar{v}) \mid (CT, \sigma_1, \ell, \cdot) \rangle \longrightarrow^* \langle s \mid \mathcal{C} \rangle$. We prove by induction on E_1 that s is ℓ_t -tail-reentrant according to Definition 14. Specifically, we claim the following.

Claim. *If $s = E'_0[E_1[s'' \text{ at-pc } pc_2]]$ where $\text{innerPc}(\ell, E'_0) \Rightarrow \ell_t$, then there is some \tilde{E}_2 , T , and pc'_2 such that $E_1[[\cdot] \text{ at-pc } pc_2] = T[\tilde{E}_2 \text{ at-pc } pc'_2]$ and $pc'_2 \not\Rightarrow \ell_t$.*

Proof of claim. This is a proof by induction on E_1 .

Case $E_1 = [\cdot]$: Because $pc_2 \not\Rightarrow \ell_t$ by assumption, letting $pc'_2 = pc_2$, $\tilde{E}_2 = [\cdot]$, and $T = [\cdot]$ proves the case.

Case $E_1 = E'_1$ at-pc pc' : There are two sub-cases to consider. If $pc' \Rightarrow \ell_t$, then the inductive hypothesis applies by replacing E'_0 with $E'_0[[\cdot]$ at-pc pc'] and E_1 by E'_1 . It then proves that $E'_1 = T'[\tilde{E}_2$ at-pc $pc'_2]$ for some $pc'_2 \not\Rightarrow \ell_t$. Letting $T = T'$ at-pc pc' completes the sub-case.

If $pc' \not\Rightarrow \ell_t$, then letting $\tilde{E}_2 = E'_1$, $pc'_2 = pc'$, and $T = [\cdot]$ proves the case.

Case $E_1 = E'_1$ with-lock ℓ : Replacing E'_0 with $E'_0[[\cdot]$ with-lock $\ell]$ and E_1 with E'_1 , the inductive hypothesis proves $E'_1 = T'[\tilde{E}_2$ at-pc $pc'_2]$ for some $pc'_2 \not\Rightarrow \ell_t$. Letting $T = T'$ with-lock ℓ completes the case.

Case $E_1 = \text{return}_\tau E'_1$: This case follows from the same logic as the previous case.

Case $E_1 = (\text{let } x = E'_1 \text{ in } e)$: Let $pc = \text{innerPc}(\ell, E'_0)$. Lemma 14 and induction on the number of steps to get to s , proves that if $\langle !\iota.m(\bar{v}) \mid (CT, \sigma_1, \ell, \cdot) \rangle \longrightarrow^* \langle s \mid (CT, \sigma, \mathcal{M}, L) \rangle$ then it must be the case that each configuration encountered along the way is ℓ_t -safe with ℓ and \cdot .

To step to s , there must be some expression e_1 such that

$$\begin{aligned} \langle !\iota.m(\bar{v}) \mid (CT, \sigma_1, \ell, \cdot) \rangle &\longrightarrow^* \langle E'_0[\text{let } x = e_1 \text{ in } e] \mid (CT, \sigma', \mathcal{M}, L) \rangle \\ &\text{and} \\ \langle e_1 \mid (CT, \sigma', \mathcal{M}, L) \rangle &\longrightarrow^* \langle E[\text{new } D(\bar{v}').m'(\bar{w})] \mid C' \rangle \end{aligned}$$

where $mtype(D, m') = \bar{\tau}_a \xrightarrow{\tilde{pc}_1 \gg \tilde{pc}_2; \lambda_o} \tilde{\tau}$ such that $\tilde{pc}_1 \not\Rightarrow \ell_t$ and $\tilde{pc}_2 \Rightarrow \ell_t$. Inversion on the typing rules and the safety of $\langle E'_0[\text{let } x = e_1 \text{ in } e] \mid (CT, \sigma', \mathcal{M}, L) \rangle$ prove that $\Sigma; \Gamma; pc; \lambda_I \vdash e_1 : \tau_1 \dashv \lambda_o$ for some $\Sigma, \Gamma, \lambda_I, \tau_1$, and λ_o , where $\lambda_o \Rightarrow \lambda_I$ and $(\bigwedge \text{getLocks}(\cdot, E'_0)) \wedge \lambda_I \Rightarrow pc \Rightarrow \ell_t$. Moreover, the safety of the configuration guarantees that $\text{getLocks}(\cdot, E'_0)$ is a prefix of L , so in particular, $\bigwedge L \Rightarrow \bigwedge \text{getLocks}(\cdot, E'_0)$.

However, Lemma 15 mandates that, since $\Sigma; \Gamma; pc; \lambda_I \vdash e_1 : \tau_1 \dashv \lambda_o$, $(\bigwedge L) \wedge (\lambda_I \vee \lambda_o) \not\Rightarrow \ell_t$. Yet we know already that $\lambda_o \Rightarrow \lambda_I$, meaning $\lambda_I \vee \lambda_o = \lambda_I$, and $\bigwedge L \Rightarrow \bigwedge \text{getLocks}(\cdot, E'_0)$. Therefore, this proves that $(\bigwedge \text{getLocks}(\cdot, E'_0)) \wedge \lambda_I \not\Rightarrow \ell_t$. This contradicts the safety result, so this case is impossible.

Case $E_1 = \text{ignore-locks-in } E'_1$: Safety of the configuration, as argued in the previous case, proves that $\ell \vdash_{\mathcal{T}} s$ cwl. Because, by assumption, $\text{innerPc}(\ell, E'_0) \Rightarrow \ell_t$, inversion on the proof rules for $\ell \vdash_{\mathcal{T}} s$ cwl demonstrates that this case is impossible. \square

Letting $E'_0 = E_0[[\cdot]$ at-pc $pc_1]$ clearly satisfies the assumptions of the claim. Therefore,

$$s = E_0[T[\tilde{E}_2[s'']] \text{ at-pc } pc'_2] \text{ at-pc } pc_1]$$

for some $pc'_2 \not\Rightarrow \ell_t$. This form satisfies Definition 14 and proves the theorem. \square

E.2 All Tail Reentrancy is Secure

We now present a proof for Theorem 4, proving that all tail reentrancy is secure. The proof follows the structure outlined in the proof sketch in Section 6.3. It requires one simple lemma and follows essentially as a corollary from a more complicated statement.

Lemma 18. *For any type τ and heap-type Σ , there exists a value v such that $\Sigma \vdash v : \tau$.*

Proof. This proof is by induction on the structure of τ . If $\tau = \text{unit}^\ell$, $v = ()$. If $\tau = \text{bool}^\ell$, $v = \text{true}$. If $\tau = (\text{ref } \tau')^\ell$, $v = \text{null}$. If $\tau = C^\ell$, let $\text{fields}(C) = \bar{x}:\bar{\tau}$. For each τ_i , by induction, there is some v_i such that $\Sigma \vdash v_i : \tau_i$. Therefore, by NEW, $\Sigma \vdash \text{new } C(\bar{v}) : C^\ell$. \square

For the main proof, we assume the existence of an nat type and constant nat values. This assumption is without loss of generality as natural numbers are simple to encode using objects. The class simply has isZero and previous methods. There are two implementations: zero returns true and this, respectively, while non-zero values have a single field pointing to the previous nat and return false and the value of their one field. We will only use nat to increment and check the value, each of which is simple with this implementation.

Lemma 19. *For any class table CT , invocation I , and heaps σ_1 and σ_2 , if*

- $\Sigma_{\sigma_1} \vdash CT$ ok *complies with locks in ℓ -code,*
- $\vdash \sigma_1$ wt,
- $\Sigma_{\sigma_1} \vdash I$, *and*
- $(I, CT, \sigma_1) \Downarrow \sigma_2$ *where all ℓ -reentrant states are ℓ -tail-reentrant,*

then there exist CT' , \bar{I} , σ'_1 , and σ'_2 such that

1. $\Sigma_{\sigma'_1} \vdash CT'$ ok *complies with locks in ℓ -code,*
2. $CT \approx_{\ell} CT'$,
3. $\vdash \sigma'_1$ wt,
4. $\Sigma_{\sigma'_1} \vdash \bar{I}$,
5. $(\bar{I}, CT', \sigma'_1) \Downarrow \sigma'_2$ *are all non- ℓ -reentrant, and*
6. $\sigma_i \approx_{\ell} \sigma'_i$ *with $\sigma_i \subseteq \sigma'_i$ for both $i = 1, 2$.*

Proof. For notation, let $I = (\ell_I, \iota_I, m_I(\overline{v_I}))$.

Step through the execution of $(I, CT, \sigma_1) \Downarrow \sigma_2$ and create a log of the following relevant events:

1. Calls from low-integrity environments into high-integrity environments.
2. Calls from high-integrity environments into low-integrity environments.
3. Returns from low-integrity environments into high-integrity environments.
4. State modifications from low-integrity environments.

For most events, we will only need to reply the event later, so logging the type of event and the statement that is evaluated is sufficient. For event 2, however, CT' will need to have different code than CT , so there must be a link to the original piece of code. Method calls already have a name and clear location in the code, but if statements can also move from high-integrity to low-integrity and have no names. To allow for unique tracking, we attach a unique name a to each branch of each conditional statement in CT . They have the same typing and semantic rules as before, but syntactically include this new annotation, denoted $\text{if}\{pc\} v \text{ then}_{a_1} e_1 \text{ else}_{a_2} e_2$.

For a semantic step $\langle s \mid (CT, \sigma, \mathcal{M}, L) \rangle \longrightarrow \langle s' \mid (CT, \sigma', \mathcal{M}', L') \rangle$, let $pc_s = \text{innerPc}(\ell_I, s)$ and $pc_{s'} = \text{innerPc}(\ell_I, s')$. The following formally defines when each type of event is emitted.

1. When $s = E[\text{new } C(\overline{v}).m(\overline{w})]$ and $mtype(C, m) = \overline{\tau}_a \xrightarrow{pc_1 \gg pc_2; \lambda_0} \tau$, if $pc_s \not\Rightarrow \ell$ and $pc_2 \Rightarrow \ell$, emit $\text{up}(pc_s, \text{new } C(\overline{v}).m(\overline{w}), \sigma)$.
2. - When $s = E[\text{new } C(\overline{v}).m(\overline{w})]$ and $mtype(C, m) = \overline{\tau}_a \xrightarrow{pc_1 \gg pc_2; \lambda_0} \tau$, if $pc_s \Rightarrow \ell$ but $pc_2 \not\Rightarrow \ell$, emit $\text{down}(pc_2, C.m)$.
- When $s = E[\text{if}\{pc\} v \text{ then}_{a_1} e_1 \text{ else}_{a_2} e_2]$, if $pc_s \Rightarrow \ell$ but $pc \not\Rightarrow \ell$, emit $\text{down}(pc, a_1)$ if $v = \text{true}$ and $\text{down}(pc, a_2)$ if $v = \text{false}$.
3. When $\sigma' = \sigma[\iota \mapsto (v, \tau)] \neq \sigma$, emit $\text{set}(\iota \mapsto (v, \tau))$.
4. When $s = E[v \text{ at-pc } pc_s]$, if $pc_s \not\Rightarrow \ell$ and $pc_{s'} \Rightarrow \ell$, emit $\text{ret}(v)$.

By inspection on the operational semantics, each step will emit at most one of the above events.

There are several important properties to note about the log. First, the only semantic steps that can change the value of $\text{innerPc}(\ell_I, s)$ are E-CALL, E-CALLATK, E-IFT, E-IFF, and E-ATPC. Type preservation (Theorem 6) ensures that each statement is well-typed, so if statements can only lower the integrity of the

pc , not raise it. Therefore, whenever $pc_s \Rightarrow \ell$ and $pc_{s'} \not\Rightarrow \ell$, the log will contain a down event, and whenever $pc_s \not\Rightarrow \ell$ and $pc_{s'} \Rightarrow \ell$, the log will contain either a up event or ret event. As a result, any two down events must be separated by either an up event or a ret event.

Additionally, the down and ret events must follow a stack discipline as the represent calls and returns. This stack discipline creates a correspondence between each down and exactly one ret, which we will refer to as the “corresponding ret” event.

We now use the log constructed from the execution of $(I, CT, \sigma_1) \Downarrow \sigma_2$ to construct CT' , \bar{I} , and σ'_1 . We will ensure by construction that all conditions hold aside from Condition 6 with σ_2 and σ'_2 . We will then argue Condition 6 on σ_2 holds.

Constructing CT' , \bar{I} , and σ'_1 . Initialize $\bar{I} = I$ if $\ell_I \Rightarrow \ell$ and empty otherwise, and initialize $\sigma'_1 = \sigma$ and $CT' = CT$. We will add to \bar{I} and σ'_1 and modify CT' as the construction progresses.

Step through the log. When a $\text{down}(pc, a)$ event appears, where a can either be $C.m$ or a unique name for the branch of an if statement, note that the log must be of the form $\dots, \text{down}, \overline{\text{set}}, ev, \dots$ where ev is either up or ret. If this is the first down event at location a , add a new mapping $\iota_a \mapsto (0, \text{nat}^{pc})$ to σ'_1 where ι_a is fresh, meaning $\iota_a \notin \text{dom}(\sigma'_1) \cup \text{dom}(\sigma_2)$. Also modify the code at location a in CT' . If this is the first time encountering a , replace the existing code with code that increments ι_a and conditions on it. If this is the n th $\text{down}(pc, a)$ event in the log for $n > 1$, add a new branch to the code in CT' for if $\iota_a \mapsto n$.

The code in the conditional branch for $\iota_a \mapsto n$ will do different things depending on ev . If $ev = \text{ret}(v)$, the code in CT' performs all state modification in $\overline{\text{set}}$ and then returns v . Making these state modification may require constructing new low-integrity methods if pc does not have sufficient integrity for each. Since we know that none of the modified cells are trusted by ℓ , however, making the modifications is always possible using low-integrity code. Moreover, because the state modifications were possible in the original execution without violating locks or entering high-integrity code (there was no up prior to $ev = \text{ret}(v)$), a call graph with the same pc labels where $pc \not\Rightarrow \ell$ for each label must be possible. This guarantees that CT' continues to type-check.

If ev is an up event and this is the n th $\text{down}(pc, a)$ event for location a , then the n th entry into a in CT' simply returns some value v of the appropriate type. By Lemma 18, some such well-typed v must exist.

When a $\text{up}(pc, \text{new } C(\bar{v}).m(\bar{w}), \sigma)$ event appears in the log, modify both σ'_1 and \bar{I} . For σ'_1 , add a mapping $\iota \mapsto (\text{new } C(\bar{v}), C^{pc})$ for a fresh location $\iota \notin \text{dom}(\sigma'_1) \cup \text{dom}(\sigma_2)$. For \bar{I} , add two new invocations. The first performs all state modifications from all set events in the log prior to this up that have not already been performed by a previous invocation. As before, constructing such an invocation may require adding new low-integrity code to CT' . The second invocation added to \bar{I} is $(pc, \iota, m(\bar{w}))$ where ι is the new location added to σ'_1 .

Finally, after completing all up and down events in the log, include one final invocation with associated new code to apply any set events not included in any previous invocations.

The construction satisfies all requirements. By construction, the resulting invocations \bar{I} are non-reentrant in CT' with initial state σ'_1 . All code changes in CT' were low-integrity and remained well-typed, so $\Sigma_{\sigma'_1} \vdash CT' \text{ ok}$ complies with locks in ℓ -code and $CT \approx_{\ell} CT'$. We constructed σ'_1 by adding new well-typed low-integrity mappings to σ_1 , meaning $\vdash \sigma'_1 \text{ wt}$, $\sigma_1 \approx_{\ell} \sigma'_1$, and $\sigma_1 \subseteq \sigma'_1$, as desired. It remains to show that there is a σ'_2 such that $(\bar{I}, CT', \sigma'_1) \Downarrow \sigma'_2$ with $\sigma_2 \approx_{\ell} \sigma'_2$ and $\sigma_2 \subseteq \sigma'_2$.

Let $\tilde{\sigma}_1, \dots, \tilde{\sigma}_n$ be the sequence of heaps appearing in the up events in the log. Let I_1, \dots, I_n be the elements of \bar{I} that call into high-integrity code (note that these are every other element of \bar{I}), and let $\tilde{\sigma}'_k$ be the heap provided as input to I_k when executing $(\bar{I}, CT', \sigma'_1) \Downarrow \sigma'_2$. We now argue by induction on k that $\tilde{\sigma}_k \approx_{\ell} \tilde{\sigma}'_k$ and $\tilde{\sigma}_k \subseteq \tilde{\sigma}'_k$.

For the base case let $k = 1$. There are two sub-cases to consider: if $\ell_I \Rightarrow \ell$ and if it does not. If $\ell_I \Rightarrow \ell$, then $I_1 = I$ and there are no elements of \bar{I} before it, so $\tilde{\sigma}_1 = \sigma_1$ and $\tilde{\sigma}'_1 = \sigma'_1$, meaning the conditions on σ_1 and σ'_1 proved above are precisely the goal. If $\ell_I \not\Rightarrow \ell$, there is one invocation I_0 in \bar{I} before I_1 ,

and it executes only low-integrity code to set mappings. By construction, the code invoked by I_0 performs exactly the modifications to σ'_1 that occurred to σ_1 prior to the up event in the original invocation. Note that some of these modifications may be adding new mappings through using E-REF, which is non-deterministic. Because all mappings in σ'_1 not in σ_1 were taken to be fresh with respect to σ_2 as well, the names used in the original invocation must be free, so we can pick the same names when evaluating to $\tilde{\sigma}'_1$. Therefore, for some set of mappings $\bar{\tau} \mapsto (\bar{v}, \bar{\tau})$, $\tilde{\sigma}_1 = \sigma_1[\bar{\tau} \mapsto (\bar{v}, \bar{\tau})]$ and $\tilde{\sigma}'_1 = \sigma'_1[\bar{\tau} \mapsto (\bar{v}, \bar{\tau})]$. Since $\sigma_1 \approx_\ell \sigma'_1$ and $\sigma_1 \subseteq \sigma'_1$, the same must therefore be true of $\tilde{\sigma}_1$ and $\tilde{\sigma}'_1$, as desired.

Now assume $k > 1$ and, by induction, that $\tilde{\sigma}_{k-1} \approx_\ell \tilde{\sigma}'_{k-1}$ with $\tilde{\sigma}_{k-1} \subseteq \tilde{\sigma}'_{k-1}$. There are two sub-cases to consider depending on whether or not k th up event stems from a ℓ -reentrant call inside the call resulting in the $(k - 1)$ st up event.

If I_k does *not* correspond to a reentrant call, then I_{k-1} corresponds to a high-integrity call that executed to completion without reentrancy in the original execution. By construction of CT' , any part of that execution that operated at low-integrity corresponds to a down in the log, and since none of those produced any high-integrity calls (that would cause reentrancy), they modified the state by incrementing new low-integrity counters and otherwise making the same modifications and returning the same values as the original execution. In particular, the changes to $\tilde{\sigma}'_{k-1}$ needed to achieve the state $\hat{\sigma}'$ after completing I_{k-1} , are updates to new low-integrity counters and the changes to $\tilde{\sigma}_{k-1}$ to achieve the state $\hat{\sigma}$ after completing the original high-integrity call. Because $\tilde{\sigma}_{k-1} \approx_\ell \tilde{\sigma}'_{k-1}$ and $\tilde{\sigma}_{k-1} \subseteq \tilde{\sigma}'_{k-1}$, it must be that $\hat{\sigma} \approx_\ell \hat{\sigma}'$ and $\hat{\sigma} \subseteq \hat{\sigma}'$.

Further, any state modifications made after the high-integrity call returns (and thus after I_{k-1} completes) but before the k th up event (the beginning of I_k) must be made in a low-integrity environment. By the same logic as Lemma 9 from the proof of Noninterference, they must be updates to low-integrity state. As a result, each has a corresponding set event in the log, which we denote $\text{set}(\bar{\tau} \mapsto (\bar{v}, \bar{\tau}))$. The extra low-integrity invocation added to \bar{T} before I_k makes exactly these modifications to the state. Therefore, $\tilde{\sigma}_k = \hat{\sigma}[\bar{\tau} \mapsto (\bar{v}, \bar{\tau})]$ and $\tilde{\sigma}'_k = \hat{\sigma}'[\bar{\tau} \mapsto (\bar{v}, \bar{\tau})]$. The desired result follows from the above-proved correspondence of $\hat{\sigma}$ and $\hat{\sigma}'$.

Lastly, consider the case where I_k corresponds to a reentrant call inside the call that I_{k-1} corresponds to. That is, the log must have the form $\dots, \text{up}_{k-1}, \overline{ev}, \text{down}(pc, a), \overline{\text{set}}, \text{up}_k, \dots$ where \overline{ev} contains no up events. In this case, the code created to replace the $\text{down}(pc, a)$ event in CT' simply returns an arbitrary value of the correct type without modifying the state. Because we assumed all reentrancy was tail-reentrancy, this means up_k occurred when stepping a term of the form

$$E_0[T[E_2[\text{new } C(\bar{v}).m(\bar{w})] \text{ at-pc } pc_2] \text{ at-pc } pc_1]$$

where $pc_1 \Rightarrow \ell$, $pc_2 \not\Rightarrow \ell$, and $mtype(C, m) = \bar{\tau}_a \xrightarrow{pc'_2 \gg pc_3; \lambda_0} \tau$ with $pc_3 \Rightarrow \ell$.

In CT' , we replaced the code corresponding to $E_2[\text{new } C(\bar{v}).m(\bar{w})]$ with code that returns an arbitrary value of the correct type, and splitting the invocations means inside I_{k-1} , E_0 will be empty. Therefore, by Lemma 17, once $E_2[\text{new } C(\bar{v}).m(\bar{w})]$ evaluates to some value v , $T[v]$ will evaluate to v with no changes to the state. Similarly, I_{k-1} will return the arbitrary value returned in CT' without examining it or modifying the state at all. That means that the change from $\tilde{\sigma}'_{k-1}$ to $\hat{\sigma}'$, the heap when I_{k-1} returns, is, as before, updates to new low-integrity counters coupled with exactly the change from $\tilde{\sigma}_{k-1}$ to the heap $\hat{\sigma}$ when the $\text{down}(pc, a)$ event occurred. The low-integrity state modifications in the extra invocation before I_k are again those made by the low-integrity code in CT before the call corresponding to $\text{up}(pc', \text{new } C(\bar{v}).m(\bar{w}), \tilde{\sigma}_k)$. By the same argument as before, $\tilde{\sigma}_k \approx_\ell \tilde{\sigma}'_k$ and $\tilde{\sigma}_k \subseteq \tilde{\sigma}'_k$, as desired.

We have now shown that the state before each I_k is a ℓ -equivalent superset of the state before the corresponding call in the original execution. To see that this result extends to σ_2 and σ'_2 , note that the logic above for non-reentrant calls applies to show that the state after completing I_n is a ℓ -equivalent superset of the state after completing the call that generated the final up event in the original execution. There may be further low-integrity code in the original execution that modifies the state, but all such modifications

generate set events and are updated by the final invocation in \bar{I} as described above. Therefore, again, σ_2 and σ'_2 are acquired by making identical modifications to the heap after the return of the final high-integrity call, thereby proving $\sigma_2 \approx_\ell \sigma'_2$ and $\sigma_2 \subseteq \sigma'_2$. \square

Theorem 4. *Let CT be a class table, σ_1 and σ_2 be well-typed heaps, and I be an invocation such that $(I, CT, \sigma_1) \Downarrow \sigma_2$ where all ℓ -reentrant states are ℓ -tail-reentrant. For any ℓ -integrity predicates P and Q , if $\Sigma_{\sigma_1} \models_\ell^1 \{P\} CT \{Q\}$ and $P(\sigma_1)$, then $Q(\sigma_2)$.*

Proof. Lemma 19 proves that there exists CT' , \bar{I} , σ'_1 , and σ'_2 with the properties stated in the lemma. Because P is a ℓ -integrity predicate and $\sigma_1 \approx_\ell \sigma'_1$, the assumption that $P(\sigma_1)$ means $P(\sigma'_1)$. The definition of $\Sigma_{\sigma_1} \models_\ell^1 \{P\} CT \{Q\}$, coupled with $CT \approx_\ell CT'$ and $\Sigma_{\sigma_1} \subseteq \Sigma_{\sigma'_1}$ mean that since $P(\sigma'_1)$ holds, $Q(\sigma'_2)$ must hold. Finally, since $\sigma_2 \approx_\ell \sigma'_2$, the fact that Q is also a ℓ -integrity predicate proves $Q(\sigma_2)$, as desired. \square