Compositional Security for Reentrant Applications

Ethan Cecchetti
Cornell University
ethan@cs.cornell.edu

Siqiu Yao
Cornell University
yaosiqiu@cs.cornell.edu

Haobin Ni
Cornell University
haobin@cs.cornell.edu

Andrew C. Myers
Cornell University
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.**

***Index Terms*—information flow control, language-based security, integrity, smart contracts**

I. 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 [44, 45, 47], but other instances exist. JavaScript code is difficult to secure when running on the same web page as code from a different source [14, 28, 39]. 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 [19, 57]. 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 [47]. Since then, a variety of methods have emerged to analyze or eliminate reentrancy attacks [4, 15, 18, 27, 37], but vulnerabilities continue to appear. For example, a January 2019 audit uncovered a reentrancy vulnerability in the Uniswap decentralized exchange [16]. 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 [45], 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 [21]. 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, 27, 37] 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 [15, 18], while others provide only heuristic reentrancy protection [9, 51, 52]. 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 II show the complexity of reentrancy.
- Section III provides background on information flow control and exposes its failure to handle reentrancy.
- Section IV presents a new definition of security in the presence of reentrancy.
- Section V defines SeRIF, a core calculus that eliminates insecure reentrancy by combining a static IFC type system with a dynamic locking mechanism.
- Section VI shows formally that SeRIF enforces our formal, compositional security condition.
- Section VII describes a prototype type checker implementation and our experience using it on realistic programs.
- Section VIII discusses related work in more detail and Section IX concludes.

II. 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 [17], despite being one 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.

A. Uniswap

We begin with the Uniswap/Lendf.me reentrancy vulnerability first identified in January 2019 [16] and later exploited in April 2020 [45]. 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

```
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
19     ... // check and update balances
20     from.alertSend(to, amount);
21     to.alertReceive(from, amount);
22     return true;
23   }
24 }
```

Fig. 1. Distilled Solidity [55] code for the Uniswap bug.

`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 [56]. 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.
2) Uniswap correctly computes `prod` $= 36$ and `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 `prod` $= 72$, not 36, and `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

```
getOrCompute(key, computeFun) {
    i = _getIdx(key) // index of mapping if it exists
    if (mappings[i] == null) {
        mappings[i] = computeFun();
    }
    return mappings[i];
}
```

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

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* `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 [18] can express the assumption of approach (1), which it mandates to statically eliminate all reentrancy. Other tools either cannot properly secure the application [9, 51, 52] or force the use of computationally expensive dynamic locks even when they are unnecessary [4, 15].

B. 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 [43, 49]. 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` 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 [43].

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.

```
contract TownCrier {
  address[] requesters, callbacks;

  function deliver(uint reqId, bytes data) {
    if (msg.sender == SERVICE_ADDR
        && requesters[reqId] != 0) {
      requesters[reqId] = 0;
      SERVICE_ADDR.call{value: FEE}("");
      callbacks[reqId].call(bytes);
    }
  }

  function cancel(uint reqId) {
    if (msg.sender == requesters[reqId]) {
      requesters[reqId] = 0;
      msg.sender.call{value: FEE}("");
    }
  }
}
```

Fig. 3. Solidity [55] code for simplified partial Town Crier contract. Here `SERVICE_ADDR` is TC's trusted wallet address, and `FEE` is the request fee.

C. Town Crier

Banning all reentrancy might seem appealing, but this solution would be overly restrictive. Town Crier (TC) [65] 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.

III. 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 [24]. Most IFC work has focused on data confidentiality [50, 59], but IFC can also protect integrity [8, 61] and availability [66]. 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 [50]. 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.

A. 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 "$\ell_1$ *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 [50]. 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

$$\ell_1 \vee \ell_2 \Rightarrow \ell \iff \ell_1 \Rightarrow \ell \text{ and } \ell_2 \Rightarrow \ell$$
$$\ell \Rightarrow \ell_1 \wedge \ell_2 \iff \ell \Rightarrow \ell_1 \text{ and } \ell \Rightarrow \ell_2.$$

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 V-B. Luckily, existing label models are typically distributive lattices, including two-point lattices, subset lattices of permissions [62], and free distributive lattices over a set of principals [6, 40]. In smart-contract systems, for example, it is natural to view contracts themselves as principals with different trust relationships among them. We might then employ *decentralized* information flow control [41] where labels are constructed from principals (e.g., contracts) that can influence data or computation.

B. 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* [67], 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 [38] 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 [20, 33, 36, 62] 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.

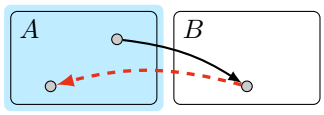
IV. Reentrancy and Security

The examples in Section II 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.

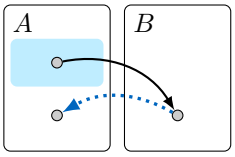
A. Defining Reentrancy

Prior work [4, 15, 27, 37] 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

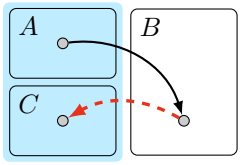
[1] Most IFC systems use *flows-to*, denoted $\sqsubseteq$. We use acts-for as we find it intuitive, and the two mean the same thing when only tracking integrity.



(a) Object reentrancy and ℓ-reentrancy are the same when object and trust boundaries match.



(b) Partially-trusted objects can create object reentrancy that is not ℓ-reentrancy.



(c) Mutually-trusting objects can create ℓ-reentrancy that is not object reentrancy.

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

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.

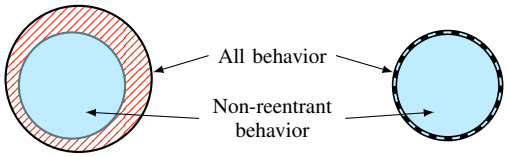
We avoid object reentrancy because it relies on object boundaries—a fundamentally syntactic construct—to define security. Instead we define reentrancy with respect to the integrity level of computation. As integrity levels are part of a semantic security specification, using them to define a security-relevant 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. 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, 44]. 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 III-A) 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.



(a) Vulnerable system (b) Secure system

Fig. 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.

B. 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 [15, 18] 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*. Tail reentrancy also provides a principled explanation for a common smart-contract programming best practice: performing

all state modifications before calling other contracts [56]. 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.

C. 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 III-B. Many services, including the examples in Section II, 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 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.

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

Fig. 6. Syntax for SeRIF

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.

V. 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) [31] with information flow labels and, to support mutation, also reference cells [46, 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 III-B. If $pc_1 \not\Rightarrow pc_2$, 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 form 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.

A. SeRIF Operational Semantics

SeRIF has a small-step substitution-based semantics. Most rules are standard for an object-oriented language with mutable references [31, 46], with a few additions for security.

Because expressions are built mostly out of values, evaluation contexts are simple. Indeed, let expressions are 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.

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

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 VI 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.

[E-IFT] $$\overline{\langle \mathsf{if}\{pc\}\ \mathsf{true\ then}\ e_1\ \mathsf{else}\ e_2 \mid \mathcal{C}\rangle \longrightarrow \langle e_1\ \mathsf{at\text{-}pc}\ pc \mid \mathcal{C}\rangle}$$

[E-ATPC] $$\overline{\langle v\ \mathsf{at\text{-}pc}\ pc \mid \mathcal{C}\rangle \longrightarrow \langle v \mid \mathcal{C}\rangle}$$

[E-REF] $$\frac{\iota \notin \mathrm{dom}(\sigma) \qquad \Sigma_\sigma \vdash v : \tau \qquad \mathcal{M} = \mathcal{M}', \ell_m \qquad \ell_m \lhd \tau}{\langle \mathsf{ref}\ v\ \tau \mid \mathcal{C}\rangle \longrightarrow \langle \iota \mid \mathcal{C}[\sigma[\iota \mapsto (v,\tau)]/\sigma]\rangle}$$

[E-ASSIGN] $$\frac{\Sigma_\sigma(\iota) = \tau \qquad \Sigma_\sigma \vdash v : \tau \qquad \mathcal{M} = \mathcal{M}', \ell_m \qquad \ell_m \lhd \tau}{\langle \iota := v \mid \mathcal{C}\rangle \longrightarrow \langle () \mid \mathcal{C}[\sigma[\iota \mapsto (v,\tau)]/\sigma]\rangle}$$

[E-CALL] $$\frac{\begin{array}{c} mbody(C,m) = (\ell_m, \overline{x}, \overline{\tau_a}, pc_1 \gg pc_2, e, \tau) \\ \mathcal{M} = \mathcal{M}', \ell'_m \qquad \ell'_m \Rightarrow pc_1 \qquad \bigwedge_{\ell \in L} (pc_1 \Rightarrow pc_2 \vee \ell) \\ \Sigma_\sigma \vdash \overline{w} : \overline{\tau_a} \qquad e' = e[\overline{x} \mapsto \overline{w}, \mathsf{this} \mapsto \mathsf{new}\ C(\overline{v})] \end{array}}{\begin{array}{l} \langle \mathsf{new}\ C(\overline{v}).m(\overline{w}) \mid \mathcal{C}\rangle \\ \quad \longrightarrow \langle \mathsf{return}_\tau\ (e'\ \mathsf{at\text{-}pc}\ pc_2) \mid \mathcal{C}[\mathcal{M}, \ell_m/\mathcal{M}]\rangle \end{array}}$$

[E-RETURN] $$\frac{\Sigma_\sigma \vdash v : \tau \qquad \mathcal{M} = \mathcal{M}', \ell_m}{\langle \mathsf{return}_\tau\ v \mid \mathcal{C}\rangle \longrightarrow \langle v \mid \mathcal{C}[\mathcal{M}'/\mathcal{M}]\rangle}$$

[E-LOCK] $$\overline{\langle \mathsf{lock}\ \ell\ \mathsf{in}\ e \mid \mathcal{C}\rangle \longrightarrow \langle e\ \mathsf{with\text{-}lock}\ \ell \mid \mathcal{C}[L, \ell/L]\rangle}$$

[E-UNLOCK] $$\frac{L = L', \ell}{\langle v\ \mathsf{with\text{-}lock}\ \ell \mid \mathcal{C}\rangle \longrightarrow \langle v \mid \mathcal{C}[L'/L]\rangle}$$

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

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 [54], and distributed IFC systems include run-time label checks [e.g., 23, 36, 63]. Moreover, when all high-integrity code is well-typed, it is sufficient to isolate memory between objects, as in Ethereum contracts [58], and to execute run-time checks when entering trusted code.

B. 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 V-A.

$$\text{[If]}\ \frac{\Sigma;\Gamma \vdash v : \mathsf{bool}^{\ell} \qquad \ell \Rightarrow pc \qquad \ell \triangleleft \tau \qquad \Sigma;\Gamma;pc;\lambda_{\mathrm{I}} \vdash e_1 : \tau \dashv \lambda_{\mathrm{o}} \qquad \Sigma;\Gamma;pc;\lambda_{\mathrm{I}} \vdash e_2 : \tau \dashv \lambda_{\mathrm{o}}}{\Sigma;\Gamma;pc;\lambda_{\mathrm{I}} \vdash \mathsf{if}\{pc\}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau \dashv \lambda_{\mathrm{o}}}$$

$$\text{[Assign]}\ \frac{\Sigma;\Gamma \vdash v_1 : (\mathsf{ref}\ \tau)^{\ell} \qquad \Sigma;\Gamma \vdash v_2 : \tau \qquad \ell \triangleleft \tau}{\Sigma;\Gamma;\ell;\lambda_{\mathrm{I}} \vdash v_1 := v_2 : \mathsf{unit}^{\ell'} \dashv \lambda_{\mathrm{o}}}$$

$$\text{[Call]}\ \frac{mtype(C,m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2;\lambda_{\mathrm{o}}} \tau_0 \qquad \Sigma;\Gamma \vdash v : C^{\ell} \qquad \Sigma;\Gamma \vdash \overline{v_a} : \overline{\tau_a} \qquad \ell \Rightarrow pc_1 \qquad pc_1 \Rightarrow pc_2 \vee \lambda_{\mathrm{I}} \qquad \tau_0 <: \tau \qquad pc_2 \vee \ell \triangleleft \tau}{\Sigma;\Gamma;pc_1;\lambda_{\mathrm{I}} \vdash v.m(\overline{v_a}) : \tau \dashv \lambda_{\mathrm{o}} \vee pc_2}$$

$$\text{[Lock]}\ \frac{\Sigma;\Gamma;pc;\lambda'_{\mathrm{I}} \vdash e : \tau \dashv \lambda'_{\mathrm{o}} \qquad \lambda'_{\mathrm{I}} \wedge \ell \Rightarrow \lambda_{\mathrm{I}} \qquad \lambda'_{\mathrm{o}} \wedge \ell \Rightarrow \lambda_{\mathrm{o}}}{\Sigma;\Gamma;pc;\lambda_{\mathrm{I}} \vdash \mathsf{lock}\ \ell\ \mathsf{in}\ e : \tau \dashv \lambda_{\mathrm{o}}}$$

$$\text{[Method-Ok]}\ \frac{\lambda_{\mathrm{I}} \Rightarrow pc_2 \qquad \ell_C \Rightarrow pc_2 \qquad \lambda_{\mathrm{I}} \vee \lambda'_{\mathrm{o}} \Rightarrow \lambda_{\mathrm{o}} \qquad pc_1 \triangleleft \overline{\tau_a} \qquad \Sigma;\overline{x}:\overline{\tau_a}, \mathsf{this}:C^{pc_2};pc_2;\lambda_{\mathrm{I}} \vdash e : \tau \dashv \lambda'_{\mathrm{o}} \qquad CT(C) = \mathsf{class}\ C[\ell_C]\ \mathsf{extends}\ D\ \{\cdots\} \qquad (D,m) \in \mathrm{dom}(mtype) \implies mtype(D,m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2;\lambda_{\mathrm{o}}} \tau}{\Sigma \vdash \tau\ m\{pc_1 \gg pc_2;\lambda_{\mathrm{o}}\}(\overline{x}:\overline{\tau_a})\ \{e\}\ \mathsf{ok\ in}\ C}$$

Fig. 8. Selected typing rules for SeRIF

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 $\Sigma;\Gamma;pc;\lambda_{\mathrm{I}} \vdash e : \tau \dashv \lambda_{\mathrm{o}}$.

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.

$$\text{[Let]}\ \frac{\Sigma;\Gamma;pc;\lambda_{\mathrm{I}} \vdash e_1 : \tau_1 \dashv \lambda'_{\mathrm{o}} \qquad \lambda'_{\mathrm{o}} \Rightarrow \lambda_{\mathrm{I}} \qquad \Sigma;\Gamma, x:\tau_1;pc;\lambda_{\mathrm{I}} \vdash e_2 : \tau_2 \dashv \lambda_{\mathrm{o}}}{\Sigma;\Gamma;pc;\lambda_{\mathrm{I}} \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \dashv \lambda_{\mathrm{o}}}$$

This rule is standard except that it requires $\lambda'_{\mathrm{o}} \Rightarrow \lambda_{\mathrm{I}}$, 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_{o}.

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 Figure 10 (Appendix A).

Most typing rules (e.g., If and Assign) are standard for an information flow calculus [50]. 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 ($pc_1 \Rightarrow pc_2 \vee \lambda_{\mathrm{I}}$) 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_{o}, 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_{o}—the locks maintained by the method—but instead only $\lambda_{\mathrm{o}} \vee pc_2$. 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 [41]: you can only be hurt by an adversary you trust. We therefore attenuate the claimed lock label λ_{o} 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 $\lambda'_{\mathrm{I}} \wedge \ell \Rightarrow \lambda_{\mathrm{I}}$. 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 $\lambda'_{\mathrm{o}} \wedge \ell \Rightarrow \lambda_{\mathrm{o}}$. 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.

C. 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 $\overline{I} = I_1, \ldots, I_n$ and $(I_i, CT, \sigma_{i-1}) \Downarrow \sigma_i$ for each $1 \le i \le n$, 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:

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

D. Examples Revisited

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

Uniswap: The vulnerability (Section II-A) 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 $U \not\Rightarrow T$.

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

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 $\lambda'_o \Rightarrow \lambda_I \Rightarrow T$, or a dynamic lock on label T around (at least) the first transfer. These options correspond precisely to the solutions suggested in Section II-A. Requiring $\lambda'_o \Rightarrow T$ 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'_o.

Notably, `transferTo` can type-check with $\lambda'_o \Rightarrow T$ 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 $U \gg T; \lambda_o$ for some λ_o. The user-provided computation function is not trusted, so it gets label $pc \gg U; \lambda'_o$ for some labels pc and λ'_o.

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 $\lambda'_o \vee U \Rightarrow \lambda_I \Rightarrow T$. 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 $\lambda'_o \Rightarrow T$. 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 II-C and the original paper [65], 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:t^U, callback:address^U)
void cancel{U ≫ T; U}(id:int^U)
void deliver{T ≫ T; U}(id:int^T, data:bytes^T)
```

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.

VI. Formalizing Security Guarantees

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

A. 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 [48, 53]. 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.

$$\begin{array}{rcl} e & ::= & \cdots \mid \mathsf{ignore\text{-}locks\text{-}in}\ e \\ E & ::= & \cdots \mid \mathsf{ignore\text{-}locks\text{-}in}\ E \end{array}$$

$$[\text{E-IgnoreLocks}]\ \frac{}{\langle \mathsf{ignore\text{-}locks\text{-}in}\ v \mid \mathcal{C}\rangle \longrightarrow \langle v \mid \mathcal{C}\rangle}$$

$$[\text{IgnoreLocks}]\ \frac{\Sigma; \Gamma; pc; \lambda_{\mathrm{I}}' \vdash e : \tau \dashv \lambda_{\mathrm{o}}'}{\Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash \mathsf{ignore\text{-}locks\text{-}in}\ e : \tau \dashv \lambda_{\mathrm{o}}}$$

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.

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

$$CT(C) = \mathsf{class}\ C[\ell_C]\ \mathsf{extends}\ D\ \{\overline{f} : \overline{\tau_f}\ ;\ K\ ;\ \overline{M}\}$$

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

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.

$$[\text{E-CallAtk}]\ \frac{\begin{array}{c} mbody(C, m) = (\ell_m, \overline{x}, \overline{\tau_a}, pc_1 \gg pc_2, e, \tau) \\ \mathcal{M} = \mathcal{M}', \ell_m' \qquad \ell_m' \Rightarrow pc_1 \qquad \ell_{\mathcal{A}} \Rightarrow pc_2 \\ \Sigma_\sigma \vdash \overline{w} : \overline{\tau_a} \qquad e' = e[\overline{x} \mapsto \overline{w}, \mathsf{this} \mapsto \mathsf{new}\ C(\overline{v})] \end{array}}{\begin{array}{l} \langle \mathsf{new}\ C(\overline{v}).m(\overline{w}) \mid \mathcal{C}\rangle \\ \qquad \longrightarrow \langle \mathsf{return}_\tau\ (e'\ \mathsf{at\text{-}pc}\ pc_2) \mid \mathcal{C}[\mathcal{M}, \ell_m/\mathcal{M}]\rangle \end{array}}$$

This rule is identical to E-Call, except instead of checking dynamic locks, it checks that pc_2 is untrusted ($\ell_{\mathcal{A}} \Rightarrow pc_2$).

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 $\ell_{\mathcal{A}}$ cannot be a well-defined label. The proofs in the technical report [13] 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

[2] Our 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 $\ell_{\mathcal{A}} = \bigwedge \mathcal{A}$.

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.

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

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:

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

B. Noninterference

A typical goal for security in IFC systems, including our core calculus, is *noninterference* [24], which for integrity means untrusted data should not influence trusted data at all. As we argued in Section III-B, 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

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

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 $\mathsf{endorse}\ v\ \mathsf{from}\ \ell_1\ \mathsf{to}\ \ell_2$, 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 $\simeq_\ell$ 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 $\mathrm{dom}(f) \subseteq \mathrm{dom}(f')$ and $f(x) = f'(x)$ wherever f is defined.

Theorem 1 (Noninterference)**.** *Let CT be a class table where $\Sigma \vdash CT\ \mathsf{ok}$ 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 the technical report [13] for details.

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

C. Formalizing Reentrancy

Definition 1 in Section IV-A 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 $\mathsf{at\text{-}pc}$ 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 $\mathsf{at\text{-}pc}$ 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,

$$s = E_0\Big[E_1\big[E_2[s'\ \mathsf{at\text{-}pc}\ pc_3]\ \mathsf{at\text{-}pc}\ pc_2\big]\ \mathsf{at\text{-}pc}\ pc_1\Big]$$

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 IV-B. 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 [29]. 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 $\Sigma \vDash_\ell \{P\}\ CT\ \{Q\}$, 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 $\overline{I}$ such that $\Sigma_{\sigma_1} \vdash \overline{I}$ and $(\overline{I}, CT', \sigma_1) \Downarrow \sigma_2$, 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 $\Sigma \vDash^1_\ell \{P\}\ CT\ \{Q\}$, if CT satisfies (P, Q) at ℓ in Σ when restricted to invocation sequences $\overline{I}$ 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, $\Sigma \vDash^1_\ell \{P\}\ CT\ \{Q\}$ implies $\Sigma \vDash_\ell \{P\}\ CT\ \{Q\}$.

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 our technical report [13]. □

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 $\Sigma_{\sigma_1} \vDash^1_\ell \{P\}\ CT\ \{Q\}$ and $P(\sigma_1)$, then $Q(\sigma_2)$.*

Proof Sketch. Examine the execution of I and build a CT' and $\overline{I}$ 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 $\overline{I}$ 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 $\Sigma_{\sigma_1} \vDash^1_\ell \{P\}\ CT\ \{Q\}$,

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

See the technical report [13] for details. □

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 $\Sigma_{\sigma_1} \vDash^1_\ell \{P\}\ CT\ \{Q\}$, Theorem 4 says that if $P(\sigma_1)$ then $Q(\sigma_2)$, which is precisely the definition of $\Sigma_{\sigma_1} \vDash_\ell \{P\}\ CT\ \{Q\}$. □

VII. Implementation

We implemented a type checker for SeRIF in 4,200 lines of Java, using JFlex [32] and CUP [30]. We employ the SHErrLoc constraint solver [64] 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 II, 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 [47]. It is one

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 I. Evaluation of SeRIF type checker. Asterisks indicate vulnerable implementations.

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 II. It also accepted as secure patched implementations following the suggested fixes, both with and without dynamic locks.

Developer Overhead: Table I 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 V-D, 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.

```
bool^T transferTo{T ≫ T; T}(from:H^T, to:H^T, amount:int^T)
```

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:H^U, amount:int^U)
void alertReceive{U ≫ U; U}(from:H^U, amount:int^U)
```

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.

VIII. Related Work

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

Formal Reentrancy Security: Grossman et al. [27] 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 [52] 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 II-A).

Obsidian [15] and Flint [51] 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 [18] 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 [37] searches for anti-patterns in code, TEETHER [34] 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 [42] and ETHBMC [22] prove security against specific classes of vulnerabilities using symbolic execution and bounded model checking, respectively. EtherTrust [25] allows developers to specify program properties as Horn clauses and verify them using a formal semantics for EVM [26]. SOLYTHESIS [35] 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 [36] is a system and language for building distributed systems that allows secure data and code sharing between nodes despite mutual distrust. DStar [63] 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 I: they do not defend against reentrancy attacks. The IFC-based instruction set of Zagieboylo et al. [60] 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.

IX. 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*, vol. 4, no. OOPSLA, Nov. 2020.

[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*, May 2012, pp. 191–205.

[6] O. Arden, J. Liu, and A. C. Myers, "Flow-limited authorization," in *28th IEEE Computer Security Foundations Symp. (CSF)*, Jul. 2015, pp. 569–583.

[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)*, Oct. 2016, pp. 91–96.

[8] K. J. Biba, "Integrity considerations for secure computer systems," USAF Electronic Systems Division, Bedford, MA, Tech. Rep. ESD-TR-76-372, Apr. 1977, (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 Jul. 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)*, Jun. 2020, p. 454–469.

[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] ——, "Compositional security for reentrant applications," Cornell University Computing and Information Science, Tech. Rep. arXiv:2103.08577, Mar. 2021.

[14] 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)*, Jun. 2009.

[15] 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*, vol. 42, no. 3, Nov. 2020.

[16] ConsenSys Diligence, "Uniswap audit," https://github.com/ConsenSys/Uniswap-audit-report-2018-12#31-liquidity-pool-can-be-stolen-in-some-tokens-eg-erc-777-29, Jan. 2019, accessed March 2021.

[17] P. Daian, "Analysis of the DAO exploit," https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/, 18 Jun. 2016, accessed March 2021.

[18] 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.

[19] 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*. Springer, 2017, pp. 195–216.

[20] 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.

[21] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, 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)*, Nov. 2014, pp. 1092–1104.

[22] J. Frank, C. Aschermann, and T. Holz, "ETHBMC: A bounded model checker for smart contracts," in *29th USENIX Security Symp.*, Aug. 2020.

[23] 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)*. USENIX Association, 2012, pp. 47–60.

[24] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symp. on Security and Privacy*, Apr. 1982, pp. 11–20.

[25] 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)*. Springer, 2018, pp. 51–78.

[26] ——, "A semantic framework for the security analysis of Ethereum smart contracts," in *Int'l Conf. on Principles of Security and Trust (POST)*. Springer, 2018, pp. 243–269.

[27] 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*, vol. 2, no. POPL, pp. 1–28, Dec. 2017.

[28] D. Hedin and A. Sabelfeld, "Information-flow security for a core of JavaScript," in *25th IEEE Computer Security Foundations Symp. (CSF)*, Jun. 2012.

[29] C. A. R. Hoare, "Proof of correctness of data representations," *Acta Informatica*, vol. 1, no. 4, pp. 271–281, 1972.

[30] S. Hudson, F. Flannery, C. S. Ananian, and M. Petter, "CUP 0.11b: Construction of Useful Parsers," Jun. 2014, software release, http://www2.cs.tum.edu/projects/cup.

[31] A. Igarashi, B. Pierce, and P. Wadler, "Featherweight Java: A minimal core calculus for Java and GJ," *ACM Trans. on Programming Languages and Systems*, vol. 23, no. 3, pp. 396–450, 2001.

[32] G. Klein, S. Rowe, and R. Decamp, "JFlex 1.8.2," May 2020, software release, https://jflex.de.

[33] 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.

[34] J. Krupp and C. Rossow, "TEETHER: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symp.*, Aug. 2018.

[35] 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)*, Jun. 2020, pp. 438–453.

[36] J. Liu, O. Arden, M. D. George, and A. C. Myers, "Fabric: Building open distributed systems securely by construction," *J. Computer Security*, vol. 25, no. 4–5, pp. 319–321, May 2017.

[37] 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)*, 2016, pp. 254–269.

[38] T. Magrino, J. Liu, O. Arden, C. Isradisaikul, and A. C. Myers, "Jif 3.5: Java information flow," Jun. 2016, software release, https://www.cs.cornell.edu/jif.

[39] 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.

[40] A. C. Myers and B. Liskov, "Complete, safe information flow with decentralized labels," in *IEEE Symp. on Security and Privacy*, May 1998, pp. 186–197.

[41] ——, "Protecting privacy using the decentralized label model," *ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 4, pp. 410–442, Oct. 2000.

[42] 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*, Dec. 2018, pp. 653–663.

[43] 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(K,java.util.function.Function), Sep. 2020, accessed March 2021.

[44] 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.

[45] PeckShield, "Uniswap/Lendf.Me hacks: Root cause and loss analysis," https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09, Apr. 2020, accessed March 2021.

[46] B. C. Pierce, *Types and programming languages*. MIT press, 2002.

[47] N. Popper, "A hacking of more than $50 million dashes hopes in the world of virtual currency," *The New York Times*, 17 Jun. 2016.

[48] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur. (TISSEC)*, vol. 15, no. 1, Mar. 2012.

[49] "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.Entry.html#method.or_insert_with, Nov. 2020, accessed March 2021.

[50] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[51] 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*, 2018, pp. 218–219.

[52] 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*, vol. 3, no. OOPSLA, pp. 1–30, Oct. 2019.

[53] 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)*, Oct. 2007, p. 552–561.

[54] J. Siek and W. Taha, "Gradual typing for objects," in *21st European Conf. on Object-Oriented Programming*, Jul. 2007, pp. 2–27.

[55] "Solidity documentation. Release 0.7.5," https://docs.soliditylang.org/en/v0.7.5/, Nov. 18 2020, accessed December 2020.

[56] "Solidity security considerations," https://solidity.readthedocs.io/en/latest/security-considerations.html#use-the-checks-effects-interactions-pattern, 2021, accessed March 2021.

[57] The Open Group, "SOA standards," https://publications.opengroup.org/standards/soa, accessed December 2020.

[58] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2014.

[59] 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)*, 2012, pp. 85–96.

[60] 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)*, Jun. 2019.

[61] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Secure program partitioning," *ACM Trans. on Computer Systems*, vol. 20, no. 3, pp. 283–328, Aug. 2002.

[62] 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)*, 2006, pp. 263–278.

[63] 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)*, 2008, pp. 293–308.

[64] D. Zhang, A. C. Myers, D. Vytiniotis, and S. Peyton Jones, "SHErrLoc: A static holistic error locator," *ACM Trans. on Programming Languages and Systems*, vol. 39, no. 4, p. 18, Aug. 2017.

[65] 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)*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 270–282.

[66] L. Zheng and A. C. Myers, "End-to-end availability policies and noninterference," in *18th IEEE Computer Security Foundations Workshop (CSFW)*, Jun. 2005, pp. 272–286.

[67] 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*, May 2003, pp. 236–250.

APPENDIX A
FULL SERIF RULES

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

APPENDIX 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.

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

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, $\sigma_1 \simeq \sigma_2 \overset{\triangle}{\iff} \exists\theta.\, \sigma_1 = \theta(\sigma_2)$.

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 $(s_1, \mathcal{C}_1) \simeq (s_2, \mathcal{C}_2)$.

[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] $$\frac{}{\langle \mathsf{let}\ x = v\ \mathsf{in}\ e \mid \mathcal{C}\rangle \longrightarrow \langle e[x \mapsto v] \mid \mathcal{C}\rangle}$$

[E-IFT] $$\frac{}{\langle \mathsf{if}\{pc\}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathcal{C}\rangle \longrightarrow \langle e_1\ \mathsf{at\text{-}pc}\ pc \mid \mathcal{C}\rangle}$$

[E-IFF] $$\frac{}{\langle \mathsf{if}\{pc\}\ \mathsf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathcal{C}\rangle \longrightarrow \langle e_2\ \mathsf{at\text{-}pc}\ pc \mid \mathcal{C}\rangle}$$

[E-ATPC] $$\frac{}{\langle v\ \mathsf{at\text{-}pc}\ pc \mid \mathcal{C}\rangle \longrightarrow \langle v \mid \mathcal{C}\rangle}$$

[E-REF] $$\frac{\iota \notin \mathrm{dom}(\sigma) \qquad \Sigma_\sigma \vdash v : \tau \qquad \mathcal{M} = \mathcal{M}', \ell_m \qquad \ell_m \triangleleft \tau}{\langle \mathsf{ref}\ v\ \tau \mid \mathcal{C}\rangle \longrightarrow \langle \iota \mid \mathcal{C}[\sigma[\iota \mapsto (v, \tau)]/\sigma]\rangle}$$

[E-DEREF] $$\frac{\sigma(\iota) = (v, \tau)}{\langle !\iota \mid \mathcal{C}\rangle \longrightarrow \langle v \mid \mathcal{C}\rangle}$$

[E-ASSIGN] $$\frac{\Sigma_\sigma(\iota) = \tau \qquad \Sigma_\sigma \vdash v : \tau \qquad \mathcal{M} = \mathcal{M}', \ell_m \qquad \ell_m \triangleleft \tau}{\langle \iota := v \mid \mathcal{C}\rangle \longrightarrow \langle () \mid \mathcal{C}[\sigma[\iota \mapsto (v, \tau)]/\sigma]\rangle}$$

[E-CAST] $$\frac{D <: C}{\langle (C)(\mathsf{new}\ D(\overline{v})) \mid \mathcal{C}\rangle \longrightarrow \langle \mathsf{new}\ D(\overline{v}) \mid \mathcal{C}\rangle}$$

[E-FIELD] $$\frac{}{\langle \mathsf{new}\ C(\overline{v}).f_i \mid \mathcal{C}\rangle \longrightarrow \langle v_i \mid \mathcal{C}\rangle}$$

[E-CALL] $$\frac{\begin{array}{c} mbody(C, m) = (\ell_m, \overline{x}, \overline{\tau_a}, pc_1 \gg pc_2, e, \tau) \\ \mathcal{M} = \mathcal{M}', \ell'_m \qquad \ell'_m \Rightarrow pc_1 \qquad \bigwedge_{\ell \in L} (pc_1 \Rightarrow pc_2 \vee \ell) \\ \Sigma_\sigma \vdash \overline{w} : \overline{\tau_a} \qquad e' = e[\overline{x} \mapsto \overline{w}, \mathsf{this} \mapsto \mathsf{new}\ C(\overline{v})] \end{array}}{\begin{array}{l} \langle \mathsf{new}\ C(\overline{v}).m(\overline{w}) \mid \mathcal{C}\rangle \\ \quad \longrightarrow \langle \mathsf{return}_\tau\ (e'\ \mathsf{at\text{-}pc}\ pc_2) \mid \mathcal{C}[\mathcal{M}, \ell_m/\mathcal{M}]\rangle \end{array}}$$

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

[E-RETURN] $$\frac{\Sigma_\sigma \vdash v : \tau \qquad \mathcal{M} = \mathcal{M}', \ell_m}{\langle \mathsf{return}_\tau\ v \mid \mathcal{C}\rangle \longrightarrow \langle v \mid \mathcal{C}[\mathcal{M}'/\mathcal{M}]\rangle}$$

[E-LOCK] $$\frac{}{\langle \mathsf{lock}\ \ell\ \mathsf{in}\ e \mid \mathcal{C}\rangle \longrightarrow \langle e\ \mathsf{with\text{-}lock}\ \ell \mid \mathcal{C}[L, \ell/L]\rangle}$$

[E-UNLOCK] $$\frac{L = L', \ell}{\langle v\ \mathsf{with\text{-}lock}\ \ell \mid \mathcal{C}\rangle \longrightarrow \langle v \mid \mathcal{C}[L'/L]\rangle}$$

[E-ENDORSE] $$\frac{}{\langle \mathsf{endorse}\ v\ \mathsf{from}\ \ell'\ \mathsf{to}\ \ell \mid \mathcal{C}\rangle \longrightarrow \langle v \mid \mathcal{C}\rangle}$$

[E-IGNORELOCKS] $$\frac{}{\langle \mathsf{ignore\text{-}locks\text{-}in}\ v \mid \mathcal{C}\rangle \longrightarrow \langle v \mid \mathcal{C}\rangle}$$

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

Value Typing

$$[\textsc{Var}]\ \frac{\Gamma(x) = \tau}{\Sigma; \Gamma \vdash x : \tau} \qquad [\textsc{Unit}]\ \frac{}{\Sigma; \Gamma \vdash () : \mathsf{unit}^{\ell}} \qquad [\textsc{True}]\ \frac{}{\Sigma; \Gamma \vdash \mathsf{true} : \mathsf{bool}^{\ell}} \qquad [\textsc{False}]\ \frac{}{\Sigma; \Gamma \vdash \mathsf{false} : \mathsf{bool}^{\ell}}$$

$$[\textsc{New}]\ \frac{\begin{array}{c}\mathit{fields}(C) = \overline{f}{:}\overline{\tau} \\ \Sigma; \Gamma \vdash \overline{v} : \overline{\tau}\end{array}}{\Sigma; \Gamma \vdash \mathsf{new}\ C(\overline{v}) : C^{\ell}} \qquad [\textsc{Loc}]\ \frac{\Sigma(\iota) = \tau}{\Sigma; \Gamma \vdash \iota : (\mathsf{ref}\ \tau)^{\ell}} \qquad [\textsc{Null}]\ \frac{}{\Sigma; \Gamma \vdash \mathsf{null} : (\mathsf{ref}\ \tau)^{\ell}} \qquad [\textsc{SubtypeV}]\ \frac{\Sigma; \Gamma \vdash v : \tau' \qquad \tau' <: \tau}{\Sigma; \Gamma \vdash v : \tau}$$

Core Expression Typing

$$[\textsc{Val}]\ \frac{\Sigma; \Gamma \vdash v : \tau}{\Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash v : \tau \dashv \lambda_{\mathrm{o}}} \qquad [\textsc{Endorse}]\ \frac{\Sigma; \Gamma \vdash v : t^{\ell'}}{\Sigma; \Gamma; \ell; \lambda_{\mathrm{I}} \vdash \mathsf{endorse}\ v\ \mathsf{from}\ \ell'\ \mathsf{to}\ \ell : t^{\ell} \dashv \lambda_{\mathrm{o}}} \qquad [\textsc{Cast}]\ \frac{\Sigma; \Gamma \vdash v : D^{\ell}}{\Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash (C)v : C^{\ell} \dashv \lambda_{\mathrm{o}}}$$

$$[\textsc{Field}]\ \frac{\begin{array}{c}\Sigma; \Gamma \vdash v : C^{\ell} \\ \mathit{fields}(C) = \overline{f}{:}\overline{\tau} \\ \tau_i <: \tau \qquad \ell \lhd \tau\end{array}}{\Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash v.f_i : \tau \dashv \lambda_{\mathrm{o}}} \qquad [\textsc{Call}]\ \frac{\begin{array}{c}\mathit{mtype}(C, m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \lambda_{\mathrm{o}}} \tau_0 \\ \Sigma; \Gamma \vdash v : C^{\ell} \qquad \Sigma; \Gamma \vdash \overline{v_a} : \overline{\tau_a} \\ \ell \Rightarrow pc_1 \qquad pc_1 \Rightarrow pc_2 \vee \lambda_{\mathrm{I}} \\ \tau_0 <: \tau \qquad pc_2 \vee \ell \lhd \tau\end{array}}{\Sigma; \Gamma; pc_1; \lambda_{\mathrm{I}} \vdash v.m(\overline{v_a}) : \tau \dashv \lambda_{\mathrm{o}} \vee pc_2}$$

$$[\textsc{If}]\ \frac{\begin{array}{c}\Sigma; \Gamma \vdash v : \mathsf{bool}^{\ell} \qquad \ell \Rightarrow pc \qquad \ell \lhd \tau \\ \Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash e_1 : \tau \dashv \lambda_{\mathrm{o}} \\ \Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash e_2 : \tau \dashv \lambda_{\mathrm{o}}\end{array}}{\Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash \mathsf{if}\{pc\}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau \dashv \lambda_{\mathrm{o}}}$$

$$[\textsc{Ref}]\ \frac{\Sigma; \Gamma \vdash v : \tau \qquad pc \lhd \tau}{\Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash \mathsf{ref}\ v\ \tau : (\mathsf{ref}\ \tau)^{\ell} \dashv \lambda_{\mathrm{o}}} \qquad [\textsc{Deref}]\ \frac{\begin{array}{c}\Sigma; \Gamma \vdash v : (\mathsf{ref}\ \tau')^{\ell} \\ \tau' <: \tau \qquad \ell \lhd \tau\end{array}}{\Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash !v : \tau \dashv \lambda_{\mathrm{o}}} \qquad [\textsc{Assign}]\ \frac{\begin{array}{c}\Sigma; \Gamma \vdash v_1 : (\mathsf{ref}\ \tau)^{\ell} \\ \Sigma; \Gamma \vdash v_2 : \tau \qquad \ell \lhd \tau\end{array}}{\Sigma; \Gamma; \ell; \lambda_{\mathrm{I}} \vdash v_1 := v_2 : \mathsf{unit}^{\ell'} \dashv \lambda_{\mathrm{o}}}$$

$$[\textsc{Lock}]\ \frac{\begin{array}{c}\Sigma; \Gamma; pc; \lambda'_{\mathrm{I}} \vdash e : \tau \dashv \lambda'_{\mathrm{o}} \\ \lambda'_{\mathrm{I}} \wedge \ell \Rightarrow \lambda_{\mathrm{I}} \qquad \lambda'_{\mathrm{o}} \wedge \ell \Rightarrow \lambda_{\mathrm{o}}\end{array}}{\Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash \mathsf{lock}\ \ell\ \mathsf{in}\ e : \tau \dashv \lambda_{\mathrm{o}}} \qquad [\textsc{Let}]\ \frac{\begin{array}{c}\Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash e_1 : \tau_1 \dashv \lambda'_{\mathrm{o}} \qquad \lambda'_{\mathrm{o}} \Rightarrow \lambda_{\mathrm{I}} \\ \Sigma; \Gamma, x{:}\tau_1; pc; \lambda_{\mathrm{I}} \vdash e_2 : \tau_2 \dashv \lambda_{\mathrm{o}}\end{array}}{\Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \dashv \lambda_{\mathrm{o}}} \qquad [\textsc{Variance}]\ \frac{\begin{array}{c}\Sigma; \Gamma; pc'; \lambda'_{\mathrm{I}} \vdash e : \tau' \dashv \lambda'_{\mathrm{o}} \\ \tau' <: \tau \qquad pc \Rightarrow pc' \\ \lambda'_{\mathrm{I}} \Rightarrow \lambda_{\mathrm{I}} \qquad \lambda'_{\mathrm{o}} \Rightarrow \lambda_{\mathrm{o}}\end{array}}{\Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash e : \tau \dashv \lambda_{\mathrm{o}}}$$

Tracking Statement Typing

$$[\textsc{AtPc}]\ \frac{\Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash s : \tau \dashv \lambda_{\mathrm{o}}}{\Sigma; \Gamma; pc'; \lambda_{\mathrm{I}} \vdash s\ \mathsf{at\text{-}pc}\ pc : \tau \dashv \lambda_{\mathrm{o}}} \qquad [\textsc{WithLock}]\ \frac{\begin{array}{c}\Sigma; \Gamma; pc; \lambda'_{\mathrm{I}} \vdash s : \tau \dashv \lambda'_{\mathrm{o}} \\ \lambda'_{\mathrm{I}} \wedge \ell \Rightarrow \lambda_{\mathrm{I}} \qquad \lambda'_{\mathrm{o}} \wedge \ell \Rightarrow \lambda_{\mathrm{o}}\end{array}}{\Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash s\ \mathsf{with\text{-}lock}\ \ell : \tau \dashv \lambda_{\mathrm{o}}} \qquad [\textsc{Return}]\ \frac{\begin{array}{c}\Sigma; \cdot; pc; \lambda'_{\mathrm{I}} \vdash s : \tau \dashv \lambda'_{\mathrm{o}} \\ \lambda'_{\mathrm{I}} \vee \lambda'_{\mathrm{o}} \Rightarrow \lambda_{\mathrm{o}}\end{array}}{\Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash \mathsf{return}_{\tau}\ s : \tau \dashv \lambda_{\mathrm{o}}}$$

Attacker-Model Expression Typing

$$[\textsc{IgnoreLocks}]\ \frac{\Sigma; \Gamma; pc; \lambda'_{\mathrm{I}} \vdash e : \tau \dashv \lambda'_{\mathrm{o}}}{\Sigma; \Gamma; pc; \lambda_{\mathrm{I}} \vdash \mathsf{ignore\text{-}locks\text{-}in}\ e : \tau \dashv \lambda_{\mathrm{o}}}$$

Class Typing

$$[\textsc{Method-Ok}]\ \frac{\begin{array}{c}\lambda_{\mathrm{I}} \Rightarrow pc_2 \qquad \ell_C \Rightarrow pc_2 \qquad \lambda_{\mathrm{I}} \vee \lambda'_{\mathrm{o}} \Rightarrow \lambda_{\mathrm{o}} \qquad pc_1 \lhd \overline{\tau_a} \\ \Sigma; \overline{x}{:}\overline{\tau_a}, \mathsf{this}{:}C^{pc_2}; pc_2; \lambda_{\mathrm{I}} \vdash e : \tau \dashv \lambda'_{\mathrm{o}} \\ CT(C) = \mathsf{class}\ C[\ell_C]\ \mathsf{extends}\ D\ \{\cdots\} \\ \mathit{can\text{-}override}(D, m, \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \lambda_{\mathrm{o}}} \tau)\end{array}}{\Sigma \vdash \tau\ m\{pc_1 \gg pc_2; \lambda_{\mathrm{o}}\}(\overline{x}{:}\overline{\tau_a})\ \{e\}\ \mathsf{ok\ in}\ C}$$

$$[\textsc{Class-Ok}]\ \frac{\begin{array}{c}\mathit{fields}(D) = \overline{g}{:}\overline{\tau_g} \\ K = C(\overline{g}{:}\overline{\tau_g}\ ;\ \overline{f}{:}\overline{\tau_f})\ \{\mathsf{super}(\overline{g})\ ;\ \mathsf{this}.\overline{f} = \overline{f}\} \\ \Sigma \vdash \overline{M}\ \mathsf{ok\ in}\ C\end{array}}{\Sigma \vdash \mathsf{class}\ C[\ell_C]\ \mathsf{extends}\ D\ \{\overline{f}{:}\overline{\tau_f}\ ;\ K\ ;\ \overline{M}\}\ \mathsf{ok}}$$

$$[\text{CT-Ok}]\ \frac{\begin{array}{c}C\ \text{referenced in any type} \implies C \in \mathrm{dom}(CT) \\ \forall C \in \mathrm{dom}(CT).\, \Sigma \vdash CT(C)\ \mathsf{ok}\end{array}}{\Sigma \vdash CT\ \mathsf{ok}}$$

Lookup Functions

$$\frac{\begin{array}{c}CT(C) = \mathsf{class}\ C[\ell_C]\ \mathsf{extends}\ D\ \{\overline{f}{:}\overline{\tau_f}\ ;\ K\ ;\ \overline{M}\} \\ \mathit{fields}(D) = \overline{g}{:}\overline{\tau_g}\end{array}}{\mathit{fields}(C) = \overline{g}{:}\overline{\tau_g}\ ;\ \overline{f}{:}\overline{\tau_f}}$$

$$\frac{\begin{array}{c}CT(C) = \mathsf{class}\ C[\ell_C]\ \mathsf{extends}\ D\ \{\overline{f}{:}\overline{\tau_f}\ ;\ K\ ;\ \overline{M}\} \\ \tau\ m\{pc_1 \gg pc_2; \lambda_0\}(\overline{x}{:}\overline{\tau_a})\ \{e\} \in \overline{M}\end{array}}{\begin{array}{c}\mathit{mtype}(C, m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \lambda_{\mathrm{o}}} \tau \\ \mathit{mbody}(C, m) = (\ell_C, \overline{x}, \overline{\tau_a}, pc_1 \gg pc_2, e, \tau)\end{array}}$$

$$\frac{\begin{array}{c}CT(C) = \mathsf{class}\ C[\ell_C]\ \mathsf{extends}\ D\ \{\overline{f}{:}\overline{\tau_f}\ ;\ K\ ;\ \overline{M}\} \\ m\ \text{not defined in}\ \overline{M}\end{array}}{\begin{array}{c}\mathit{mtype}(C, m) = \mathit{mtype}(D, m) \\ \mathit{mbody}(C, m) = \mathit{mbody}(D, m)\end{array}}$$

$$\frac{(D, m) \in \mathrm{dom}(\mathit{mtype}) \implies \mathit{mtype}(D, m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \lambda_{\mathrm{o}}} \tau}{\mathit{can\text{-}override}(D, m, \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \lambda_{\mathrm{o}}} \tau)}$$

Protection

$$\frac{\ell \Rightarrow \ell'}{\ell \lhd t^{\ell'}}$$

Subtyping

$$\frac{\ell \Rightarrow \ell'}{t^{\ell} <: t^{\ell'}} \qquad \frac{CT(C) = \mathsf{class}\ C[\ell_C]\ \mathsf{extends}\ D\ \{\cdots\}}{C^{\ell} <: D^{\ell}} \qquad \frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

Heap Typing

$$\frac{\sigma(\iota) = (v, \tau) \implies \Sigma_{\sigma} \vdash v : \tau}{\vdash \sigma\ \mathsf{wt}}$$

Fig. 10. Full typing rules for SeRIF.

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). □

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 $\sigma_1 \simeq_\ell \sigma_2$, if there exists a θ such that $\sigma_1|_{\ell_t} = \theta(\sigma_2)|_{\ell_t}$.

APPENDIX C
PRESERVATION AND PROGRESS

We now prove preservation and progress theorems for SeRIF. The proofs of all theorems in this section follow by induction and are available in the technical report [13].

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

Theorem 6 (Type Preservation)**.** *If*

- $\langle s \mid (CT, \sigma, \mathcal{M}, L)\rangle \longrightarrow \langle s' \mid (CT, \sigma', \mathcal{M}', L')\rangle$,
- $\Sigma_\sigma \vdash CT$ ok,
- $\Sigma_\sigma; \Gamma; pc; \lambda_\mathrm{I} \vdash s : \tau \dashv \lambda_\mathrm{O}$, *and*
- $\vdash \sigma$ wt,

then

- $\Sigma_\sigma \subseteq \Sigma_{\sigma'}$,
- $\vdash \sigma'$ wt, *and*
- $\Sigma_{\sigma'}; \Gamma; pc; \lambda_\mathrm{I} \vdash s' : \tau \dashv \lambda_\mathrm{O}$.

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 $\mathcal{M} \leftrightsquigarrow s$, 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 $\mathcal{M} \leftrightsquigarrow E$.

$$\frac{}{\ell_m \leftrightsquigarrow [\cdot]} \qquad \frac{\mathcal{M} \leftrightsquigarrow E}{\mathcal{M} \leftrightsquigarrow \mathsf{let}\ x = E\ \mathsf{in}\ e} \qquad \frac{\mathcal{M} \leftrightsquigarrow E}{\mathcal{M} \leftrightsquigarrow E\ \mathsf{with\text{-}lock}\ \ell}$$

$$\frac{\mathcal{M} \leftrightsquigarrow E}{\mathcal{M} \leftrightsquigarrow \mathsf{ignore\text{-}locks\text{-}in}\ E}$$

$$\frac{\mathcal{M} \leftrightsquigarrow E}{\ell, \mathcal{M} \leftrightsquigarrow \mathsf{return}_\tau\ E} \qquad \frac{\ell, \mathcal{M} \leftrightsquigarrow E \qquad \ell \Rightarrow pc}{\ell, \mathcal{M} \leftrightsquigarrow E\ \mathsf{at\text{-}pc}\ pc}$$

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 $\vdash CT$ ok and $(\ell_m, \mathcal{M}) \leftrightsquigarrow s$ and $\Sigma_\sigma; \Gamma; \ell_m; \lambda_\mathrm{I} \vdash s : \tau \dashv \lambda_\mathrm{O}$ and $\langle s \mid \mathcal{C}\rangle \longrightarrow \langle s' \mid \mathcal{C}'\rangle$, then $\mathcal{M}' \leftrightsquigarrow s'$.*

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*

- $\Sigma_\sigma; \cdot; pc; \lambda_\mathrm{I} \vdash s : \tau \dashv \lambda_\mathrm{O}$,
- $\ell_m \Rightarrow pc$, *and*
- $(\ell_m, \mathcal{M}) \leftrightsquigarrow s$,

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) *$s = E[(C)(\mathsf{new}\ D(\overline{v}))]$ where $D \not<: C$,*
4) *$s = E[!\mathsf{null}]$ or $s = E[\mathsf{null} := v]$, or*
5) *$s = E[\mathsf{new}\ C(\overline{v}).m(\overline{w})]$ for a C and m such that $mtype(C, m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \lambda_\mathrm{O}} \tau$ and there is some $\ell_m \in L$ such that $pc_1 \not\Rightarrow pc_2 \vee \ell_m$.*

Note that, for any invocation $I = (\ell, \iota, m(\overline{v}))$, $\ell \leftrightsquigarrow !\iota.m(\overline{v})$. 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.