# SCIF: A Language for Compositional Smart Contract Security

SIQIU YAO, Cornell University, USA
HAOBIN NI, Cornell University, USA
ANDREW C. MYERS, Cornell University, USA
ETHAN CECCHETTI, University of Wisconsin–Madison, USA

Securing smart contracts remains a fundamental challenge. At its core, it is about building software that is secure in composition with untrusted code, a challenge that extends far beyond blockchains. We introduce SCIF, a language for building smart contracts that are compositionally secure. SCIF is based on the fundamentally compositional principle of secure information flow, but extends this core mechanism to include protection against reentrancy attacks, confused deputy attacks, and improper error handling, even in the presence of malicious contracts that do not follow SCIF's rules. SCIF supports a rich ecosystem of interacting principals with partial trust through its mechanisms for dynamic trust management. SCIF has been implemented as a compiler to Solidity. We describe the SCIF language, including its static checking rules and runtime. Finally, we implement several applications with intricate security reasoning, showing how SCIF supports building complex smart contracts securely and gives programmer accurate diagnostics about potential security bugs.

## 1 INTRODUCTION

Smart contracts remain a promising platform for decentralized computation and storage, despite recent setbacks. However, they are perhaps the clearest demonstration of the difficulty of building secure software compositionally. Even an ever-expanding set of tools and best practices for smart contract development have failed to prevent numerous highly expensive vulnerabilities.

A core challenge is that a smart contract is not stand-alone program, but a piece of an ever-growing on-chain library of code, whose future interactions are impossible to predict. Modern smart contracts do not merely interact with a small fixed set of off-chain users. Blockchain applications often comprise multiple smart contracts whose complex protocols create opportunities for adversaries to subvert intended control flow. Consequently, attempts to implement such contracts have often resulted in subtle vulnerabilities, leading to billions of dollars in losses.

Recent work has shown that most value on Ethereum is placed in a small fraction of contracts [73]. While one might think that the security of other contracts is therefore less critical, we argue that this fact demonstrates that many users lack faith in most contracts and are only willing to entrust a few with valuable assets. It suggests that better methods are needed for obtaining assurance that contracts are secure, and for building contracts worthy of trust.

Existing automated tools for smart contract analysis based on model checking and fuzzing (e.g., [39, 43, 44, 50, 56, 65, 68, 79, 88]) have proved effective at finding numerous vulnerable, already deployed contracts. But as a tool for developers to build secure contracts, they are fundamentally limited by their lack of information about the security policies that contracts are intended to respect. Instead, they must rely on brittle, evolving heuristics about what resources are security-critical and about trust relationships between interacting contracts. As a result, existing tools frequently disagree about whether contracts are secure. For example, analyses usually consider native coins like Ether security-critical but often miss vulnerabilities associated with other resources [79].

Rather than relying on heuristics, our approach is to create a language that permits principled, compositional analysis of smart contracts. The aim is to guide developers to implement contracts with high security assurance with respect to broad classes of clearly defined security vulnerabilities. The SCIF (Smart Contract Information Flow) language is a new smart-contract language that

Authors' addresses: Siqiu Yao, Cornell University, Ithaca, New York, USA, yaosiqiu@cs.cornell.edu; Haobin Ni, Cornell University, Ithaca, New York, USA, haobin@cs.cornell.edu; Andrew C. Myers, Cornell University, Ithaca, New York, USA, andru@cs.cornell.edu; Ethan Cecchetti, University of Wisconsin–Madison, Madison, Wisconsin, USA, cecchetti@wisc.edu.

syntactically resembles Solidity but provides compositional security: it permits contracts to interact with confidence that their combination will be secure for all interacting contracts, even in the presence of untrusted unknown contracts. Given the high cost of smart contract audits [10, 11, 19], the cost of SCIF security annotations should be a small price to pay for the added assurance.

As a baseline protection, SCIF enforces secure information flow, ensuring that untrusted information cannot influence trusted information without explicit programmer authorization [9]. Information-flow analysis is useful in preventing improper influence in smart contracts [14], but it alone is not enough to guard against sophisticated attacks that exploit the interaction between multiple contracts. For instance, reentrancy attacks have already been shown to require additional mechanisms to constrain control flow [17, 18]. SCIF leverages information flow control to obtain strong defenses against a broader range of vulnerabilities. Its new features include the following:

- **Confused deputy prevention:** Confused deputy attacks (CDAs) are a long-known vulnerability of complex systems, and have quickly become a concern for smart contracts [24, 28]. One challenge has been the absence of a crisp definition of what constitutes a CDA. In this work, we develop a principled, more general definition and a new mechanism for comprehensively preventing CDAs.
- **Secure reentrancy:** Reentrancy vulnerabilities can occur in many settings [35], but have been particularly damaging in smart contracts. Dating back to The DAO [76], numerous reentrancy attacks have cost smart contracts hundreds of millions of dollars [26, 27, 29, 33, 72]. Best practices for coding smart contracts help but are insufficient to prevent damaging attacks [32]. The SeRIF calculus [18] showed that secure reentrancy is properly viewed as an information-flow property. SCIF shows how to harmoniously extend and integrate these ideas into a full language design.
- **Secure, atomic exception handling:** An important feature of smart contracts is that their effects on state can be reliably rolled back, particularly when unexpected errors occur. The default behavior of a failing contract is thus to do nothing. However, this exception mechanism only operates reliably at the level of a single contract. Despite the best practice of checking for failures whenever possible [31], failure to effectively reason about errors has led to damaging attacks [59, 94]. SCIF incorporates a novel and useful exception mechanism that distinguishes exceptions causing rollback from exceptions that do not, while enforcing a strong form of control-flow integrity.
- **Dynamic trust management:** The blockchain is not truly a zero-trust environment; functionality is made possible through mechanisms for trust and authentication. Moreover, contracts need to dynamically declare and revoke trust relationships as trust can evolve over time. Adding trust is required to interact with other contracts. Revoking is also necessary as many attacks [23, 25, 30] compromise private keys of trusted entities. Existing contracts invent their own mechanisms for trust and authentication; SCIF instead offers a customizable framework for trust and authentication management that allows contracts to specify, query, and update trust. Importantly, this mechanism is tightly integrated with the other security enforcement mechanisms in SCIF.

SCIF is targeted at smart contracts, but the lessons learned from securing code in a highly adversarial environment have value far beyond blockchains. The challenge of secure smart contracts is really the challenge of building secure software in a decentralized world with powerful adversaries.

# 2 BACKGROUND

## 2.1 The SCIF Threat Model

SCIF is intended to defend against powerful adversaries who need not follow the rules of SCIF. We assume adversaries can see the code and state of all deployed contracts. They also control some set of addresses, including principals and both SCIF and non-SCIF contracts. As SCIF contracts keep track of the addresses they trust and implicitly trust the principal that created them, adversaries are assumed to control any SCIF contract that trusts a principal they control. The adversaries may also define their own non-SCIF contracts that need not respect the rules of SCIF; these contracts can still interact with SCIF contracts by making calls or by having SCIF contracts call them by passing in their addresses as if they were SCIF contracts. Adversaries can initiate arbitrary transactions from any address they control. However, they may not forge calls to make it appear that they come from a principal (or contract) they do not control. And adversaries are only able to interact with SCIF contracts they do not control by making calls to them.

SCIF addresses many security concerns but is only indirectly helpful with some issues. It does not have any special support for reasoning about purely numeric issues such as overflow and rounding, though it does enforce validation of untrusted numeric values. SCIF has no control over transaction reordering, so it does not address concerns of maximal extractable value (MEV) [37, 77].

## 2.2 Integrity via Information Flow

The core security enforcement mechanism of SCIF is information flow control (IFC) [81]. Each expression has a security label $\ell$ reflecting the possible influences on its value, and SCIF uses a type system to identify and eliminate improper information flows at compile time. We write $\ell_1 \Rightarrow \ell_2$, read as "$\ell_1$ flows to $\ell_2$," if information with label $\ell_1$ can securely influence information with label $\ell_2$.

IFC is typically used to enforce data confidentiality, but it can also enforce integrity by preventing influence of trusted data by untrusted sources, as originally proposed by Biba [9] and implemented in some existing languages [64, 96].

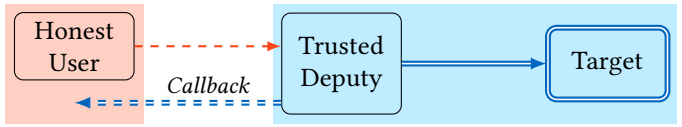
For instance, the following SCIF code snippet declares variables `x` and `y` with labels `untrusted` and `trusted`: It is safe for trusted information to affect untrusted information (`trusted` $\Rightarrow$ `untrusted`), so information flow from `trusted` to `untrusted` is permissible, as demonstrated in line 3, whereas the reverse direction (line 4) is prohibited.

```
1 uint{untrusted} x;
2 uint{trusted} y;
3 x = y;   // legal
4 y = x;   // illegal
```

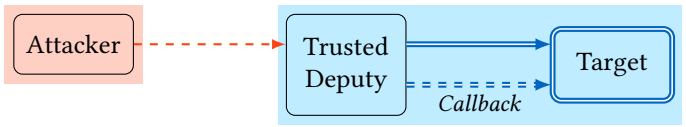
IFC type systems also track influences on control flow. Consider the code `if (x > 5) y = 0`. The assignment `y = 0` only executes if `x` is large enough, so `x` influences the value of `y`—which should not be allowed with the labels in the above snippet. To track these *implicit flows*, SCIF uses a standard *pc* (program-counter) label [81] that captures influences on control flow.

Through the lens of information flow, smart contract vulnerabilities often represent insecure flows in which untrusted information affects trusted information in unintended ways. In some cases, the insecure flow is obvious. For example, in 2017, an attacker used a publicly visible initialization method to set the trusted owner of Parity multi-sig wallets from attacker-controlled code [13, 17]. The SCIF type system is designed to prevent such information flows.

While IFC is a useful way to understand and prevent smart contract vulnerabilities, it is not a panacea. Strict enforcement of secure information flow guarantees noninterference [45], meaning untrusted code and data cannot affect trusted information in any way. However, most interesting contracts must permit untrusted users some limited influence on trusted data—which violates




(a) Intended use of a trusted deputy by an honest user




(b) Attacker confuses the deputy to exploit the target.

**Fig. 1.** Exploitation of a confused deputy. Dashed arrows denote calls where the user controls what is called, and double blue lines denote calls carrying (or interfaces requiring) the trusted authority of the deputy. The callback is dashed *and* blue; hence, the attacker can exploit the target.

noninterference. Such limited influence is supported in IFC systems through the downgrading operation of *endorsement* [96], in which trusted code can selectively boost the level of trust in particular information. Endorsement adds needed expressive power, but vulnerabilities can result from attacker exploitation of endorsement. In fact, since noninterference guarantees that the attacker is powerless, *all* corruption of trusted data by an untrusted attacker within a single transaction must involve some form of endorsement.

SCIF follows earlier IFC-based languages [40, 61, 64, 66, 97] by giving the expressive power needed to build arbitrary contracts through explicit endorsement annotations. However, endorsement is restricted to avoid mistakes: code can only endorse data up to the trust level of the code's own control context, preventing implicit endorsement of adversary influence on control flow.

The philosophy of SCIF is to prevent implicit endorsement. Making all endorsement explicit prompts programmers to think carefully about their use. By contrast, in Solidity [85], standard programming patterns implicitly endorse both data and control flow, resulting in vulnerabilities.

### 2.3 Confused Deputy Attacks

One particularly subtle class of attacks resulting from implicit endorsement is confused deputy attacks (CDA). A CDA occurs when an attacker manipulates a trusted party (the confused deputy) into misusing its authority, resulting in data corruption or leaks.

Figure 1 depicts a CDA. An untrusted user interacts with a trusted deputy, and the deputy interacts with a security-critical target and separately invokes a user-provided callback. In the intended use case (Figure 1a), an honest user provides a callback pointing to entities in the user's security domain, which cannot harm the target. In an attack (Figure 1b), the attacker instead provides a callback pointing to the target, which accepts the dangerous call because it comes from the trusted deputy—an implicit endorsement of the choice of what to call.

CDAs have resulted in damaging attacks on smart contracts. One emblematic CDA attacked Dexible [28], a token exchange. To efficiently swap tokens that may be difficult or impossible to exchange directly, Dexible users specify a *sequence* of swaps, exchanging the initial token for a second, the second for a third, and so forth. Figure 2 presents a simplified version of the Dexible code that performs a single intermediate exchange of ERC20 tokens [93]: the `swap` method allows users to perform a swap from an `amount` of `tokenIn` type tokens by invoking a separate exchange contract at address `router`. The user may also provide additional arguments to the exchange contract in

```
contract Dexible {
  function swap(uint amount, address tokenIn,
                address router, bytes routerData) external {
    if (IERC(tokenIn).transferFrom(msg.sender, address(this), amount)) {
      IERC(tokenIn).safeApprove(router, amount);
      (bool succ, ) = router.call(routerData);
      assert(succ, "Failed to swap");
    } else {
      revert("Insufficient balance");
} } }
```

**Fig. 2.** Simplified Solidity code for the Dexible bug.

```
contract Uniswap {
  Token tX, tY;

  function sellXForY(uint xSold) returns uint {
    uint prod = tX.getBal(this) * tY.getBal(this);
    uint yKept = prod / (tX.getBal(this) + xSold);
    uint yBought = tY.getBal(this) - yKept;

    assert tX.transferFrom(msg.sender, this, xSold);
    assert tY.transfer(this, msg.sender, yBought);
    return yBought;
} }
```

**Fig. 3.** Distilled Solidity code for the Uniswap bug.

`routerData`, specifying an arbitrary method to call along with additional arguments. In February 2023, an attacker exploited this functionality by inducing Dexible to call a token manager and transfer tokens from Dexible to the attacker [28]. Since the transfer request originated from Dexible itself—acting as a confused deputy—the token contract accepted the call and transferred the tokens.

The root vulnerability is that the Dexible code assumes that the user-specified call executes with only the user's authority, whereas the callee assumes the call conveys the full authority of the caller, Dexible. This mismatch in assumptions is the core confusion that creates a vulnerability: it causes an unexpected implicit endorsement of the control flow.

### 2.4 Reentrancy Vulnerabilities

Another style of attack exploiting endorsements is reentrancy, where an attacker unexpectedly *reenters* an application while it is in an intermediate state. A long string of reentrancy attacks have resulted in hundreds of millions of dollars of damage [26, 27, 29, 33, 72, 76].

The Uniswap token exchange fell victim to a reentrancy vulnerability in 2020 [72], showing that the combination of multiple contracts—each seemingly secure in isolation—can cause reentrancy vulnerabilities. Figure 3 shows a simplified segment of Uniswap's vulnerable code. The `sellXForY` function allows users to exchange tokens of type `X` for those of type `Y`. Uniswap determines the rate of exchange by holding constant the product of its balance of `X` and its balance of `Y`. Both Uniswap and its accompanying token contracts were originally thought reentrancy-secure because they follow the best-practice paradigm of checks–effects–interactions [86], but their combination unwittingly opens the door to reentrancy attacks. During the invocation of `transferFrom` at line 9, the client receives a notification, giving it control of execution and allowing an attacker

```
1  contract KoET {
2    address currentMonarch;
3    uint currentClaimPrice;
4
5    function claimThrone() external {
6      require(msg.value == currentClaimPrice);
7      uint compensation = calcCompensation(currentClaimPrice);
8      currentMonarch.send(compensation);
9      currentMonarch = msg.sender;
10     currentClaimPrice = calcNewClaimPrice(currentClaimPrice);
11 } }
```

**Fig. 4.** Distilled Solidity code for the KoET bug.

to opportunistically reenter `sellXForY`. Because the exchange rate depends on Uniswap's token balances and one transfer is still pending, Uniswap computes the exchange rate incorrectly in the reentrant call. The attacker then receives too favorable a rate, extracting tokens from Uniswap.

As Cecchetti et al. [18] observed, IFC offers a way to define and to prevent reentrancy vulnerabilities. Unlike confused deputy attacks, reentrancy attacks do not result from implicit endorsement but from attacker manipulation of explicit endorsement. A typical public method of a smart contract automatically endorses control flow, which is needed so that the smart contract can modify its own state. Because methods are annotated as public, this *auto-endorsement* [20] is explicit in the code.

Reentrancy vulnerabilities arise because in general, smart-contract state must obey some invariants for the contract to be correct, but those invariants may be temporarily broken while a method executes. If an attacker gains control of execution while the contract is in this inconsistent state (such as through a callback), they can engineer a reentrant call into a public method. Though the call comes from attacker integrity, the public method auto-endorses and accepts the call. Because contract invariants are temporarily broken, the contract might behave improperly.

### 2.5 Exception-based Vulnerabilities

Incorrect exception handling has long been a source of bugs and consequently, vulnerabilities. One study [95] concluded that "almost all (92%) of the catastrophic system failures are the result of incorrect handling of non-fatal errors explicitly signaled in software."

In Solidity, contracts can throw exceptions, which revert state changes within the ongoing transaction, and catch exceptions thrown in external calls. Solidity's type system, however, ignores exceptions: there is no static guarantee that exceptions are handled. In the absence of such verification, it is likely that developers will overlook exceptions. C# makes a similar design choice, but one study looking at a large C# codebase [15] found that 90% of potential exceptions remain undocumented. Without static checking, it is likely that many exceptions will not be handled (or even considered by developers), especially as smart contracts grow in complexity.

Particularly problematic is that Solidity's low-level mechanism for calling external contracts silently catches and discards exceptions by default. Not checking for such exceptions can open up vulnerabilities. A classic example is the "King of the Ether Throne (KoET)" [60], a smart contract application whose participants compete for a prize by initiating a transaction of greater value than the current prize. Figure 4 has a simplified version of the `claimThrone` method within KoET. At line 8, the method performs a low-level call to compensate the previous winner. However, if this call throws an exception, it merely results in a return value of `false` rather than propagating the exception. So if the call fails for any reason, the compensation remains in the KoET contract, and

the method continues, updating a new winner. Such a vulnerability led to the temporary shutdown and subsequent reboot of the KoET contract in 2016 [59]. A clear and principled way to handle exceptions is needed to prevent such vulnerabilities.

## 3 OVERVIEW OF SCIF

SCIF contracts are high-level programs annotated with information flow labels, with a syntax and run-time semantics similar to that of Solidity. The addition of information flow labels allows the SCIF language to effectively identify and eliminate potential vulnerabilities.

We illustrate the SCIF language through examples showing how its new features address real-world vulnerabilities.

### 3.1 Information Flow Labels

The SCIF type system tracks information flow statically, as described in Section 2.2. Because smart contracts often have their own primitive security concerns and each has an existing unique identifier—its on-chain address—SCIF labels are elements of the free distributive lattice over the set of contract addresses. This structure also allows any address to be interpreted as a label, though not all labels are addresses. For instance, `this` denotes the integrity level of the current contract, which is generally the most trusted possible label from that contract's perspective. The label `any` represents the least trusted label, for data that may be influenced by anyone. In SCIF, off-chain users are a special category of smart contracts, with unique on-chain addresses and storage for their cryptocurrency balances. While they can initiate method calls on other contracts, they do not host callable methods themselves.

To reduce annotation burden, SCIF assigns default labels to fields and method arguments, and the compiler infers most labels inside method bodies.

In addition to labels on the arguments and return type, SCIF method signatures are annotated with up to three labels, with the syntax $\{pc_{\text{ex}} \rightarrow pc_{\text{in}}; \ell_{\text{L}}\}$. The *external pc label* $pc_{\text{ex}}$ specifies the control-flow integrity required to call the method. It also serves as the default label for method arguments. If provided, the optional $pc_{\text{in}}$ label separately denotes the *internal pc label*, specifying the integrity of the control flow when the method body begins execution. When $pc_{\text{ex}}$ is less trusted than $pc_{\text{in}}$, the method performs an auto-endorsement on entry, explicitly endorsing the control flow. When $pc_{\text{in}}$ is not specified, it defaults to $pc_{\text{ex}}$. Finally, the *lock label* $\ell_{\text{L}}$, adopted from Cecchetti et al. [18], specifies the *reentrancy lock* integrity this method respects: the key annotation used to prevent reentrancy attacks. If not specified, $\ell_{\text{L}}$ defaults to $pc_{\text{in}}$. If only $pc_{\text{ex}}$ is specified, all three labels will be the same.

*Example* (Parity Wallet). To see how information flow labels can help, we examine an Ethereum wallet produced by Parity Technologies, which fell victim to two separate attacks in 2017, each costing over $30 million [13, 71]. Though the second attack is more famous, we focus on the first. The wallet code was split into two contracts: a library housing core operations, and an instance contract with user-specific data. The instance wallet delegated to the library using Ethereum's `delegatecall` instruction, executing library code in the memory space of the instance wallet. Unfortunately, the interaction exposed a serious bug. The library contract contained a public method that initialized the owner of the wallet with no authorization checks. The attacker managed to call this method via `delegatecall`, and change the owner of the wallet.

Figure 5 shows a simplified piece of the wallet as code in SCIF, leveraging its inheritance and default labeling features. In the `WalletLibrary` library contract, the sensitive `owner` field has the default label: `this`. The `initOwner` method is not marked as public, so it defaults to private and

```
contract WalletLibrary {
  address owner;
  void initOwner(address newOwner) {
    owner = newOwner;
  }
}
contract Wallet extends WalletLibrary { ... }
```

**Fig. 5.** Simplified Parity Wallet implementation in SCIF.

requires an external *pc* label of `this`. These labels ensure the control flow and argument are sufficiently trusted to alter `owner`.

As `Wallet` extends `WalletLibrary`, it seamlessly inherits all of its field members and methods. Enforcing the labels then makes the original attack impossible. Because the external *pc* label of `initOwner` is `this`, the calling control flow must already be trusted by the instance of `Wallet`. An untrusted attacker cannot call `initOwner`.

It is possible to write a vulnerable version of `initOwner` in SCIF, but doing so requires additional actions by the programmer, which should make them think twice. First, they must mark `initOwner` as `@public`. Second, they must either explicitly mark `owner` as untrusted—overriding the default label of `this` in an obviously unsafe way—or explicitly endorse `newOwner`, as in the following code.

```
@public void initOwner(address newOwner) {
  owner = endorse(newOwner, sender -> this);
}
```

This intentionally verbose pattern makes it clear that anyone can modify `owner`. In essence, SCIF defaults to a secure implementation, requiring the programmer to explicitly opt in to vulnerabilities, while in Solidity the difference between insecure and secure implementations is much less obvious.

### 3.2 CDA Prevention

To understand how SCIF can prevent both the Dexible attack introduced in Section 2.3 and CDAs more generally, let us look at CDAs more abstractly. Recall from Section 2.3 that CDAs follow a particular pattern: an untrusted attacker tricks a trusted deputy, usually through a callback, into tricking a target into performing a dangerous action on a security-critical resource. From this perspective, a CDA violates information flow security because the attacker has influenced trusted actions of the target without any explicit endorsement. Conversely, correct enforcement of information flow security—ruling out implicit endorsements—eliminates CDAs. Because SCIF's type system enforces IFC, no extra work would be needed if all contracts involved were well-typed SCIF contracts.

However, in an open blockchain system, we cannot assume untrusted contracts are well-typed—or even written in SCIF. Absent an additional run-time defense, an attacker can pass a SCIF contract a callback of a different type than expected. From an IFC perspective, this *type confusion* is essential to mounting a CDA, because confusion of types that talk about information flow enables *information flow* confusion as well.

Consider the Dexible attack discussed in Section 2.3. Dexible's public `swap` operation invoked a user-provided callback, and an attacker provided the token manager's `transfer` method as that callback. Dexible, acting as a confused deputy, directly requested the target token manager transfer Dexible's funds to the attacker, a request the token manager faithfully executed. In SCIF, the labels on the type signature of the token manager's `transfer` method would be different from those on a valid user-provided callback. The token manager requires the full authority of Dexible, while a

```
contract Dexible {
  exception FailedSwap();

  @public
  void swap(IERC20 tokIn, IERC20 tokOut, final IExchange router, uint amt)
      throws (FailedSwap) {
    atomic {
      tokIn.approve(sender, router, amt);
      assert router => sender;
      router.exchange(sender, tokIn, tokOut, amt);
    } rescue * {
      throw FailedSwap();
} } }

interface IExchange {
  void exchange{user -> this}(final address user, IERC20{user} tokIn,
                              IERC20{user} tokOut, uint{user} amt);
}
```

**Fig. 6.** Simplified Dexible implementation in SCIF.

valid callback should require only the user's authority. Consequently, if all involved contracts were well-typed, the callback could not point to the token manager and the CDA would be impossible.

Unfortunately, Dexible and the token manager both being well-typed is not sufficient to prevent this attack. Because type confusion is caused by untyped attacker code, dynamic checking is needed to catch it. However, type-checking all references passed at run time is infeasible.

Fortunately, avoiding CDA attacks only requires the caller and callee (deputy and target) to agree on the run-time type of *the method being called*, a localized check that is much simpler to perform. The caller can pass what it believes to be the full method signature, including information flow labels, as an additional implicit argument when it calls a method. The callee can then check that signature against what it knows to be its own signature. If the caller is sufficiently trusted to invoke the method but the expected signature does not match the true signature, there might be a confused deputy attack in progress and the callee (target) can abort the call.

The SCIF compiler extends the standard smart contract internal dispatch mechanism [85] to minimize the complexity and cost of this check. Solidity uses a method's name and argument base types to perform method dispatch, but SCIF uses the full method signature, including labels. Successful method dispatch then ensures that the caller and the callee agree about the type of the method, eliminating type confusion. Unlike Solidity, SCIF disallows direct low-level calls that take a raw `bytes` argument including dispatch information. Instead all calls must go through a declared interface, insuring that calling code directly specifies the labels it expects. Section 5 describes this implemention in more detail.

*Example* (Dexible). Figure 6 shows a CDA-secure Dexible contract written in SCIF. There are a few key differences from the Solidity implementation shown in Figure 2. First, all operations in the `swap` method are inside an `atomic–rescue` block. This block functions similarly to a `try–catch` block with standard exceptions, except (1) it catches failures instead of exceptions, which SCIF considers different, and (2) any changes made in the `atomic` block are reverted if the body fails. That way we know that the swap either entirely succeeds or entirely fails. Section 3.4 provides detail on how SCIF handles exceptions and failures.

Second, the `exchange` method of the `IExchange` interface includes explicit labels specifying that the router only requires the integrity of the user to execute. Based on this interface and using the approach described above, the SCIF compiler automatically generates the dynamic type check as part of the call to `router.exchange` on line 10. This check catches any type confusion on the call and prevents the original attack while retaining Dexible's ability to interact with user-provided router contracts.

Moreover, while a programmer could write a signature for `exchange` that matches the token manager's `transfer` signature, allowing the confused deputy attack to pass the dynamic check, Dexible would not compile. The call would require `this` integrity, but the `router` argument to Dexible's `swap` has `sender` integrity by default, so a standard static IFC check would reject the call.

Notably, these concerns about attacker-induced type confusion do not apply to IFC systems that only transmit simple data. The data is untrusted, and hence cannot influence trusted actions. For callbacks, however, type confusion allows a method that *does* convey trusted authority to appear as one that does not, requiring our additional checks. In fact, type confusion can be used to launch new, more subtle forms of CDAs. For example, an attacker might lie about the reentrancy lock label of a method and use the deputy to launch a reentrancy attack. By eliminating type confusion, SCIF prevents these attacks as well.

### 3.3 Reentrancy Attack Prevention

SCIF adopts and improves the mechanism from SeRIF [18] for preventing reentrancy attacks, combining static and dynamic *reentrancy locks* to prevent reentrant auto-endorsement, so reentrant calls do not introduce new attacks.

SeRIF requires any untrusted call made without dynamic locks to be in tail position, forbidding any subsequent operations. This approach prevents dangerous reentrancy, but it also enforces two limiting constraints.

(1) Trusted values computed before an untrusted call cannot be returned afterward.
(2) In auto-endorse functions, untrusted operations cannot execute after an untrusted call returns, even though they inherently cannot create reentrancy concerns.

SCIF maintains the security of SeRIF's reentrancy protection, while improving the precision to allow for these two useful code patterns. First, methods define their return values by assigning to a special `result` variable. A method must assign to this variable on every return path. The usual syntax `return` *e* is just syntactic sugar for assigning `result` = *e* and then returning. Second, after an untrusted call, the control-flow integrity (the *pc* label) is modified, restricting future operations to only those that cannot violate high-integrity invariants. Neither of these changes can introduce reentrancy concerns, and both simplify programs.

*Example* (Uniswap). Recall from Section 2.4 that the Uniswap exchange had a reentrancy vulnerability stemming from a complex interaction with the exchange and tokens. Figure 7 shows how we might use SCIF to implement `sellXForY` and specify the standard ERC-20 token interface [93].

Following the ERC-20 standard [93], interface `IERC20` includes a `transfer` method to directly transfer tokens owned by the caller and a `transferFrom` method to transfer tokens whose owner has previously authorized the caller to move them. To reflect these expectations, `transfer` requires the integrity of `from`, the user whose tokens are moving, and auto-endorses the control flow to `this`, the integrity of the token contract, which is necessary to modify token balances. However, `transferFrom` allows any caller, but only auto-endorses to `from`, enabling adjustments to the allowances of tokens owned by `from` and proving sufficient integrity to call `transfer` and actually

```
1  contract Uniswap {
2    IERC20 tX;
3    IERC20 tY;
4
5    @public uint sellXForY(final address buyer, uint xSold) {
6      uint prod = tX.getBal(this) * tY.getBal(this);
7      uint yKept = prod / (tX.getBal(this) + xSold);
8      uint yBought = endorse(tY.getBal(this) - yKept, sender -> this);
9
10     lock(this) {
11       assert tX.transferFrom(buyer, this, xSold);
12       assert tY.transfer(this, buyer, yBought);
13     }
14     return yBought;
15   }
16 }
17
18 interface IERC20 {
19   @public bool{this} transfer{from -> this; any}(final address from, address to, uint amount);
20
21   @public bool{from} transferFrom{sender -> from; any}(final address from, address to,
22                                                        uint amount);
23 }
```

**Fig. 7.** Simplified Uniswap implementation in SCIF.

move the tokens. Since both methods may invoke untrusted confirmation methods provided by contracts `from` and `to`, the reentrancy lock label for both methods is `any`.

In the Uniswap contract, `sellXForY` is meant to be a publicly-accessible method that must modify trusted state, so we annotate it as `@public` and the default labels for public methods: `{sender -> this; this}`. That is, `sellXForY` is an entry point anyone can call that auto-endorses to `this`, and it promises not to call untrusted code without a dynamic lock (reentrancy lock label `this`).

Because `transferFrom` respects no reentrancy locks but `transfer`, which requires high integrity, is called after `transferFrom` returns, the dynamic lock on line 10 is necessary for security and correctly required by the type system. We could remove this lock if we changed the `IERC20` methods to maintain high-integrity locks, which would then preclude notification of untrusted parties during transfers.

To see the value of SCIF's improved flexibility over SeRIF, consider the following implementation of the `IERC20` `transfer` method.

```
1 @public
2 bool transfer{from -> this; any}(final address from, final address to, uint amount) {
3   ... // check and update balances
4   result = true;
5   assert from.confirmSent(to, amount);
6   assert to.confirmReceived(from, amount);
7 }
```

Without resorting to expensive dynamic locks, this method securely returns a trusted boolean through early assignment to `result` (line 4) before executing two untrusted calls. Because neither `confirmSent` nor `confirmReceived` requires high integrity to invoke, these calls can safely execute

in sequence, even though the first does not maintain reentrancy locks. SeRIF allows neither pattern. Instead `transfer` must be split across multiple methods, and there is no way to return a high-integrity boolean without dynamic locks.

### 3.4 Exception Handling

SCIF differentiates between *exceptions* and *failures*:

- **Exceptions** define alternative execution paths. SCIF exceptions behave similarly to exceptions in languages like Java. They leave state changes in place and can be managed with standard `try-catch` blocks. Methods must declare in their signature any exceptions that they may throw and not catch internally. These declarations help programmers avoid unexpected exceptions and enable static analysis of the security of exceptional control flow.
- **Failures** represent unrecoverable errors, such as out-of-gas or stack overflow. Failures ensure that any state changes made in the offending method prior to the failure are entirely rolled back. SCIF does allow handling of failures using an `atomic-rescue` syntax. These blocks are similar to `try-catch` blocks, except that if the body of the `atomic` block produces a failure, any changes are rolled back to the beginning of the `atomic` block. Uncaught exceptions that reach an `atomic` barrier transform into failures, causing rollback.

By distinguishing between exceptions and failures, SCIF offers developers finer-grained control over error handling, improving robustness, clarity, and predictability of code.

*Example* (KoET). The SCIF implementation of KoET, shown in Figure 8, appears nearly identical to the Solidity implementation, but is much more robust. SCIF proactively eliminates vulnerabilities by disallowing the implicit disregard of failures, aligning with best practices [31]. In SCIF, a failure at line 8 automatically reverts all state changes in `claimThrone` and propagates to the caller, preventing the original attack.

*Example* (Town Crier). Town Crier (TC) [100] is an authenticated data feed backed by trusted hardware, providing data to smart contracts on request. The `deliver` method (Figure 9) delivers processed requests from the trusted hardware to the requester. It marks the request as delivered, sends the operator the request fee, and delivers the data through a user-supplied callback. In this case, if the user-supplied callback (line 14) fails, TC needs to keep the fee, so it must *not* roll back the entire call. Hence, it wraps the line 14 in an `atomic-rescue` block to catch the failure, and intentionally discards it. Failure rolls back the user-supplied operation, but not fee delivery.

```
1  contract KoET {
2    address currentMonarch;
3    uint currentClaimPrice;
4
5    @public void claimThrone() {
6      assert value == currentClaimPrice;
7      uint compensation = calcCompensation(currentClaimPrice);
8      send(currentMonarch, compensation);
9      currentMonarch = sender;
10     currentClaimPrice = calcNewClaimPrice(currentClaimPrice);
11 } }
```

**Fig. 8.** Simplified KoET implementation in SCIF.

```
contract TownCrier {
  User requester;
  address sgx;
  final uint FEE = 100;
  exception NoPendingRequest();

  @public void deliver{this; any}(bytes data) throws (NoPendingRequest) {
    if (requester == 0) {
      throw NoPendingRequest();
    }
    requester = 0;
    send(sgx, FEE);
    atomic {
      requester.callback(data);
    } rescue * {
      // Do Nothing. The callback reverts but TownCrier keeps the fee.
} } }
```

**Fig. 9.** Town Crier implementation snippet in SCIF.

As these examples show, SCIF encourages explicit, intentional failure management, and ensures that failures are not ignored implicitly. It helps programmers to handle failures robustly.

### 3.5 Dynamic Integrity Checks

SCIF contracts may use two types of dynamic trust checks: programmer-specified, and automatically generated. The expression $e_1$ `=>` $e_2$ dynamically checks whether $e_1$ flows to $e_2$. This check are useful when a specific flow is required. For instance, line 9 of the Dexible code in Figure 6 has the check `assert router => sender`, to ensure that execution only occurs when the caller trusts `router`. Notably, after this assertion, the compiler type-checks the rest of the method under the safe assumption that `router` $\Rightarrow$ `sender`, enabling information flows that otherwise would be disallowed.

SCIF also automatically generates dynamic checks in two places. In an open system, anyone can call a public method, so SCIF cannot statically ensure the caller is trusted at the method's external *pc* label. Instead, public methods dynamically check that the caller has sufficient integrity. This check is not needed if the external *pc* label is `sender` or `any`, but for any other $pc_{\text{ex}}$, SCIF automatically inserts a dynamic check equivalent to `assert sender =>` $pc_{\text{ex}}$ at the top of the method.

### 3.6 Contract Interfaces

Dynamic integrity checks and other run-time management are performed by the SCIF contract itself. Each contract must implement the `Contract` interface (Figure 10); compiled code generates calls to this interface, but ordinary code cannot call the methods.

- The role of the `trusts` method is to determine whether *this* contract believes `a => b`.
- Auto-endorse methods invoke `bypassLocks` to check that the caller is trusted enough to safely bypass any existing dynamic reentrancy locks.
- Methods `acquireLock` and `releaseLock` manage reentrancy locks. These methods are used to implement `lock(l) {...}` blocks.

SCIF provides a simple default implementation of `Contract` called `ContractImpl`, as well as more complex implementations. However, programmers may freely provide their own implementations,

```
interface Contract {
  @public bool trusts(address a,
                      address b);
  @public bool bypassLocks(address l);
  @public bool acquireLock(address l);
  @public bool releaseLock(address l);
}
```

**Fig. 10.** Interface `Contract`

```
interface ManagedContract extends Contract {
  @public bool addTrust(address trustee);
  @public bool revokeTrust(address trustee);
}

interface ExternallyManagedContract
    extends ManagedContract {
  @public bool directlyTrusts(address);
  @public address[] directTrustees();

  @public TrustManager trustManager();
  @public LockManager lockManager();
}
```

**Fig. 11.** Managed-contract interfaces

because the interface design limits the danger of buggy or malicious implementations to contracts that have a bad implementation (or that trust others that do).

*3.6.1 External Trust Management.* The `Contract` interface is sufficient for contracts whose trust relationships are fixed over time, but some trust relationships evolve. To support this feature, SCIF provides two extended interfaces shown in Figure 11. `ManagedContract` adds two simple methods: `addTrust` and `revokeTrust`, which add or remove addresses from the set that the current contract trusts. In the presence of complex trust relationships, implementing decentralized trust checks may be complex and expensive. SCIF therefore supports outsourcing this work to trusted managers through the `ExternallyManagedContract` interface. The `directTrusts` and `directTrustees` methods only concern who the contract trusts *directly*, and leave *indirect* trust, implied by transitivity, to an external `TrustManager` contract.

`TrustManager` and `LockManager` allow a contract to outsource the operations of `ManagedContract`. Specifically, `TrustManager` includes the trust-related methods (`trusts`, `addTrust`, and `revokeTrust`), while `LockManager` includes the lock-related methods (`bypassLocks`, `acquireLock`, and `releaseLock`). So, an `ExternallyManagedContract` implementation can conveniently implement its management method as pass-throughs to the correct manager.

These interfaces are designed to be flexible. Simple managers can resolve queries locally, conservatively (and safely) assuming there is no trust they are unaware of. More complicated managers can communicate with each other in a decentralized fashion, searching for trust between contracts they do not manage. As with implementations of `Contract`, a contract only relies on the correctness of its own manager and the managers of other contracts it trusts.

## 4 FORMALIZING CORE SCIF

To more precisely describe SCIF, we define a simplified version called Core SCIF. Core SCIF is an object-oriented core calculus. It extends SeRIF [18] with support for exceptions, transactional failures, and more flexible programming paradigms, as described in Section 3. SeRIF, in turn, is an extension of Featherweight Java [54] augmented with standard mutable references [74, Chapter 13], information-flow labels, and reentrancy protection.

Figure 12 presents the syntax of Core SCIF. Integrity labels in SCIF may be the constants this, any, a contract address $\alpha$, or conjunctions or disjunctions of other labels. All types carry an information flow label to track their integrity. Contracts, constructors, and method declarations are formalized

$$
\begin{array}{rcl}
\ell & ::= & \mathsf{this} \mid \mathsf{any} \mid \alpha \mid \ell \vee \ell \mid \ell \wedge \ell \\
\tau & ::= & \mathsf{unit}^{\ell} \mid \mathsf{bool}^{\ell} \mid (\mathsf{ref}\ \tau)^{\ell} \mid C^{\ell} \mid ex^{\ell} \\
Con & ::= & \mathsf{contract}\ C\ \mathsf{extends}\ C\ \{\overline{f}:\overline{\tau}\ ;\ \overline{Ex}\ ;\ K\ ;\ \overline{M}\} \\
K & ::= & C(\overline{f}:\overline{\tau})\ \{\mathsf{super}(\overline{f})\ ;\ \mathsf{this}.\overline{f} = \overline{f}\} \\
Ex & ::= & \mathsf{exception}\ ex(\overline{x}:\overline{\tau}) \\
M & ::= & \tau:\ell\ m\{\ell \gg \ell; \ell\}(\overline{x}:\overline{\tau})\ \mathsf{throws}\ \overline{ex^{\ell}}\ \{e\} \\
v & ::= & x \mid () \mid \mathsf{true} \mid \mathsf{false} \mid ex(\overline{v}) \mid \iota \mid \alpha \\
o & ::= & v \mid \mathsf{throw}\ v \mid \mathsf{fail}\ v \\
e & ::= & o \mid \mathsf{ref}\ v\ \tau \mid\ !v \mid \mathsf{new}\ C(\overline{v}) \mid (C)v \mid v.f \mid v.m(\overline{v}) \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \\
 & \mid & \mathsf{if}_{pc}\ v\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{if}_{pc}\ (v \Rightarrow v)\ \mathsf{then}\ e\ \mathsf{else}\ e \\
 & \mid & \mathsf{endorse}\ v\ \mathsf{from}\ \ell\ \mathsf{to}\ \ell \mid \mathsf{lock}\ \ell\ \mathsf{in}\ e \\
 & \mid & \mathsf{try}\ e\ \mathsf{catch}\ x:ex\ e \mid \mathsf{atomic}\ e\ \mathsf{rescue}\ x\ e
\end{array}
$$

**Fig. 12.** Syntax for Core SCIF

$$
\text{[E-Call]}\ \frac{\begin{array}{ccc} O(\alpha) = C(\overline{v}) & mbody(C,m) = (\overline{x}, pc_1 \gg pc_2, e, \overline{ex}) & \\ \mathcal{M} = \mathcal{M}', \ell'_m \quad \ell'_m \Rightarrow pc_1 & \bigwedge_{\ell \in L}(pc_1 \Rightarrow pc_2 \vee \ell) & e' = e[\overline{x} \mapsto \overline{w}, \mathsf{this} \mapsto \alpha] \end{array}}{\langle \alpha.m(\overline{w}) \mid C\rangle \longrightarrow \langle \mathsf{return}_{\overline{ex}}(e'\ \mathsf{at\text{-}pc}\ pc_2) \mid C[\mathcal{M}, \alpha/\mathcal{M}]\rangle}
$$

$$
\text{[E-Atomic]}\ \frac{}{\langle \mathsf{atomic}\ e_1\ \mathsf{rescue}\ x\ e_2 \mid C\rangle \longrightarrow \langle \mathsf{trans}\ e_1\ \mathsf{rescue}\ x\ e_2 \mid C[S, \sigma/S]\rangle}
$$

$$
\text{[E-AtomicCommit]}\ \frac{S = S', \sigma'}{\langle \mathsf{trans}\ v\ \mathsf{rescue}\ x\ e \mid C\rangle \longrightarrow \langle v \mid C[S/S']\rangle}
$$

$$
\text{[E-AtomicRescued]}\ \frac{S = S', \sigma'}{\langle \mathsf{trans}\ (\mathsf{fail}\ v)\ \mathsf{rescue}\ x\ e \mid C\rangle \longrightarrow \langle e[x \mapsto v] \mid C[\sigma'/\sigma; S'/S]\rangle}
$$

**Fig. 13.** Selected small-step semantic rules for SCIF.

similarly to SeRIF [18]. New features include exceptions and failures, and a more accurate treatment of contract addresses.

SCIF expressions are mostly standard, with a few interesting notes. To simplify the language, expressions generally are (open) values, which lack subexpressions. The exception is let-expressions, which are used to encode sequential composition. Second, SCIF has conditionals to dynamically test trust relationships $v_1 \Rightarrow v_2$, interpreting $v_1$ and $v_2$ as contract addresses. The type system then assumes the flow exists in the then-branch.

SCIF distinguishes exceptions from transactional failures. As in the surface language, exceptions and try–catch behave like typical exceptions in other languages: state changes persist regardless of whether the exception is caught. An atomic–rescue, however, creates an atomic transaction that is entirely reverted if a failure occurs, whether or not it is rescued.

### 4.1 Operational Semantics

The operational semantics of Core SCIF is defined as a small-step operational semantics similar to that of SeRIF [18], with extensions to support exceptions, failures, dynamic trust checks, and CDA prevention. Figure 13 shows selected operational semantics rules.[1]

The calculus tracks *pc* labels dynamically so that SeRIF's definition of reentrancy still applies. To support this tracking and simplify the operational semantics, if statements include a syntactic *pc* label for the branches. In practice, it is easy to infer automatically.

A *semantic configuration* is a tuple $C = (CT, O, \sigma, S, \mathcal{M}, L)$ with the following elements: $CT$ is a contract table that holds all contract types and code; $O$ is the contract heap, mapping addresses to objects; $\sigma$ is the mutable heap, mapping locations to values; $S$ is a list of saved memories used to implement transactions with atomic–rescue; $\mathcal{M}$ tracks the integrity of the executing contract instance; $L$ is a list that tracks the dynamically locked integrity. To simplify notation, we refer to components of $C$ freely when only one configuration is in scope. We write $C[X/L]$ to denote $(CT, O, \sigma, S, \mathcal{M}, X)$, and similarly for the other elements.

Compared to SeRIF, SCIF does not track the integrity of the executing code. Rather, it tracks the integrity of the executing contract instances. For example, the rule E-Call checks whether the caller is at least as trusted as the external pc label $pc_1$ of the callee.

Rules E-AtomicCommit and E-AtomicRescued Whereas a try–catch block keeps state changes made by the try block, an atomic–rescue block rolls back all state changes made by the atomic block. The typing rules disallow uncaught exceptions inside atomic blocks, so there is no semantic rule to handle them.

### 4.2 Type System

The type system of Core SCIF also extends that of SeRIF [18] to support features including dynamic trust management and exception handling.

SCIF has separate typing judgments for values and expressions. Value judgments take the form $\Sigma; \Gamma; \mathcal{T} \vdash v : \tau$, where $\Sigma$ is a heap type, mapping heap locations and contract addresses to types, $\Gamma$ is a standard typing environment mapping variables to types, and $\mathcal{T}$ is a set of trust relationships that have been checked dynamically. When a program dynamically checks that $\ell_1 \Rightarrow \ell_2$, the type system needs to include this information when checking future flows. We therefore include $\mathcal{T}$ in the typing judgment and write $\mathcal{T} \vdash \ell_1 \Rightarrow \ell_2$ to check that the flow holds in the current environment.[2]

Expression judgments $\Gamma; \mathcal{T}; pc; \ell_L \vdash e : \tau \dashv \Psi$ are a bit more complicated. Here $\Gamma$ and $\mathcal{T}$ are as above and $pc$ is the standard program-counter label. The lock label $\ell_L$, taken from SeRIF, is the reentrancy *input lock* the expression must maintain to continue execution with the same integrity. Finally, because SCIF supports exceptions, $e$ may terminate in multiple different ways—normally or through one of multiple possible exceptions. Following Jif [66], the context $\Psi$ tracks the integrity of these different possible termination paths. A path can either be normal termination ($\underline{\text{n}}$), an exception ($ex$), or $\underline{\text{fl}}$, denoting a failure. SCIF must track both the integrity of the control flow and reentrancy locks, so $\Psi$ maps possible termination paths to pairs of labels $(pc, L)$.

Figure 14 shows selected typing rules for Core SCIF.[3] We write $\Psi[p].pc$ and $\Psi[p].L$ to denote the values for a path $p$, and $\Psi_1 \vee \Psi_2$ as the pointwise join of two mappings, including any values for paths $p$ where only one of $\Psi_1[p]$ and $\Psi_2[p]$ is defined.

[1] The remaining rules can be found in Appendix A.

[2] Mathematically, we quotient our original free distributive lattice over addresses (see Section 3.1) by the relationships in $\mathcal{T}$ and check the flow in the resulting quotient lattice.

[3] The remaining rules are available in Appendix A.

$$[\textsc{Let}]\ \frac{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\text{L}} \vdash e_1 : \tau_1 \dashv \Psi_1 \qquad \ell'_{\text{L}} = \Psi_1[\underline{\text{n}}].L \vee \ell_{\text{L}} \qquad \Sigma;\Gamma, x:\tau_1;\mathcal{T};pc';\ell'_{\text{L}} \vdash e_2 : \tau_2 \dashv \Psi_2 \quad \mathcal{T} \vdash \Psi_1[\underline{\text{n}}].pc \Rightarrow pc' \qquad \mathcal{T} \vdash \Psi_1[\underline{\text{n}}].L \Rightarrow \ell_{\text{L}} \vee pc' \qquad \mathcal{T} \vdash \Psi_1[\underline{\text{n}}].L \Rightarrow \Psi_2[\underline{\text{n}}].L}{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\text{L}} \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 \dashv (\Psi_1 \setminus \underline{\text{n}}) \vee \Psi_2}$$

$$[\textsc{IfTrust}]\ \frac{\Sigma;\Gamma;\mathcal{T} \vdash v_1 : C_1^{\ell} \qquad \Sigma;\Gamma;\mathcal{T}, v_1 \Rightarrow v_2;pc \vee \ell;\ell_{\text{L}} \vdash e_1 : \tau \dashv \Psi_1 \quad \Sigma;\Gamma;\mathcal{T} \vdash v_2 : C_2^{\ell} \qquad \Sigma;\Gamma;\mathcal{T};pc \vee \ell;\ell_{\text{L}} \vdash e_2 : \tau \dashv \Psi_2 \quad \mathcal{T} \vdash \ell \triangleleft \tau}{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\text{L}} \vdash \text{if}_{pc}\ (v_1 \Rightarrow v_2) \text{ then } e_1 \text{ else } e_2 : \tau \dashv \Psi_1 \vee \Psi_2}$$

$$[\textsc{Call}]\ \frac{mtype(C,m) = \overline{\tau_a} \xrightarrow{pc_1 \twoheadrightarrow pc_2;L} \tau_0 : \ell_{\underline{\text{n}}}, \overline{ex^{\ell_e}} \qquad \Sigma;\Gamma;\mathcal{T} \vdash v : C^{\ell} \quad \Sigma;\Gamma;\mathcal{T} \vdash \overline{v_a} : \overline{\tau_a} \qquad \mathcal{T} \vdash \tau_0 <: \tau \qquad \mathcal{T} \vdash pc \vee \ell \Rightarrow pc_1 \qquad \mathcal{T} \vdash pc_1 \Rightarrow pc_2 \vee \ell_{\text{L}} \qquad \mathcal{T} \vdash \ell \triangleleft \tau \quad \ell_{\underline{\text{fl}}} = \ell_{\underline{\text{n}}} \vee \ell \vee \bigvee \overline{\ell_e} \qquad \ell'_{\text{L}} = L \vee \ell \qquad \Psi = \left\{\underline{\text{n}} \mapsto (\ell_{\underline{\text{n}}} \vee \ell, \ell'_{\text{L}}), \underline{\text{fl}} \mapsto (\ell_{\underline{\text{fl}}}, \ell'_{\text{L}}), \overline{ex} \mapsto \left(\overline{\ell_e} \vee \ell, \ell'_{\text{L}}\right)\right\}}{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\text{L}} \vdash v.m(\overline{v_a}) : \tau \dashv \Psi}$$

$$[\textsc{Throw}]\ \frac{\Sigma;\Gamma;\mathcal{T} \vdash v : ex^{\ell}}{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\text{L}} \vdash \text{throw } v : \tau \dashv \{ex \mapsto (pc \vee \ell, \ell_{\text{L}})\}}$$

$$[\textsc{TryCatch}]\ \frac{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\text{L}} \vdash e : \tau \dashv \Psi \qquad \mathcal{T} \vdash \Psi[ex].L \Rightarrow \ell_{\text{L}} \quad pc' = \Psi[ex].pc \qquad \Sigma;\Gamma, x : ex^{pc'};\mathcal{T};pc';\ell_{\text{L}} \vdash e' : \tau \dashv \Psi'}{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\text{L}} \vdash \text{try } e \text{ catch } x : ex\ e' : \tau \dashv (\Psi \setminus ex) \vee \Psi'}$$

$$[\textsc{AtomicRescue}]\ \frac{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\text{L}} \vdash e : \tau \dashv \Psi \qquad \text{dom}(\Psi) \subseteq \{\underline{\text{n}}, \underline{\text{fl}}\} \quad \mathcal{T} \vdash \Psi[\underline{\text{fl}}].L \Rightarrow \ell_{\text{L}} \qquad pc' = \Psi[\underline{\text{fl}}].pc \qquad \Sigma;\Gamma, x : \text{fl}^{pc'};\mathcal{T};pc';\ell_{\text{L}} \vdash e' : \tau \dashv \Psi'}{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\text{L}} \vdash \text{atomic } e \text{ rescue } x : \text{fl}\ e' : \tau \dashv (\Psi \setminus \underline{\text{fl}}) \vee \Psi'}$$

**Fig. 14.** Selected typing rules for Core SCIF

Rule Let defines sequential composition: a mostly standard let-binding that also enforces information flow and reentrancy security with greater precision than in SeRIF [18]. The second expression $e_2$ executes with integrity $pc'$, where $pc'$ is no more trusted than the initial $pc$ label. However, it may be less trusted to enforce reentrancy security. Recall (Section 3.3) that a method cannot safely perform trusted operations after calling an untrusted method. Therefore, if $pc'$ is trusted to perform an operation that is also protected by the reentrancy input lock $\ell_{\text{L}}$, $e_1$ must maintain that lock. Formally, for any integrity level $\ell$, if $pc' \Rightarrow \ell$ and $\ell_{\text{L}} \Rightarrow \ell$, then $\Psi_1[\underline{\text{n}}].L \Rightarrow \ell$. The condition $\mathcal{T} \vdash \Psi_1[\underline{\text{n}}].L \Rightarrow \ell_{\text{L}} \vee pc'$ precisely enforces this restriction. By modifying the $pc$ integrity in this way, we release any locks that $e_1$ does not maintain. It is therefore safe to type-check $e_2$ with only the locks both present before and maintained by $e_1$: $\Psi_1[\underline{\text{n}}].L \vee \ell_{\text{L}}$.

Rule IfTrust describes dynamic flows-to checks. To support differing trust management schemes and to make this check practical, SCIF restricts to only checking relationships between primitive principals: contract addresses. We also include $v_1 \Rightarrow v_2$ in $\mathcal{T}$ when typing $e_1$, since we know that flow holds in that context. Conservatively assuming flows do not hold unless proven to, we do not need to track that $v_1 \nRightarrow v_2$ when checking $e_2$.

Rule Call has the most premises, but most are standard. Notably, it checks the static reentrancy locks ($\mathcal{T} \vdash pc_1 \Rightarrow pc_2 \vee \ell_{\text{L}}$). To ensure that a contract we do not trust cannot hurt us, Call only trusts the labels specified by the method type as much as it trusts the claim that $v$ actually has type $C$, which is captured by $\ell$. Call therefore attenuates trust in the return value, the output lock label, and the label of each return path by $\ell$. Finally, the integrity of each termination path comes

directly from the method type except the failure path, which is not explicitly tracked. Since a failure occurs precisely when there are neither exceptions nor normal termination, the integrity of the failure path is just the join of the integrities of the other termination paths.

The final rules concern throwing and catching exceptions, and all make use of the termination path labels. Throw indicates that only exceptional termination (with the correct exception type) is possible. TryCatch uses the labels from $\Psi[ex]$ and removes $ex$ from the possible termination paths in $\Psi$ before joining it with $\Psi'$, as such an exception has now been handled. AtomicRescue behavies nearly identically to TryCatch, but using the single distinguished failure path $\underline{\text{fl}}$. It also requires that the body can only terminate normally or with a failure, not with uncaught exceptions.

### 4.3 CDA Safety

Recall from Section 2.3 that a confused deputy attack occurs when an attacker tricks a trusted deputy into performing a security-critical operation with the deputy's full authority when only the attacker's authority is appropriate. To formalize this definition, we note that CDA attacks can only occur at the point of interaction between a deputy and a potential victim. In SCIF, all contract interactions occur through method calls, so we need only consider call boundaries.

At each call, the proper authority to pass to the callee is the integrity of the calling environment, which is tracked by the $pc$ label $pc_{\text{env}}$. The method signature specifies the integrity required to invoke the method in its external $pc$ label $pc_{\text{ex}}$. E-Call only checks that the calling *contract* $\alpha$ has $pc_{\text{ex}}$ integrity, as that is the information directly available to a real smart contract. A CDA occurs if a call occurs—meaning $\alpha \Rightarrow pc_{\text{ex}}$—but $pc_{\text{env}} \not\Rightarrow pc_{\text{ex}}$.

More formally, we parameterize our CDA definition on an integrity level $\ell$. An $\ell$-CDA occurs when an environment that $\ell$ does not trust successfully calls a method requiring at least $\ell$ integrity.

**Definition 1** ($\ell$-CDA Event)**.** A method call is an *$\ell$-CDA event* if $pc_{\text{env}} \not\Rightarrow \ell$ and $pc_{\text{ex}} \Rightarrow \ell$.

*4.3.1 Modeling CDA and Enforcing CDA Safety.* The Call typing rule ensures that SCIF code is free from $\ell$-CDA events at all labels $\ell$ when every contract type-checks. Unfortunately, smart contract environments are open systems with no guarantee that attacker-provided code is well-typed.

Our core calculus instead empowers the attacker to provide contracts of the wrong type as arguments. We still require all code to be well-typed, but add a special atk-cast term to the language, which attackers may use freely.

$$v ::= \cdots \mid \text{atk-cast } v \text{ as } C$$

The type system handles atk-cast like a regular cast, but with far less semantic validation. The lack of validation adds power that is seemingly narrow, but is significant. By passing a contract of type $C$ with a high-integrity method $m$ to a method of a trusted deputy expecting an argument of type $D$ with a *low*-integrity method $m$, an attacker can induce a CDA.

This danger is somewhat curious from the information-flow standpoint. Normally, if an attacker passes a high-integrity value to a method expecting a low-integrity argument, that is no concern; reducing the integrity of data is safe. Method types, however, are different because their $pc$ labels are contravariant. That is, it is safe to use a method requiring *lower* integrity than what is statically expected, but not one requiring *higher* integrity. A CDA occurs precisely when a method expecting higher integrity is used in place of one expecting lower integrity. Our key insight is that CDAs arise because of an interaction between type confusion and contravariance.

To prevent CDAs in the presence of malicious type casts, SCIF adds a simple run-time check: the type of the method being called must be the type expected. The E-AtkCall rule below captures

this check by requiring $mtype(D, m) = mtype(C, m)$, and is otherwise identical to E-Call.

$$[\text{E-AtkCall}]\ \frac{O(\alpha) = C(\overline{v}) \quad mbody(C, m) = (\overline{x}, pc_1 \gg pc_2, e, \overline{ex}) \quad mtype(D, m) = mtype(C, m) \quad \mathcal{M} = \mathcal{M}', \ell'_m \quad \ell'_m \Rightarrow pc_1 \quad \bigwedge_{\ell \in L}(pc_1 \Rightarrow pc_2 \vee \ell) \quad e' = e[\overline{x} \mapsto \overline{w}, \text{this} \mapsto \alpha]}{\langle (\text{atk-cast } \alpha \text{ as } D).m(\overline{w}) \mid C \rangle \longrightarrow \langle \text{return}_{\overline{ex}}(e' \text{ at-pc } pc_2) \mid C[\mathcal{M}, \alpha/\mathcal{M}] \rangle}$$

By eliminating type confusion on method calls, SCIF ensures that $pc_{\text{env}} \Rightarrow pc_{\text{ex}}$ for every call, and therefore eliminates all $\ell$-CDA events. Notably, a more permissive rule requiring only a subtyping relationship between $D.m$ and $C.m$ with proper contravariance on the $pc$ would be sound. However, it would be far more difficult to implement, so we opt for the simpler requirement.

## 5 IMPLEMENTATION

The SCIF compiler consists of just over 12,000 lines of Java code. It uses JFlex [58] and CUP [53] for parsing and does type checking by generating type constraints that it passes to the SHErrLoc constraint solver [99]. The compiler outputs Solidity code in which labels have been erased and run-time mechanisms have been inserted.

Full SCIF supports more types than described in the core language: integer, byte, and more complex types such as arrays, mappings, and structs. Notably, it allows *dependent maps*, which are mappings from contract addresses to values, where the label of each value depends on the address that maps to it. This feature is useful for fine-grained information flow policies in multi-user contracts like ERC-20 tokens.

### 5.1 Run-time Mechanisms

The SCIF compiler adds run-time checks to enforce security that it cannot guarantee statically. If these checks detect a potential security threat, they immediately revert the operation and throw a failure (see Section 3.4).

*5.1.1 CDA Prevention.* As described in Section 3.2, CDA prevention leverages Solidity's method dispatch mechanism, which selects a public method by interpreting the first 4 bytes of data passed to the contract as the (truncated) hash of the method signature. Our compiler integrates information flow labels into the method names of the generated Solidity code, so type confusion results in a mismatch in dispatch hashes between the caller and the callee. Dispatch fails, preventing an attack without adding run-time overhead.

*5.1.2 Dynamic Locks and Trust Management.* The SCIF compiler adds calls to methods in the `Contract` interface (Figure 10) for run-time security enforcement, consisting of the following:

(1) At the beginning of public methods, the compiler inserts a call `trusts(`$pc_{\text{ex}}$`, sender)` to confirm the trustworthiness of the caller; if there may be an auto-endorsement ($pc_{\text{ex}} \nRightarrow pc_{\text{in}}$), the compiler additionally inserts `trusts(`$pc_{\text{in}}$`,` $pc_{\text{ex}}$`) || bypassLocks(`$pc_{\text{ex}}$`)`, ensuring that if any auto-endorsement occurs, no current dynamic reentrancy locks block the caller.

(2) For each `lock(l) {...}` block, the compiler places `acquireLock(l)` immediately before the contents and `releaseLock(l)` immediately after.

(3) Explicit trust relationship queries `l1 => l2` between contract instances are translated to `trusts(l2, l1)`.

*5.1.3 Default Contract Implementations.* SCIF offers default implementations for interfaces described in Section 3.6:

(1) `ContractImpl` adopts a simple trust model. The contract trusts only itself and uses a boolean to monitor lock status. This approach is similar to how Solidity contracts use `ReentrancyGuard` [69]

to prevent reentrancy attacks. But unlike `ReentrancyGuard`, which requires manual marking of methods as `nonReentrant`, SCIF automatically generates dynamic lock checks only when necessary: that is, when an auto-endorsement is possible.

(2) `ManagedContractImpl` still maintains a single lock like `ContractImpl`, but also manages a set of trusted contract addresses. Trustees can freely call methods in the contract without reentrancy concerns and are allowed to dynamically manage the contract's trustees.

(3) `ExternallyManagedContractImpl` defers to two trusted contracts, `TrustManager` and `LockManager`, for trust and lock management. It manages a local set of direct trustees but consults the trust manager for indirect trust queries, and consults the lock manager for lock queries. This enables more sophisticated algorithms for managing trust and reentrancy locks across multiple contracts, while still preventing reentrancy attacks, especially those spanning multiple contracts.

### 5.2 Exception Handling

Solidity's exception handling is limited to a single external call and only reverts when there is a failure [87]. As a result, it cannot be directly used to implement SCIF's exceptions, which do not roll back. Instead, SCIF embeds non-failure termination results in the return value of methods. A SCIF method that may throw an exception returns two values when compiled to Solidity: an integer indicating termination status and a byte array containing the return value (for normal termination) or arguments of the exception.

When compiling `try-catch` in SCIF, the compiler maintains the exception's identifier and arguments thrown within the `try` block. It executes the `catch` block matching the thrown exception's identifier, if any.

The `atomic-rescue` construct supports failures that revert state changes made inside the `atomic` block. Unlike Solidity, where failures undo state changes in the current transaction, SCIF localizes rollback to the `atomic` block, by generating a Solidity trampoline method for the `atomic` block and invoking it as an external call. An optimization left to future work would be avoid a trampoline when the `atomic` block holds a trusted single external function call that throws no checked exceptions.

### 5.3 Error Localization

SCIF solves type constraints entirely using SHErrLoc, a constraint solver shown to substantially improve error localization for IFC [99]. For programs that do not type-check, SCIF reports the most likely error locations as identified by SHErrLoc.

As an example, the Uniswap implementation in Figure 7, but without the dynamic lock—making it ill-typed—generates the following error report.

```
1. Uniswap.scif, line 14: call to this method violates the reentrancy lock.
        assert tY.transfer(this, buyer, yBought);
                           ^
2. Uniswap.scif, line 6: lock label of this method is not maintained.
    uint sellXForY(final address buyer, uint xSold) {
                   ^
```

The most likely error is the call that makes the reentrancy attack possible. SCIF suggests modifying it to respect a reentrancy lock, which fixes the vulnerability. The second suggestion is to change the lock label on the method signature. That change would not fix the error but if had been no second trusted external call, it would have, making it a reasonable guess.

### 5.4 Limitations

Our current SCIF implementation has some limitations:

| Application | LoC | Compilation time (s) | Explicit endorses | Necessary annotations | Bytecode size (bytes) | Solidity bytecode size (bytes) | RSE calls |
|---|---|---|---|---|---|---|---|
| ERC-20 1 | 102 | 1.6 | 13 | 17 | 3845 | 2097 | 26 |
| ERC-20 2 | 88 | 0.55 | 9 | 19 | 3046 | | 20 |
| Uniswap 1 | 270 | 113 | 47 | 57 | 16939 | 10317 | 74 |
| Uniswap 2 | 280 | 106 | 47 | 57 | 16897 | | 86 |
| Dexible swap | 29 | 0.13 | 0 | 2 | 3728 | * | 0 |
| KoET | 164 | 3.61 | 2 | 6 | 3618 | 2400 | 10 |
| Poly Network | 115 | 2.55 | 6 | 8 | 6127 | * | 8 |
| HODLWallet | 73 | 0.33 | 8 | 13 | 1667 | 2137 | 10 |
| SysEscrow | 138 | 0.62 | 3 | 6 | 2370 | 2579 | 13 |

**Table 1.** SCIF case studies.

(1) Address variables must be `final` to be used as information flow principals.

(2) Because of how CDA prevention is implemented, the current implementation does not support a convenient way to interact with contracts that do not implement SCIF.

## 6 EVALUATION

Evaluating a new programming language is not easy. The most interesting question is how effectively (and cost-effectively) SCIF prevents subtle security bugs. There is no appropriate corpus of benchmark contracts to compare against. Therefore, we evaluated the expressiveness and effectiveness of SCIF by implementing and analyzing several challenging real-world smart contracts.

Where feasible, we modified the original Solidity code as little as possible. Table 1 summarizes the results of these case studies. Tests were run on a Macbook Pro 14 with an Apple M1 Max CPU and 64 GB RAM. Some results reflect an immature compiler prototype, and focused engineering effort would likely improve them substantially.

Compilation times range from under a second to nearly 2 minutes. Most time is taken by the SHErrLoc constraint solver, which is slowed by its support for accurate error localization. Added programmer effort is quantified through the number of explicit endorsements and necessary information-flow annotations (1–20% of the lines). As our example contracts are especially dense in security issues, this likely provides an upper bound for the typical annotation burden.

We quantified the overhead of SCIF's run-time security mechanisms through a bytecode size comparison with Solidity and a count of calls for run-time security enforcement in the compiled code ("RSE calls"). While these metrics do not directly represent the run-time overhead, they provide insights for understanding the complexity introduced by the compiler's handling of SCIF's run-time security mechanisms. The Solidity bytecode size for Dexible `swap` and Poly Network are absent because we did not attempt to implement their complex functionality unrelated to the core vulnerabilities, making a fair comparison is not possible. Our bytecode is shorter on HODL Wallet and SysEscrow, because they require Solidity compiler versions that are far lower than what our compiler uses.

The full SCIF code for all contracts listed in Table 1 is available in the supplementary material.

*Case Study* (ERC-20). Our ERC-20 implementation follows that of OpenZeppelin ERC-20 [70]. Table 1 includes two SCIF versions: ERC-20 1 minimizes annotations, while ERC-20 2 leverages SCIF's dependent maps to maintain fine-grained allowance policies while avoiding trust endorsements.

| Operation | ERC-20 Solidity | ERC-20 1 | ERC-20 2 |
|---|---|---|---|
| `approve` | 1440 | 1574 (+9%) | 1389 (-3%) |
| `transfer` | 2292 | 3548 (+55%) | 3371 (+47%) |
| `transferFrom` | 3147 | 4254 (+35%) | 4439 (+41%) |

**Table 2.** Gas (wei) consumed by operations in ERC-20 implementations.

Table 2 shows the run-time overhead of each implementation compared to Solidity, measured by averaging the gas consumption over 1,000 executions of each operation. The overhead for `transfer` and `transferFrom` mostly stems from SCIF's exception-handling mechanism.

The finer-grained ERC-20 2 has lower overheads except in `transferFrom` because its use of dependent map allows it to avoid auto-endorsements with costly dynamic security checks when operating on allowances. The `transferFrom` implementation, however, causes two auto-endorsements, resulting in more dynamic checks.

*Case Study* (Uniswap). We implemented secure variants of Uniswap V1 [92]. The contract was designed to serve as an exchange and token manager for ERC-20 tokens. However, an interaction with ERC-777 tokens [36]—an extension of ERC-20 providing callbacks—exposed a reentrancy vulnerability leading to a high-profile attack [72].

We again developed two versions. Uniswap 1 interacts only with ERC-20 tokens (with no callbacks), while Uniswap 2 engages with both ERC-20 and ERC-777 tokens. Both versions employ explicit dynamic locks during Ether transactions, but Uniswap 2 also requires dynamic locks in token transfers.

We compared gas consumption of the original Uniswap V1 with our SCIF implementations, using Solidity and SCIF ERC-20 implementations. Focusing on method `tokenToExchangeSwapInput`, which corresponds to `sellXForY` depicted in Figure 3, the original Uniswap V1 had an average gas consumption over 100 executions of 46,809 wei per operation. After manually applying the optimization discussed in Section 5.2, our SCIF implementation averaged 52,581 wei per operation—a 12% overhead, of which 5/6 is due to run-time enforcement of auto-endorsement and explicit reentrancy locks, and 1/6 is due to exception handling.

*Case Study* (Dexible Swap). For Dexible, we implemented only the core swap functionality, making direct comparison to the original implementation [42] infeasible. With no auto-endorsements, this implementation operates on the user's behalf and needs no dynamic checks, making it both efficient and obviously secure. SCIF's mechanisms for preventing type confusion prevent the original CDA.

*Case Study* (KoET). Our implementation closely replicates the original KoET contract [41], but is even simpler. KoET included explicit dynamic checks to ensure that only the contract owner could invoke security-critical methods. SCIF automatically enforces this access control based on method labels. Moreover, SCIF's exception mechanism prevents the KoET attack based on incorrect error handling. Interestingly, SCIF's reentrancy protections detected and prevented a previously unreported reentrancy vulnerability stemming from calling `send` before updating the local state.

*Case Study* (Poly Network). Poly Network, a blockchain interoperability application, facilitates the aggregation and response to operations across distinct blockchains. The contract executed user-specified callbacks based on signed events for other blockchains. Inadequate security validation allowed attackers in 2021 to exploit a CDA vulnerability, using Poly Network's `EthCrossChainManager` contract as a confused deputy to access another core component of the application, which then incorrectly transferred $610 million in tokens to the attacker [21].

Our SCIF adaptation closely mirrors the original `EthCrossChainManager` contract [75], but delegates verification of cross-chain operations to an unimplemented third-party contract. We defined the callback method's interface to accurately reflect user integrity levels, which combines with SCIF's dynamic type confusion checks to prevent the CDA attack.

*Case Study* (HODL Wallet). The HODL Wallet [79] was similar to an ERC-20 token wallet but only transferred tokens away from a given address up to 3 times before locking them. Balances were properly updated before executing a transfer, but the counter used to enforce transfer limits was updated only later. This sequencing flaw allowed an attacker to use reentrancy to execute more than 3 transfers from a single address. The SCIF compiler successfully identified the bug and allowed eliminating the vulnerability by moving the counter update earlier.

*Case Study* (SysEscrow). The SysEscrow [79] platform let users create, approve, release, and cancel trade orders. During the cancellation or release of an order, the seller or buyer could exploit reentrancy to illicitly claim the order's currency value. SCIF identified this vulnerability and suggested a dynamic lock, which prevented unauthorized reentrancy during currency transactions.

In summary, SCIF proves effective across a variety of contracts afflicted by subtle security bugs.

## 7 RELATED WORK

*Confused Deputy Attacks.* Rajani et al. [78] give a formal definition of CDA-freedom as a security property and prove that information flow security is sufficient to guarantee CDA-freedom. Jagadeesan et al. [55] use a refinement type system to address cross-site request forgery attacks, a form of CDA that compromises confidentiality. Both, however, assume everything is well-typed and do not address CDAs stemming from type confusion.

Jackal [48] analyzes EVM bytcode for CDAs using symbolic execution, but they do not cover CDAs involving multiple contracts and the tool is incomplete by nature.

*Reentrancy Security.* Our reentrancy security mechanisms improve on those of SeRIF [18], which defines a formal notion of $\ell$-reentrancy and an information flow type system to enforce reentrancy security. SCIF's flexible use of execution paths recognizes more code as secure.

Grossman et al. [49] and Albert et al. [2] propose the notion of Effectively Callback-Free (ECF) executions and develops a static analysis tool that uses SMT solvers to check whether contract operations can be reordered to produce the same result without callbacks. However, this requirement prevents secure interactions between mutually trusting contracts.

*Exception Handling.* Exception mechanisms that trigger transactional rollback have been explored in prior work [16, 57, 63], including Solidity itself. Verse [5] recovers from failed expressions by rolling back to a previously defined state, but it has just one type of statically checked failure and does not distinguish between exceptions and failures. The distinction between expected, statically checked conditions ("contingencies") and unexpected failures ("faults") has been identified as important [80, 102], but not tied to rollback. SCIF combines these two ideas in a novel way that guides programmers to handle foreseeable contingencies, with clean rollback on unexpected failures.

*Dynamic Trust Management.* The SCIF mechanisms for dynamic trust build on a significant body of work on trust management in programming languages [3, 4, 8, 51, 64, 90]. Unlike these prior works, SCIF ties these language-based mechanisms to smart contract identities, expressing IFC policies in the vocabulary of smart contracts.

*Secure Smart Contract Languages.* The Scilla [83] language forces a programming style that separates pure computation, state changes, and method calls. Obsidian [22] and Flint [82] use resource types and typestate to facilitate the reasoning of contract behaviors. The resource types

guarantee that assets, such as tokens, cannot be arbitrarily created or destroyed. Nomos [38] introduces resource-aware session types, which eliminates all single-contract reentrancy. However, none of these languages can guard against CDAs or sophisticated multi-contract reentrancy attacks.

*Smart Contract Security Tools.* Many stand-alone tools aim to find vulnerabilities in smart contracts. AI-based tools [1, 7, 84] provide no soundness or completeness guarantees. Bytecode modification tools that insert dynamic checks to prevent undesirable behaviors [67, 101] lack the high-level typing information SCIF uses, resulting in less precision and eliminating more safe behaviors.

Many tools statically analyze source code, bytecode or disassembled bytecode [12, 14, 34, 46, 52, 62, 89, 91], sometimes using symbolic execution or model checking [39, 43, 44, 50, 56, 65, 68, 88]. Some tools provide soundness and completeness guarantees for specific classes of vulnerabilities, but none handle CDAs and few handle reentrancy. Some tools can check properties described in formal logic, but have scalability and compositionality issues.

Existing formal verification frameworks for smart contracts [6, 47] provide high assurance but require significant user expertise and verification effort.

*Information Flow Control.* Prior work uses IFC to secure decentralized systems. Fabric [64] provides a language to build distributed systems where nodes can securely share code and data despite mutual distrust. DStar [98] tracks information flows within the operating system to secure distributed executions. These previous systems are focused on more traditional settings, rather than smart contracts, and fail to provide reentrancy security.

## 8 CONCLUSION

SCIF is the first practical smart contract programming language to provide compositional security against broad classes of vulnerabilities including reentrancy, confused deputy attacks, and improper error handling. We offer a more general, principled integrity-ba/sed definition of CDAs, which SCIF prevents even in the presence of ill-typed code. SCIF additionally improves on previous reentrancy security protections through more precise tracking of control-flow integrity. The distinction between exceptions and failures facilitates explicit reasoning of those previously implicit execution paths. Replacing ad-hoc authorization mechanisms with systematically managed dynamic trust relationships reduces security risk and development cost. The application of SCIF to a wide variety of real-world examples shows not only its effectiveness for improving security but also its success in harmoniously integrating multiple novel language features.

## 9 ACKNOWLEDGMENTS

We thank Silei Ren and Yulun Yao for their feedback on the paper. We acknowledge the support of Ripple, Inc. and of the National Science Foundation under grant 1704615.

## REFERENCES

[1] Tamer Abdelaziz and Aquinas Hobor. 2023. Smart Learning to Find Dumb Contracts. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1775–1792.

[2] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2020. Taming Callbacks for Smart Contract Modularity. *Proc. ACM on Programming Languages* 4, OOPSLA (Nov. 2020). https://doi.org/10.1145/3428277

[3] Owen Arden, Jed Liu, and Andrew C. Myers. 2015. Flow-Limited Authorization. In *28th IEEE Computer Security Foundations Symp. (CSF)*. 569–583. https://doi.org/10.1109/CSF.2015.42

[4] Owen Arden and Andrew C. Myers. 2016. A Calculus for Flow-Limited Authorization. In *29th IEEE Computer Security Foundations Symp. (CSF)*. 135–147.

[5] Lennart Augustsson, Joachim Breitner, Koen Claessen, Ranjit Jhala, Simon Peyton Jones, Olin Shivers, Guy L. Steele Jr., and Tim Sweeney. 2023. The Verse Calculus: A Core Calculus for Deterministic Functional Logic Programming. *Proc. ACM Program. Lang.* 7, ICFP, Article 203 (aug 2023), 31 pages. https://doi.org/10.1145/3607845

[6] Kushal Babel, Philip Daian, Mahimna Kelkar, and Ari Juels. 2023. Clockwork Finance: Automated Analysis of Economic Security in Smart Contracts. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2499–2516. https://doi.org/10.1109/SP46215.2023.10179346

[7] Kushal Babel, Mojan Javaheripi, Yan Ji, Mahimna Kelkar, Farinaz Koushanfar, and Ari Juels. 2023. Lanturn: Measuring Economic Security of Smart Contracts Through Adaptive Learning. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1212–1226. https://doi.org/10.1145/3576915.3623204

[8] Sruthi Bandhakavi, William Winsborough, and Marianne Winslett. 2008. A Trust Management Approach for Flexible Policy Management in Security-Typed Languages. In *Computer Security Foundations Symposium, 2008*. 33–47.

[9] K. J. Biba. 1977. *Integrity Considerations for Secure Computer Systems*. Technical Report ESD-TR-76-372. USAF Electronic Systems Division, Bedford, MA. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[10] Blockworks. 2024. Helpful hackers net more than $640k in 1 year with crypto bug bounties. https://blockworks.co/news/crypto-hackers-bug-bounties. Accessed March 2024.

[11] Blockworks. 2024. Security review competition will offer a bounty of $1.2M. https://blockworks.co/news/security-review-competition-bounty-reward. Accessed March 2024.

[12] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 161–178. https://doi.org/10.1109/SP46214.2022.9833721

[13] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. 2017. An In-Depth Look at the Parity Multisig Bug. https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/. Accessed March 2021.

[14] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities. In *41$^{st}$ ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 454–469. https://doi.org/10.1145/3385412.3385990

[15] Bruno Cabral and Paulo Marques. 2007. Hidden truth behind .NET's exception handling today. *IET Software* 1, 6 (2007).

[16] Bruno Cabral and Paulo Marques. 2011. A Transactional Model for Automatic Exception Handling. *Computer Languages, Systems & Structures* 37, 1 (April 2011), 43–61. https://doi.org/10.1016/j.cl.2010.09.002

[17] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C. Myers. 2020. Securing Smart Contracts with Information Flow. In *3$^{rd}$ Int'l Symp. on Foundations and Applications of Blockchain (FAB)*.

[18] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C. Myers. 2021. Compositional Security for Reentrant Applications. In *IEEE Symp. on Security and Privacy*. https://doi.org/10.1109/SP40001.2021.00084

[19] Chainlink. 2024. How To Audit a Smart Contract. https://chain.link/education-hub/how-to-audit-smart-contract. Accessed March 2024.

[20] Stephen Chong, K. Vikram, and Andrew C. Myers. 2007. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *16$^{th}$ USENIX Security Symp.*

[21] CNBC. 2021. Suspected hacker behind $600 million Poly Network crypto heist did it 'for fun'. https://www.cnbc.com/2021/08/12/poly-network-hacker-behind-600-million-crypto-heist-did-it-for-fun.html. Accessed March 2024.

[22] Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. 2020. Obsidian: Typestate and Assets for Safer Blockchain Programming. *ACM Trans. on Programming Languages and Systems* 42, 3, Article 14 (Nov. 2020). https://doi.org/10.1145/3417516

[23] CoinDesk. 2021. BitMart CEO Says Stolen Private Key Behind $196M Hack. https://www.coindesk.com/tech/2021/12/06/bitmart-ceo-says-stolen-private-key-behind-196m-hack/. Accessed September 2023.

[24] CoinDesk. 2021. Cross-Chain DeFi Site Poly Network Hacked; Hundreds of Millions Potentially Lost. https://www.coindesk.com/markets/2021/08/10/cross-chain-defi-site-poly-network-hacked-hundreds-of-millions-potentially-lost. Accessed August 2023.

[25] CoinDesk. 2022. Axie Infinity's Ronin Network Suffers $625M Exploit. https://www.coindesk.com/tech/2022/03/29/axie-infinitys-ronin-network-suffers-625m-exploit/. Accessed September 2023.

[26] CoinDesk. 2022. DeFi Lender Rari Capital/Fei Loses $80M in Hack. https://www.coindesk.com/business/2022/04/30/defi-lender-rari-capitalfei-loses-80m-in-hack. Accessed August 2023.

[27] CoinDesk. 2023. DeFi Protocol Conic Finance Hacked for 1,700 Ether. https://www.coindesk.com/tech/2023/07/21/defi-protocol-conic-finance-hacked-for-1700-ether/. Accessed September 2023.

[28] Cointelegraph. 2023. Dexible aggregator hacked for $2M via 'selfSwap' function. https://cointelegraph.com/news/dexibleapp-aggregator-hacked-for-2m-via-selfswap-function. Accessed August 2023.

[29] Cointelegraph. 2023. Era Lend on zkSync exploited for $3.4M in reentrancy attack. https://cointelegraph.com/news/era-lend-zksync-exploited-reentrancy-attack. Accessed August 2023.

[30] Cointelegraph. 2023. Poly Network urges users to withdraw after exploit affects 57 crypto assets. https://cointelegraph.com/news/poly-network-users-withdraw-bridge-exploit-affects-57-crypto. Accessed September 2023.

[31] Consensys. 2022. Ethereum Smart Contract Best Practices. https://consensys.github.io/smart-contract-best-practices/development-recommendations/general/external-calls/#handle-errors-in-external-calls. Accessed November 2023.

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

[33] CryptoPotato. 2023. DeFi Protocol dForce Loses $3.6M in Reentrancy Attack. https://cryptopotato.com/defi-protocol-dforce-loses-3-6m-in-reentrancy-attack/. Accessed September 2023.

[34] Siwei Cui, Gang Zhao, Yifei Gao, Tien Tavu, and Jeff Huang. 2022. VRust: Automated Vulnerability Detection for Solana Smart Contracts. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. Association for Computing Machinery, New York, NY, USA, 639–652. https://doi.org/10.1145/3548606.3560552

[35] CWE-1265 2018. CWE-1265: Unintended Reentrant Invocation of Non-reentrant Code Via Nested Calls. https://cwe.mitre.org/data/definitions/1265.html. Accessed March 2021.

[36] Jacques Dafflon, Jordi Baylina, and Thomas Shababi. 2017. ERC-777: Token Standard. https://eips.ethereum.org/EIPS/eip-777. Accessed December 2023.

[37] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2020. Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. In *IEEE Symp. on Security and Privacy*. 910–927. https://doi.org/10.1109/SP40000.2020.00040

[38] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. 2019. Resource-aware session types for digital contracts. In *34th IEEE Computer Security Foundations Symp. (CSF)*. IEEE.

[39] Yue Duan, Xin Zhao, Yu Pan, Shucheng Li, Minghao Li, Fengyuan Xu, and Mu Zhang. 2022. Towards Automated Safety Vetting of Smart Contracts in Decentralized Applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. Association for Computing Machinery, New York, NY, USA, 921–935. https://doi.org/10.1145/3548606.3559384

[40] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. 2005. Labels and Event Processes in the Asbestos Operating System. In *20th ACM Symp. on Operating System Principles (SOSP)* (Brighton, UK).

[41] Kieran Elby. 2016. King of the Ether Throne v0.4.0. https://github.com/kieranelby/KingOfTheEtherThrone/blob/v0.4.0/contracts/KingOfTheEtherThrone.sol. Accessed December 2023.

[42] Etherscan. 2023. Dexible on-chain contract. https://etherscan.io/address/0x33e690aea97e4ef25f0d140f1bf044d663091daf#code. Accessed December 2023.

[43] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. EthBMC: A Bounded Model Checker for Smart Contracts. In *29th USENIX Security Symp.*

[44] Puneet Gill, Indrani Ray, Alireza Lotfi Takami, and Mahesh Tripunitara. 2023. Finding Unchecked Low-Level Calls with Zero False Positives and Negatives in Ethereum Smart Contracts. In *Foundations and Practice of Security (Lecture Notes in Computer Science)*, Guy-Vincent Jourdan, Laurent Mounier, Carlisle Adams, Florence Sèdes, and Joaquin Garcia-Alfaro (Eds.). Springer Nature Switzerland, Cham, 305–321. https://doi.org/10.1007/978-3-031-30122-3_19

[45] Joseph A. Goguen and Jose Meseguer. 1982. Security Policies and Security Models. In *IEEE Symp. on Security and Privacy*. 11–20. https://doi.org/10.1109/SP.1982.10014

[46] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving out-of-Gas Conditions in Ethereum Smart Contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 116:1–116:27. https://doi.org/10.1145/3276486

[47] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. Foundations and Tools for the Static Analysis of Ethereum Smart Contracts. In *International Conference on Computer Aided Verification (CAV)*. Springer, 51–78.

[48] Fabio Gritti, Nicola Ruaro, Robert McLaughlin, Priyanka Bose, Dipanjan Das, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. 2023. Confusum Contractum: Confused Deputy Vulnerabilities in Ethereum Smart Contracts. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1793–1810.

[49] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM on Programming Languages* 2, POPL, Article 48 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158136

[50] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. EOSAFE: Security Analysis of EOSIO Smart Contracts. In *30th USENIX Security Symposium (USENIX Security 21)*. 1271–1288.

[51] Andrew K. Hirsch, Pedro H. Azevedo Amorim, Ethan Cecchetti, Ross Tate, and Owen Arden. 2020. First-Order Logic for Flow-Limited Authorization. In *IEEE Computer Security Foundations Symp. (CSF)*. 123–138.

[52] Sebastian Holler, Sebastian Biewer, and Clara Schneidewind. 2023. HoRStify: Sound Security Analysis of Smart Contracts. In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 245–260. https://doi.org/10.1109/CSF57540.2023.00023

[53] Scott Hudson, Frank Flannery, C. Scott Ananian, and Michael Petter. 2014. CUP 0.11b: Construction of Useful Parsers. (June 2014). Software release, http://www2.cs.tum.edu/projects/cup.

[54] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. on Programming Languages and Systems* 23, 3 (2001), 396–450.

[55] Radha Jagadeesan, Corin Pitcher, and James Riely. 2012. Succour to the Confused Deputy. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Ranjit Jhala and Atsushi Igarashi (Eds.). Springer, Berlin, Heidelberg, 66–81. https://doi.org/10.1007/978-3-642-35182-2_6

[56] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *Network and Distributed System Security Symposium*.

[57] J. Kienzle, A. Romanovsky, and A. Strohmeier. 2001. Open Multithreaded Transactions: Keeping Threads and Exceptions under Control. In *Proceedings Sixth International Workshop on Object-Oriented Real-Time Dependable Systems*. 197–205. https://doi.org/10.1109/WORDS.2001.945131

[58] Gerwin Klein, Steve Rowe, and Regis Decamp. 2020. JFlex 1.8.2. (May 2020). Software release, https://jflex.de.

[59] KoET 2016. Post-Mortem Investigation (Feb 2016). https://www.kingoftheether.com/postmortem.html.

[60] KoET 2017. King of the Ether. https://www.kingoftheether.com/. Accessed Mar 2024.

[61] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information Flow Control for Standard OS Abstractions. In *21st ACM Symp. on Operating System Principles (SOSP)*.

[62] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symp.*

[63] Barbara Liskov. 1988. Distributed Programming in Argus. *Commun. ACM* 31, 3 (March 1988), 300–312. https://doi.org/10.1145/42392.42399

[64] Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. 2017. Fabric: Building Open Distributed Systems Securely by Construction. *J. Computer Security* 25, 4–5 (May 2017), 319–321. https://doi.org/10.3233/JCS-0559

[65] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *ACM Conf. on Computer and Communications Security (CCS)* (Vienna, Austria). 254–269. https://doi.org/10.1145/2976749.2978309

[66] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *26th ACM Symp. on Principles of Programming Languages (POPL)*. 228–241. https://doi.org/10.1145/292540.292561

[67] Tai D. Nguyen, Long H. Pham, and Jun Sun. 2021. SGUARD: Towards Fixing Vulnerable Smart Contracts Automatically. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1215–1229. https://doi.org/10.1109/SP40001.2021.00057

[68] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 653–663. https://doi.org/10.1145/3274694.3274743

[69] OpenZeppelin. 2021. ReentrancyGuard. https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard. Accessed November 2023.

[70] OpenZeppelin. 2023. ERC20 Implementation. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol. Accessed December 2023.

[71] Parity Technologies. 2017. A Postmortem on the Parity Multi-Sig Library Self-Destruct. https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/. Accessed November 2023.

[72] PeckShield. 2020. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09. Accessed November 2023.

[73] Daniel Perez and Benjamin Livshits. 2021. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1325–1341.

[74] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT press.

[75] Poly Network. 2020. The Vulnerable `EthCrossChainManager` contract. https://github.com/polynetwork/eth-contracts/blob/d16252b2b857eecf8e558bd3e1f3bb14cff30e9b/contracts/core/cross_chain_manager/logic/EthCrossChainManager.sol. Accessed March 2024.

[76] Nathaniel Popper. 2016. A Hacking of More Than $50 Million Dashes Hopes in the World of Virtual Currency. *The New York Times* (17 June 2016).

[77] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying Blockchain Extractable Value: How dark is the forest?. In *IEEE Symp. on Security and Privacy*. 198–214. https://doi.org/10.1109/SP46214.2022.9833734

[78] Vineet Rajani, Deepak Garg, and Tamara Rezk. 2016. On Access Control, Capabilities, Their Equivalence, and Confused Deputy Attacks. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. 150–163. https://doi.org/10.1109/CSF.2016.18

[79] Michael Rodler, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan Karame, and Lucas Davi. 2023. EF/ CF: High Performance Smart Contract Fuzzing for Exploit Generation. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 449–471.

[80] Barry Ruzek. 2007. Effective Java Exceptions. https://www.oracle.com/technical-resources/articles/enterprise-architecture/effective-exceptions-part1.html. Accessed December 2023.

[81] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan. 2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

[82] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. 2018. Writing safe smart contracts in Flint. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. 218–219.

[83] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer smart contract programming with Scilla. *Proc. ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 1–30.

[84] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *30th USENIX Security Symposium (USENIX Security 21)*. 1361–1378.

[85] Solidity 2023. Solidity Documentation. Release 0.8.23. https://docs.soliditylang.org/en/v0.8.23/. Accessed November 2023.

[86] Solidity Team. 2023. Security Considerations. https://docs.soliditylang.org/en/v0.8.23/security-considerations.html#use-the-checks-effects-interactions-pattern. Accessed November 2023.

[87] Solidity Team. 2023. Solidity Documentation. Release 0.8.23. https://docs.soliditylang.org/en/v0.8.23/control-structures.html#try-catch. Accessed November 2023.

[88] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. 2021. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*. 555–571. https://doi.org/10.1109/SP40001.2021.00085

[89] Zhiyuan Sun, Xiapu Luo, and Yinqian Zhang. 2023. Panda: Security Analysis of Algorand Smart Contracts. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1811–1828.

[90] Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. 2006. Managing Policy Updates in Security-Typed Languages. In *19th IEEE Computer Security Foundations Workshop (CSFW)*. 202–216.

[91] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3243734.3243780

[92] Uniswap. 2018. Uniswap V1. https://github.com/Uniswap/v1-contracts/blob/master/contracts/uniswap_exchange.vy. Accessed December 2023.

[93] Fabian Vogelsteller and Vitalik Buterin. 2015. ERC-20: Token Standard. https://eips.ethereum.org/EIPS/eip-20. Accessed December 2023.

[94] Zikai Alex Wen and Andrew Miller. 2016. Scanning Live Ethereum Contracts for the 'Unchecked-Send' Bug. https://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/. Accessed December 2023.

[95] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*.

[96] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. 2002. Secure Program Partitioning. *ACM Trans. on Computer Systems* 20, 3 (Aug. 2002), 283–328. https://doi.org/10.1145/566340.566343

[97] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making Information Flow Explicit in HiStar. In *7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*. 263–278.

[98] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. 2008. Securing distributed systems with information flow control. In *5th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*. 293–308.

[99] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton Jones. 2017. SHErrLoc: A Static Holistic Error Locator. *ACM Trans. on Programming Languages and Systems* 39, 4 (Aug. 2017), 18.

[100] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town Crier: An Authenticated Data Feed for Smart Contracts. In *23rd ACM Conf. on Computer and Communications Security (CCS)* (Vienna, Austria). ACM, New York, NY, USA, 270–282. https://doi.org/10.1145/2976749.2978326

[101] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. 2020. SMARTSHIELD: Automatic Smart Contract Protection Made Easy. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 23–34. https://doi.org/10.1109/SANER48275.2020.9054825

[102] Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. 2016. Accepting Blame for Safe Tunneled Exceptions. In *37th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (Santa Barbara, California, USA). 281–295.

$$\text{[E-Eval]}\ \frac{\langle s \mid C\rangle \longrightarrow \langle s' \mid C'\rangle}{\langle E[s] \mid C\rangle \longrightarrow \langle E[s'] \mid C'\rangle} \qquad \text{[E-Let]}\ \frac{}{\langle \text{let } x = v \text{ in } e \mid C\rangle \longrightarrow \langle e[x \mapsto v] \mid C\rangle}$$

$$\text{[E-IfT]}\ \frac{}{\langle \text{if}_{pc} \text{ true then } e_1 \text{ else } e_2 \mid C\rangle \longrightarrow \langle e_1 \text{ at-pc } pc \mid C\rangle} \qquad \text{[E-IfF]}\ \frac{}{\langle \text{if}_{pc} \text{ false then } e_1 \text{ else } e_2 \mid C\rangle \longrightarrow \langle e_2 \text{ at-pc } pc \mid C\rangle}$$

$$\text{[E-IfTrustT]}\ \frac{\alpha_1 \Rrightarrow \alpha_2}{\langle \text{if}_{pc}\ \alpha_1 \Rrightarrow \alpha_2 \text{ then } e_1 \text{ else } e_2 \mid C\rangle \longrightarrow \langle e_1 \text{ at-pc } pc \mid C\rangle}$$

$$\text{[E-IfTrustF]}\ \frac{\alpha_1 \Rrightarrow \alpha_2}{\langle \text{if}_{pc}\ \alpha_1 \Rrightarrow \alpha_2 \text{ then } e_1 \text{ else } e_2 \mid C\rangle \longrightarrow \langle e_2 \text{ at-pc } pc \mid C\rangle} \qquad \text{[E-AtPc]}\ \frac{}{\langle v \text{ at-pc } pc \mid C\rangle \longrightarrow \langle v \mid C\rangle}$$

$$\text{[E-Ref]}\ \frac{\iota \notin \text{dom}(\sigma)}{\langle \text{ref } v\ \tau \mid C\rangle \longrightarrow \langle \iota \mid C[\sigma[\iota \mapsto v]/\sigma]\rangle} \qquad \text{[E-Deref]}\ \frac{}{\langle !\iota \mid C\rangle \longrightarrow \langle \sigma(\iota) \mid C\rangle}$$

$$\text{[E-Assign]}\ \frac{}{\langle \iota := v \mid C\rangle \longrightarrow \langle () \mid C[\sigma[\iota \mapsto v]/\sigma]\rangle} \qquad \text{[E-New]}\ \frac{\alpha \notin \text{dom}(O)}{\langle \text{new } C(\overline{v}) \mid C\rangle \longrightarrow \langle \alpha \mid C[O[\alpha \mapsto C(\overline{v})]/O]\rangle}$$

$$\text{[E-Cast]}\ \frac{O(\alpha) = D(\overline{v}) \qquad D <: C}{\langle (C)\alpha \mid C\rangle \longrightarrow \langle \alpha \mid C\rangle} \qquad \text{[E-Field]}\ \frac{O(\alpha) = C(\overline{v})}{\langle \alpha.f_i \mid C\rangle \longrightarrow \langle v_i \mid C\rangle} \qquad \text{[E-Endorse]}\ \frac{}{\langle \text{endorse } v \text{ from } \ell' \text{ to } \ell \mid C\rangle \longrightarrow \langle v \mid C\rangle}$$

(a) IFC Calculus Small-Step Operational Semantic Rules

## A FULL SCIF RULES

The full operational semantics for SCIF are given in Figure 15 and the full typing rules are given in Figures 16 and 17. We introduce the following syntactic forms as evaluation contexts to enable precise tracking of method boundaries, execution integrity, dynamic locks, and type confusions:

$$\begin{aligned} E \quad &::= \quad [\cdot] \mid \text{let } x = E \text{ in } e \mid \text{try } E \text{ catch } x : ex\ e \mid \text{trans } E \text{ rescue } x\ e \\ &\quad\ \mid \quad \text{return}_{\overline{ex}}\ E \mid E \text{ at-pc } pc \mid E \text{ with-lock } \ell \mid \text{atk-cast } E \text{ as } D \\ s \quad &::= \quad E[e] \end{aligned}$$

To cleanly handle exceptions and transactions, a *throw context* $T$ is an evaluation context through which unhandled exceptions and failures can freely propagate:

$$T ::= [\cdot] \mid \text{let } x = E \text{ in } e \mid E \text{ at-pc } pc \mid \text{atk-cast } E \text{ as } D$$

$$[\text{E-ThrowCtx}]\ \frac{}{\langle T[\text{throw } v] \mid C\rangle \longrightarrow \langle \text{throw } v \mid C\rangle} \qquad [\text{E-FailCtx}]\ \frac{}{\langle T[\text{fail } v] \mid C\rangle \longrightarrow \langle \text{fail } v \mid C\rangle}$$

$$[\text{E-TryCaught}]\ \frac{}{\langle \text{try (throw } ex(\overline{v})) \text{ catch } x\!:\!ex\ e \mid C\rangle \longrightarrow \langle e[x \mapsto ex(\overline{v})] \mid C\rangle}$$

$$[\text{E-TryUncaught}]\ \frac{ex \neq ex'}{\langle \text{try (throw } ex(v)) \text{ catch } x\!:\!ex'\ e \mid C\rangle \longrightarrow \langle \text{throw } ex(v) \mid C\rangle}$$

$$[\text{E-Atomic}]\ \frac{}{\langle \text{atomic } e_1 \text{ rescue } x\ e_2 \mid C\rangle \longrightarrow \langle \text{trans } e_1 \text{ rescue } x\ e_2 \mid C[S, \sigma/S]\rangle}$$

$$[\text{E-AtomicRescued}]\ \frac{S = S', \sigma'}{\langle \text{trans (fail } v) \text{ rescue } x\ e \mid C\rangle \longrightarrow \langle e[x \mapsto v] \mid C[\sigma'/\sigma; S'/S]\rangle}$$

$$[\text{E-TryRet}]\ \frac{}{\langle \text{try } v \text{ catch } x\!:\!ex\ e \mid C\rangle \longrightarrow \langle v \mid C\rangle}$$

$$[\text{E-AtomicCommit}]\ \frac{S = S', \sigma'}{\langle \text{trans } v \text{ rescue } x\ e \mid C\rangle \longrightarrow \langle v \mid C[S/S']\rangle}$$

(b) Small-step operational semantic rules for exception handling.

$$[\text{E-Lock}]\ \frac{}{\langle \text{lock } \ell \text{ in } o \mid C\rangle \longrightarrow \langle o \text{ with-lock } \ell \mid C[L, \ell/L]\rangle}$$

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

$$[\text{E-Call}]\ \frac{\begin{array}{c} O(\alpha) = C(\overline{v}) \qquad mbody(C, m) = \left(\overline{x}, pc_1 \gg pc_2, e, \overline{ex}\right) \\ \mathcal{M} = \mathcal{M}', \ell'_m \qquad \ell'_m \Rightarrow pc_1 \qquad \bigwedge_{\ell \in L} (pc_1 \Rightarrow pc_2 \vee \ell) \qquad e' = e[\overline{x} \mapsto \overline{w}, \text{this} \mapsto \alpha] \end{array}}{\langle \alpha.m(\overline{w}) \mid C\rangle \longrightarrow \langle \text{return}_{\overline{ex}}(e' \text{ at-pc } pc_2) \mid C[\mathcal{M}, \alpha/\mathcal{M}]\rangle}$$

$$[\text{E-AtkCall}]\ \frac{\begin{array}{c} O(\alpha) = C(\overline{v}) \qquad mbody(C, m) = \left(\overline{x}, pc_1 \gg pc_2, e, \overline{ex}\right) \qquad mtype(D, m) = mtype(C, m) \\ \mathcal{M} = \mathcal{M}', \ell'_m \qquad \ell'_m \Rightarrow pc_1 \qquad \bigwedge_{\ell \in L} (pc_1 \Rightarrow pc_2 \vee \ell) \qquad e' = e[\overline{x} \mapsto \overline{w}, \text{this} \mapsto \alpha] \end{array}}{\langle (\text{atk-cast } \alpha \text{ as } D).m(\overline{w}) \mid C\rangle \longrightarrow \langle \text{return}_{\overline{ex}}(e' \text{ at-pc } pc_2) \mid C[\mathcal{M}, \alpha/\mathcal{M}]\rangle}$$

$$[\text{E-CallLowInteg}]\ \frac{\begin{array}{c} O(\alpha) = C(\overline{v}) \qquad mbody(C, m) = \left(\overline{x}, pc_1 \gg pc_2, e, \overline{ex}\right) \\ \mathcal{M} = \mathcal{M}', \ell'_m \qquad \ell'_m \Rightarrow pc_1 \qquad \ell_{\mathcal{A}} \Rightarrow pc_2 \qquad e' = e[\overline{x} \mapsto \overline{w}, \text{this} \mapsto \alpha] \end{array}}{\langle \alpha.m(\overline{w}) \mid C\rangle \longrightarrow \langle \text{return}_{\overline{ex}}(e' \text{ at-pc } pc_2) \mid C[\mathcal{M}, \ell_m/\mathcal{M}]\rangle}$$

$$[\text{E-ReturnV}]\ \frac{\mathcal{M} = \mathcal{M}', \ell_m}{\langle \text{return}_{\overline{ex}}\, v \mid C\rangle \longrightarrow \langle v \mid C[\mathcal{M}'/\mathcal{M}]\rangle}$$

$$[\text{E-ReturnE}]\ \frac{\mathcal{M} = \mathcal{M}', \ell_m}{\langle \text{return}_{\overline{ex}}(\text{throw } ex_i(\overline{v})) \mid C\rangle \longrightarrow \langle \text{throw } ex_i(\overline{v}) \mid C[\mathcal{M}'/\mathcal{M}]\rangle}$$

$$[\text{E-ReturnEF}]\ \frac{\mathcal{M} = \mathcal{M}', \ell_m \qquad ex \notin \overline{ex}}{\langle \text{return}_{\overline{ex}}(\text{throw } ex(\overline{v})) \mid C\rangle \longrightarrow \langle \text{fail } ex(\overline{v}) \mid C[\mathcal{M}'/\mathcal{M}]\rangle}$$

$$[\text{E-ReturnF}]\ \frac{\mathcal{M} = \mathcal{M}', \ell_m}{\langle \text{return}_{\overline{ex}}(\text{fail } v) \mid C\rangle \longrightarrow \langle \text{fail } v \mid C[\mathcal{M}'/\mathcal{M}]\rangle}$$

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

(c) Lock-aware small-step operational semantic rules.

**Fig. 15.** Full small-step operational semantics for SCIF.

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

$$[\textsc{Addr}]\ \frac{\Sigma_C(\alpha) = C}{\Sigma;\Gamma;\mathcal{T} \vdash \alpha : C^{\ell}} \qquad [\textsc{Loc}]\ \frac{\Sigma_R(\iota) = \tau}{\Sigma;\Gamma;\mathcal{T} \vdash \iota : (\mathsf{ref}\ \tau)^{\ell}} \qquad [\textsc{Null}]\ \frac{}{\Sigma;\Gamma;\mathcal{T} \vdash \mathsf{null} : (\mathsf{ref}\ \tau)^{\ell}} \qquad [\textsc{AtkCast}]\ \frac{\Sigma;\Gamma;\mathcal{T} \vdash v : C^{\ell}}{\Sigma;\Gamma;\mathcal{T} \vdash \mathsf{atk\text{-}cast}\ v\ \mathsf{as}\ D : D^{\ell}}$$

$$[\textsc{SubtypeV}]\ \frac{\Sigma;\Gamma;\mathcal{T} \vdash v : \tau' \qquad \mathcal{T} \vdash \tau' <: \tau}{\Sigma;\Gamma;\mathcal{T} \vdash v : \tau}$$

(a) Value typing

$$[\textsc{Val}]\ \frac{\Sigma;\Gamma;\mathcal{T} \vdash v : \tau}{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\mathrm{L}} \vdash v : \tau \dashv \{\underline{\mathsf{n}} \mapsto (pc, \ell'_{\mathrm{L}})\}} \qquad [\textsc{New}]\ \frac{\mathit{fields}(C) = \overline{f} : \overline{\tau} \qquad \Sigma;\Gamma;\mathcal{T} \vdash \overline{v} : \overline{\tau}}{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\mathrm{L}} \vdash \mathsf{new}\ C(\overline{v}) : C^{\ell} \dashv \{\underline{\mathsf{n}} \mapsto (pc, \ell'_{\mathrm{L}})\}}$$

$$[\textsc{Cast}]\ \frac{\Sigma;\Gamma;\mathcal{T} \vdash v : D^{\ell}}{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\mathrm{L}} \vdash (C)v : C^{\ell} \dashv \{\underline{\mathsf{n}} \mapsto (pc, \ell'_{\mathrm{L}})\}} \qquad [\textsc{Field}]\ \frac{\Sigma;\Gamma;\mathcal{T} \vdash v : C^{\ell} \qquad \mathit{fields}(C) = \overline{f} : \overline{\tau} \qquad \mathcal{T} \vdash \tau_i <: \tau \qquad \mathcal{T} \vdash \ell \triangleleft \tau}{\Sigma;\Gamma;\mathcal{T};pc;\lambda_{\mathrm{I}} \vdash v.f_i : \tau \dashv \{\underline{\mathsf{n}} \mapsto (pc, \ell'_{\mathrm{L}})\}}$$

$$[\textsc{Ref}]\ \frac{\Sigma;\Gamma;\mathcal{T} \vdash v : \tau \qquad \mathcal{T} \vdash pc \triangleleft \tau}{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\mathrm{L}} \vdash \mathsf{ref}\ v\ \tau : (\mathsf{ref}\ \tau)^{\ell} \dashv \{\underline{\mathsf{n}} \mapsto (pc, \ell'_{\mathrm{L}})\}} \qquad [\textsc{Deref}]\ \frac{\Sigma;\Gamma;\mathcal{T} \vdash v : (\mathsf{ref}\ \tau')^{\ell} \qquad \mathcal{T} \vdash \tau' <: \tau \qquad \mathcal{T} \vdash \ell \triangleleft \tau}{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\mathrm{L}} \vdash !v : \tau \dashv \{\underline{\mathsf{n}} \mapsto (pc, \ell'_{\mathrm{L}})\}}$$

$$[\textsc{Variance}]\ \frac{\begin{array}{c}\Sigma;\Gamma;\mathcal{T};pc';\ell'_{\mathrm{L}} \vdash e : \tau' \dashv \Psi \\ \mathcal{T} \vdash \tau' <: \tau \qquad \mathcal{T} \vdash pc \Rightarrow pc' \qquad \mathcal{T} \vdash \ell'_{\mathrm{L}} \Rightarrow \ell_{\mathrm{L}} \qquad \mathcal{T} \vdash \Psi[p].pc \Rightarrow pc'' \qquad \mathcal{T} \vdash \Psi[p].L \Rightarrow L''\end{array}}{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\mathrm{L}} \vdash e : \tau \dashv \Psi[p \mapsto (pc'', L'')]}$$

(b) Primitive Expression Typing

[Endorse]

$$\frac{\Sigma; \Gamma; \mathcal{T} \vdash v : t^{\ell'}}{\Sigma; \Gamma; \mathcal{T}; \ell; \ell_{\mathrm{L}} \vdash \mathsf{endorse}\ v\ \mathsf{from}\ \ell'\ \mathsf{to}\ \ell : t^{\ell} \dashv \{\underline{\mathrm{n}} \mapsto (\ell, \ell'_{\mathrm{L}})\}}$$

[Call]

$$\frac{\begin{array}{c} mtype(C, m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; L} \tau_0 : \ell_{\underline{\mathrm{n}}}, \overline{ex^{\ell_e}} \qquad \Sigma; \Gamma; \mathcal{T} \vdash v : C^{\ell} \\ \Sigma; \Gamma; \mathcal{T} \vdash \overline{v_a} : \overline{\tau_a} \qquad \mathcal{T} \vdash \tau_0 <: \tau \qquad \mathcal{T} \vdash pc \vee \ell \Rightarrow pc_1 \qquad \mathcal{T} \vdash pc_1 \Rightarrow pc_2 \vee \ell_{\mathrm{L}} \qquad \mathcal{T} \vdash \ell \triangleleft \tau \\ \ell_{\underline{\mathrm{fl}}} = \ell_{\underline{\mathrm{n}}} \vee \ell \vee \bigvee \overline{\ell_e} \qquad \ell'_{\mathrm{L}} = L \vee \ell \qquad \Psi = \left\{\underline{\mathrm{n}} \mapsto \left(\ell_{\underline{\mathrm{n}}} \vee \ell, \ell'_{\mathrm{L}}\right), \underline{\mathrm{fl}} \mapsto \left(\ell_{\underline{\mathrm{fl}}}, \ell'_{\mathrm{L}}\right), \overline{ex} \mapsto \left(\overline{\ell_e} \vee \ell, \ell'_{\mathrm{L}}\right)\right\} \end{array}}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash v.m(\overline{v_a}) : \tau \dashv \Psi}$$

[If]

$$\frac{\begin{array}{c} \Sigma; \Gamma; \mathcal{T} \vdash v : \mathsf{bool}^{\ell} \qquad \mathcal{T} \vdash \ell \triangleleft \tau \\ \Sigma; \Gamma; \mathcal{T}; pc \vee \ell; \ell_{\mathrm{L}} \vdash e_1 : \tau \dashv \Psi_1 \\ \Sigma; \Gamma; \mathcal{T}; pc \vee \ell; \ell_{\mathrm{L}} \vdash e_2 : \tau \dashv \Psi_2 \end{array}}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash \mathsf{if}_{pc}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau \dashv \Psi_1 \vee \Psi_2}$$

[IfTrust]

$$\frac{\begin{array}{ll} \Sigma; \Gamma; \mathcal{T} \vdash v_1 : C_1^{\ell} & \Sigma; \Gamma; \mathcal{T}, v_1 \Rightarrow v_2; pc \vee \ell; \ell_{\mathrm{L}} \vdash e_1 : \tau \dashv \Psi_1 \qquad \mathcal{T} \vdash \ell \triangleleft \tau \\ \Sigma; \Gamma; \mathcal{T} \vdash v_2 : C_2^{\ell} & \Sigma; \Gamma; \mathcal{T}; pc \vee \ell; \ell_{\mathrm{L}} \vdash e_2 : \tau \dashv \Psi_2 \end{array}}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash \mathsf{if}_{pc}\ (v_1 \Rightarrow v_2)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau \dashv \Psi_1 \vee \Psi_2}$$

[Assign]

$$\frac{\begin{array}{c} \Sigma; \Gamma; \mathcal{T} \vdash v_1 : (\mathsf{ref}\ \tau)^{\ell} \\ \Sigma; \Gamma; \mathcal{T} \vdash v_2 : \tau \qquad \mathcal{T} \vdash \ell \triangleleft \tau \end{array}}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash v_1 := v_2 : \mathsf{unit}^{\ell'} \dashv \{\underline{\mathrm{n}} \mapsto (pc, \ell'_{\mathrm{L}})\}}$$

[Lock]

$$\frac{\begin{array}{c} \Sigma; \Gamma; \mathcal{T}; pc; \ell'_{\mathrm{L}} \vdash e : \tau \dashv \Psi' \qquad \mathcal{T} \vdash \ell'_{\mathrm{L}} \wedge \ell \Rightarrow \ell_{\mathrm{L}} \\ \mathrm{dom}(\Psi) = \mathrm{dom}(\Psi') \qquad (\Psi'[p].pc = \Psi[p].pc)^{p \in \mathrm{dom}(\Psi)} \qquad (\mathcal{T} \vdash \Psi'[p].L \wedge \ell \Rightarrow \Psi[p].L)^{p \in \mathrm{dom}(\Psi)} \end{array}}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash \mathsf{lock}\ \ell\ \mathsf{in}\ e : \tau \dashv \Psi}$$

[Let]

$$\frac{\begin{array}{c} \Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash e_1 : \tau_1 \dashv \Psi_1 \qquad \ell'_{\mathrm{L}} = \Psi_1[\underline{\mathrm{n}}].L \vee \ell_{\mathrm{L}} \qquad \Sigma; \Gamma, x : \tau_1; \mathcal{T}; pc'; \ell'_{\mathrm{L}} \vdash e_2 : \tau_2 \dashv \Psi_2 \\ \mathcal{T} \vdash \Psi_1[\underline{\mathrm{n}}].pc \Rightarrow pc' \qquad \mathcal{T} \vdash \Psi_1[\underline{\mathrm{n}}].L \Rightarrow \ell_{\mathrm{L}} \vee pc' \qquad \mathcal{T} \vdash \Psi_1[\underline{\mathrm{n}}].L \Rightarrow \Psi_2[\underline{\mathrm{n}}].L \end{array}}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \dashv (\Psi_1 \setminus \underline{\mathrm{n}}) \vee \Psi_2}$$

[TryCatch]

$$\frac{\begin{array}{c} \Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash e : \tau \dashv \Psi \qquad \mathcal{T} \vdash \Psi[ex].L \Rightarrow \ell_{\mathrm{L}} \\ pc' = \Psi[ex].pc \qquad \Sigma; \Gamma, x : ex^{pc'}; \mathcal{T}; pc'; \ell_{\mathrm{L}} \vdash e' : \tau \dashv \Psi' \end{array}}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash \mathsf{try}\ e\ \mathsf{catch}\ x : ex\ e' : \tau \dashv (\Psi \setminus ex) \vee \Psi'}$$

[AtomicRescue]

$$\frac{\begin{array}{c} \Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash e : \tau \dashv \Psi \qquad \mathrm{dom}(\Psi) \subseteq \{\underline{\mathrm{n}}, \underline{\mathrm{fl}}\} \\ \mathcal{T} \vdash \Psi[\underline{\mathrm{fl}}].L \Rightarrow \ell_{\mathrm{L}} \qquad pc' = \Psi[\underline{\mathrm{fl}}].pc \qquad \Sigma; \Gamma, x : \mathsf{fl}^{pc'}; \mathcal{T}; pc'; \ell_{\mathrm{L}} \vdash e' : \tau \dashv \Psi' \end{array}}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash \mathsf{atomic}\ e\ \mathsf{rescue}\ x : \mathsf{fl}\ e' : \tau \dashv (\Psi \setminus \underline{\mathrm{fl}}) \vee \Psi'}$$

[Throw]

$$\frac{\Sigma; \Gamma; \mathcal{T} \vdash v : ex^{\ell}}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash \mathsf{throw}\ v : \tau \dashv \{ex \mapsto (pc \vee \ell, \ell_{\mathrm{L}})\}}$$

[Fail]

$$\frac{\Sigma; \Gamma; \mathcal{T} \vdash v : t^{\ell}}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash \mathsf{fail}\ v : \tau \dashv \{\underline{\mathrm{fl}} \mapsto (pc \vee \ell, \ell_{\mathrm{L}})\}}$$

(c) Core expression typing

[SinglePath]

$$\frac{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash e : \tau \dashv \Psi \qquad pc' = \Psi[p].pc \wedge (pc \vee \bigvee_{p' \in \mathrm{dom}(\Psi), p' \neq p} \Psi[p'].pc)}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash e : \tau \dashv \Psi[p \mapsto pc']}$$

(d) Single Path Rule

[AtPc]

$$\frac{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash s : \tau \dashv \Psi}{\Sigma; \Gamma; \mathcal{T}; pc'; \ell_{\mathrm{L}} \vdash s \text{ at-pc } pc : \tau \dashv \Psi}$$

[Transact]

$$\frac{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash e : \tau \dashv \Psi \qquad \mathcal{T} \vdash \Psi[\underline{\mathrm{fl}}].L \Rightarrow \ell_{\mathrm{L}} \qquad pc_{\mathsf{fl}} = \Psi[\underline{\mathrm{fl}}].pc \qquad \Sigma; \Gamma, x : \mathsf{fl}^{pc_{\mathsf{fl}}}; \mathcal{T}; pc_{\mathsf{fl}}; \ell_{\mathrm{L}} \vdash e' : \tau \dashv \Psi'}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash \text{trans } e \text{ rescue } x : \mathsf{fl}\ e' : \tau \dashv (\Psi \setminus \underline{\mathrm{fl}}) \vee \Psi'}$$

[WithLock]

$$\frac{\Sigma; \Gamma; \mathcal{T}; pc; \ell'_{\mathrm{L}} \vdash s : \tau \dashv \Psi' \qquad \mathcal{T} \vdash \ell'_{\mathrm{L}} \wedge \ell \Rightarrow \ell_{\mathrm{L}} \qquad \mathrm{dom}(\Psi) = \mathrm{dom}(\Psi') \qquad (\Psi'[p].pc = \Psi[p].pc)^{p \in \mathrm{dom}(\Psi)} \qquad (\mathcal{T} \vdash \Psi'[p].L \wedge \ell \Rightarrow \Psi[p].L)^{p \in \mathrm{dom}(\Psi)}}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash s \text{ with-lock } \ell : \tau \dashv \Psi}$$

[Return]

$$\frac{\Sigma; \cdot; \mathcal{T}; pc; \ell'_{\mathrm{L}} \vdash s : \tau \dashv \Psi' \qquad \mathrm{dom}(\Psi) = \mathrm{dom}(\Psi') \qquad (\Psi'[p].pc = \Psi[p].pc)^{p \in \mathrm{dom}(\Psi)} \qquad (\mathcal{T} \vdash \Psi'[p].L \vee \ell'_{\mathrm{L}} \Rightarrow \Psi[p].L)^{p \in \mathrm{dom}(\Psi)}}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash \mathrm{return}_{\overline{ex}}\, s : \tau \dashv \Psi}$$

(e) Tracking statement typing

[IgnoreLocks]

$$\frac{\Sigma; \Gamma; \mathcal{T}; pc; \ell'_{\mathrm{L}} \vdash e : \tau \dashv \Psi' \qquad \mathrm{dom}(\Psi) = \mathrm{dom}(\Psi') \qquad (\Psi'[p].pc = \Psi[p].pc)^{p \in \mathrm{dom}(\Psi)}}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash \text{ignore-locks-in } e : \tau \dashv \Psi}$$

[AttackCast]

$$\frac{\Sigma; \Gamma; \mathcal{T} \vdash v : D^{\ell}}{\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash \text{atk-cast } v \text{ as } C : C^{\ell} \dashv \{\underline{\mathrm{n}} \mapsto (pc, \ell'_{\mathrm{L}})\}}$$

(f) Attacker-model expression typing

**Fig. 16.** Full typing rules for SCIF values, expressions, and statements.

[Method-Ok]

$$\frac{\begin{array}{c}\ell_{\text{L}} \Rightarrow pc_2 \qquad pc_1 \triangleleft \overline{\tau_a} \\ \Sigma; \overline{x}:\overline{\tau_a}, \text{this}:C^{pc_2}; \{\}; pc_2; \ell_{\text{L}} \vdash e : \tau \dashv \Psi \\ \text{dom}(\Psi) \subseteq \{\overline{ex}, \underline{\text{n}}, \underline{\text{fl}}\} \qquad (\ell_{\text{L}} \vee \Psi[p].L \Rightarrow \lambda_{\text{O}})^{p \in \text{dom}(\Psi)} \qquad (\Psi[ex].pc \Rightarrow \ell_{ex})^{ex \in \overline{ex^{\ell_{ex}}}} \\ \Psi[\underline{\text{n}}].pc \Rightarrow \ell_0 \qquad CT(C) = \text{contract } C \text{ extends } D\ \{\cdots\} \qquad \overline{can\text{-}override(D, m, \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \lambda_{\text{O}}} \tau)}\end{array}}{\Sigma \vdash \tau : \ell_0\ m\{pc_1 \gg pc_2; \lambda_{\text{O}}\}(\overline{x}:\overline{\tau_a}) \text{ throws } \overline{ex^{\ell}}\ \{e\} \text{ ok in } C}$$

[Class-Ok]

$$\frac{\begin{array}{c}fields(D) = \overline{g}:\overline{\tau_g} \\ K = C(\overline{g}:\overline{\tau_g}\ ;\ \overline{f}:\overline{\tau_f})\ \{\text{super}(\overline{g})\ ;\ \text{this}.\overline{f} = \overline{f}\} \\ \Sigma \vdash \overline{M} \text{ ok in } C\end{array}}{\Sigma \vdash \text{contract } C \text{ extends } D\ \{\overline{f}:\overline{\tau_f}\ ;\ E\ ;\ K\ ;\ \overline{M}\} \text{ ok}}$$

[CT-Ok]

$$\frac{\begin{array}{c}C \text{ referenced in any type} \implies C \in \text{dom}(CT) \\ \forall C \in \text{dom}(CT).\ \Sigma \vdash CT(C) \text{ ok}\end{array}}{\Sigma \vdash CT \text{ ok}}$$

(a) Class typing

$$\frac{CT(C) = \text{contract } C \text{ extends } D\ \{\overline{f}:\overline{\tau_f}\ ;\ E\ ;\ K\ ;\ \overline{M}\} \qquad fields(D) = \overline{g}:\overline{\tau_g}}{fields(C) = \overline{g}:\overline{\tau_g}\ ;\ \overline{f}:\overline{\tau_f}}$$

$$\frac{CT(C) = \text{contract } C \text{ extends } D\ \{\overline{f}:\overline{\tau_f}\ ;\ E\ ;\ K\ ;\ \overline{M}\} \qquad \tau:\ell_0\ m\{pc_1 \gg pc_2; \lambda_{\text{O}}\}(\overline{x}:\overline{\tau_a}) \text{ throws } \overline{ex^{\ell}}\ \{e\} \in \overline{M}}{\begin{array}{c}mtype(C, m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \lambda_{\text{O}}} \tau \\ mbody(C, m) = \left(\overline{x}, pc_1 \gg pc_2, e, \overline{ex}\right)\end{array}}$$

$$\frac{CT(C) = \text{contract } C \text{ extends } D\ \{\overline{f}:\overline{\tau_f}\ ;\ E\ ;\ K\ ;\ \overline{M}\} \qquad m \text{ not defined in } \overline{M}}{\begin{array}{c}mtype(C, m) = mtype(D, m) \\ mbody(C, m) = mbody(D, m)\end{array}}$$

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

(b) Lookup functions

$$\frac{\mathcal{T} \vdash \ell \Rightarrow \ell'}{\mathcal{T} \vdash t^{\ell} <: t^{\ell'}} \qquad \frac{CT(C) = \text{contract } C \text{ extends } D\ \{\cdots\}}{C^{\ell} <: D^{\ell}} \qquad \frac{\mathcal{T} \vdash \tau_1 <: \tau_2 \qquad \mathcal{T} \vdash \tau_2 <: \tau_3}{\mathcal{T} \vdash \tau_1 <: \tau_3}$$

(c) Subtyping

$$\frac{\mathcal{T} \vdash \ell \Rightarrow \ell'}{\mathcal{T} \vdash \ell \triangleleft t^{\ell'}}$$

(d) Protection

**Fig. 17.** Typing rules for SCIF classes, auxiliary lookup functions, and relations.