A Language for Smart Contracts with Secure Control Flow

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

Securing smart contracts remains a fundamental challenge. Because bugs can be so damaging, security audits are standard practice prior to deployment. However, even with expert inspection and use of heuristic tools, new security exploits frequently cause losses of tens of millions of dollars. Some of the most challenging classes of vulnerabilities for auditors to identify arise from complex control flow among multiple contracts. Examples of such control-flow attacks include reentrancy, confused deputies, and incorrect error handling.

We introduce SCIF, a new smart contract language based on Solidity that enforces secure information flow, meaning vulnerabilities can only arise where code explicit endorses untrusted inputs. Moreover, SCIF extends traditional information flow mechanisms to protect against all three classes of control-flow attacks listed above. SCIF thus greatly supports securing smart contracts both by removing some of the vulnerabilities that are the most difficult to audit, and by allowing auditors to focus their attention on endorsements. We describe the SCIF language and its implementation, including its static checking rules and runtime system. Finally, we show how SCIF supports building complex smart contracts securely and efficiently through several case studies of applications with intricate security reasoning.

1 Introduction

Smart contracts remain a promising platform for decentralized computation and storage, despite some setbacks. However, they are perhaps the clearest demonstration of the difficulty of building secure software compositionally. Even though smart contracts are usually carefully audited for security vulnerabilities, relying on an ever-expanding set of analysis tools and best practices, highly expensive vulnerabilities continue to emerge on a regular basis.

A core challenge is that a smart contract is not stand-alone program, but rather a fragment of an ever-growing on-chain library of code, whose future interactions are impossible to predict. Modern blockchain applications must interact securely with adversarial users and *their* contracts. This task is more challenging because the applications often comprise multiple smart contracts that implement complex protocols which transfer control among the contracts. These protocols create opportunities for adversaries to subvert intended control flow. Consequently, subtle vulnerabilities have often eluded even expert auditors, leading to total losses in the billions of dollars. These losses, along with the high cost of auditing [9, 10, 18], suggest that a better approach is needed.

This paper introduces SCIF (for Smart Contract Information Flow), a new smart contract language that is based on Solidity but adds features to both prevent security vulnerabilities and to aid smart contract auditors. These features particularly target the complex control-flow vulnerabilities that have proved so challenging. The paper presents the design and implementation of SCIF, and evaluates it on a variety of applications with subtle potential security vulnerabilities.

The first insight behind SCIF is to support security reasoning by analyzing information flow in programs to see where trusted data might be influenced by untrusted data influenced by adversaries.

Authors' Contact Information: Siqiu Yao, Cornell University, Ithaca, New York, USA, yaosiqiu@cs.cornell.edu; Haobin Ni, Cornell University, Ithaca, New York, USA, haobin@cs.cornell.edu; Stephanie Ma, Cornell University, Ithaca, New York, USA, ym363@cornell.edu; Noah Schiff, Cornell University, Ithaca, New York, USA, nps39@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.

Language-based information flow control [84] effectively supports auditing and constraining even code provided by determined attackers [40, 70], and it has yielded benefits and provable guarantees in some previous work on smart contracts [13, 16, 17].

Information flow analysis is a good starting point but not a complete solution. Traditional information flow control (IFC) aims to enforce noninterference [46], which in the context of smart contracts would mean that untrusted parties have no influence on trusted data [8]. Of course, many contracts need to allow such influence to do their job; in prior work this kind of flexibility is usually obtained through *endorsement* [99], a *downgrading* mechanism for information integrity. All violations of secure information flow must involve some form of endorsement. The idea of SCIF is that all endorsement must be explicit in the code, allowing programmers and auditors to focus their attention on a limited set of code locations.

While information flow analysis is central to SCIF, traditional IFC [84] does not adequately handle complex control-flow vulnerabilities. From the IFC viewpoint, these methods fall short because they do not detect the complex control-flow attacks above: the attacks either effectively introduce *implicit* endorsement of information, invisible to programmers and auditors, or they subvert existing explicit endorsements. Therefore, SCIF extends traditional IFC with new mechanisms that address these vulnerabilities:

- **Confused deputy prevention:** Confused deputy attacks (CDAs) are a long-known vulnerability of complex systems, and have quickly become a concern for smart contracts [23, 26, 27]. A whitelist of user and target contracts has been practiced to prevent CDAs, but this approach is either too restrictive or fails to block all attacks [26]. One challenge has been the absence of a crisp definition of what constitutes a CDA. This work develops a principled, more general definition and a new mechanism for comprehensively preventing CDAs.
- **Secure reentrancy:** Reentrancy vulnerabilities can occur in many settings [33], but have been particularly damaging in smart contracts. Dating back to The DAO [78], numerous reentrancy attacks have cost smart contracts hundreds of millions of dollars [24, 25, 28, 31, 75]. Best practices for coding smart contracts help but are insufficient to prevent damaging attacks [30]. The SeRIF calculus [17] showed that secure reentrancy is best viewed as an information-flow property. SCIF extends this approach to be more flexible, allowing more contracts to be implemented securely, and harmoniously integrates secure reentrancy 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 [29], failure to effectively reason about errors has led to damaging attacks [61, 97]. 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.

For reasons of space, some features of SCIF are left out of this paper, such as its support for run-time trust management. Although SCIF is a new programming language, it is based on Solidity, so auditors can use complementary analysis tools that have proved useful, such as automated tools for smart contract analysis based on model checking and fuzzing (e.g., [37, 43, 44, 51, 58, 67, 72, 82, 92]). However, SCIF has the advantage of integrating security reasoning directly into the design process.

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.

The SCIF implementation [1] and reference manual [4] are publicly available.

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) [34, 79].

2.2 Integrity via Information Flow

The core security enforcement mechanism of SCIF is information flow control (IFC) [84]. Each expression has a security label ℓ 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 ℓ_2,"' if information with label ℓ_1 can securely influence information with label ℓ_2.

IFC is typically used to enforce data confidentiality, but it is used in SCIF to enforce integrity by preventing influence of trusted data by untrusted sources, as originally proposed by Biba [8] and implemented in some prior work [66, 99].

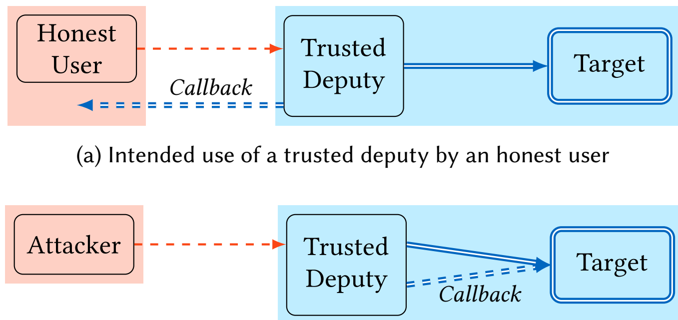
For instance, the SCIF code snippet on the right declares variables `x` and `y` with labels `untrusted` and `trusted`: it is safe for trusted information to affect untrusted information (`trusted` ⇒ `untrusted`), so information flow from `trusted` to `untrusted` is permissible, as demonstrated in line 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 [84], which captures influences on control flow.

Through the lens of information flow, smart contract vulnerabilities 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 [12, 16]. The SCIF type system is designed to directly prevent such insecure 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 noninterference. Such limited influence is supported in IFC systems through the downgrading



(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 controlled by the untrusted user, 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.

operation of *endorsement* [99], 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 [38, 63, 66, 69, 100] 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 should prompt anyone reading the code, including programmers to think carefully about its use. By contrast, in Solidity [89], standard programming patterns *implicitly* endorse both data and control flow, resulting in vulnerabilities.

2.3 Confused Deputy Attacks

One subtle class of attacks resulting from implicit endorsement is confused deputy attacks (CDAs). A CDA occurs when an attacker manipulates a trusted party (the confused deputy) into misusing its authority, enabling data corruption and attacker-controlled behavior.

Figure 1 depicts a common form of CDA. An untrusted user interacts with a trusted deputy; the deputy then 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. Since the attacker chooses which contract is called, the target implicitly endorses the choice of the attacker when it accepts the call from the deputy.

CDAs have resulted in damaging attacks on even carefully audited smart contracts. One emblematic CDA attacked Dexible [27], 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 [96]: 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

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

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

```
1  contract Uniswap {
2    Token tX, tY;
3
4    function sellXForY(uint xSold)
5        returns uint {
6      uint prod = tX.getBal(this) *
7                  tY.getBal(this);
8      uint yKept = prod /
9            (tX.getBal(this) + xSold);
10     uint yBought = tY.getBal(this) - yKept;
11
12     assert tX.transferFrom(msg.sender,
13                            this, xSold);
14     assert tY.transfer(this, msg.sender,
15                        yBought);
16     return yBought;
17 } }
```

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

the exchange contract in `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 [27]. Since the transfer request originated from Dexible itself—acting as a confused deputy—the token contract accepted the call and transferred the tokens, implicity endorsing the attacker's request.

2.4 Reentrancy Vulnerabilities

Another kind of attack exploiting endorsement 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 [24, 25, 28, 31, 75, 78].

The Uniswap token exchange fell victim to a reentrancy vulnerability in 2020 [75], showing that the combination of multiple contracts—each seemingly secure in isolation—can be vulnerable. Figure 3 shows a simplified segment of Uniswap 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 [90], but their combination unwittingly opens the door to reentrancy attacks. During the invocation of `transferFrom` at line 13, the client receives a notification, giving it control of execution and allowing an attacker 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. [17] 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 these methods have an annotation `@public`, this form of endorsement (*auto-endorsement* [20]) is explicitly signaled 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 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 [98] concluded that "almost all (92%) of the catastrophic system failures are the result of incorrect handling of non-fatal errors explicitly signaled in software." Another study [68] found that "nearly 70% of the examined smart contracts are exposed to potential failures due to missing error handing, e.g., unchecked external calls."

In Solidity, contracts can throw exceptions, which revert state changes within the ongoing transaction, and can 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 [14] 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) [62], a smart contract application whose participants compete for a prize by sending the contract more money than the current prize. Figure 4 has a simplified version of the KoET's `claimThrone` method. Line 8 attempts to compensate the previous winner using a low-level call, but if the call throws an exception, it merely produces a return value of `false`, which is ignored. As a result, the compensation remains in the KoET contract, and the method continues, updating a new winner. This vulnerability led to a shutdown and replacement of the KoET contract in 2016 [61]. To prevent such vulnerabilities, we need a clear, principled way to handle exceptions.

```
1  contract KoET {
2    address monarch;
3    uint claimPrice;
4
5    function claimThrone() external {
6      require(msg.value == claimPrice);
7      uint retVal = calcReturn(claimPrice);
8      monarch.send(retVal);
9      monarch = msg.sender;
10     claimPrice = calcNewPrice(claimPrice);
11 } }
```

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

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 $\{pc_{\text{ex}} \to pc_{\text{in}}; \ell_{\text{L}}\}$. The *external pc label* pc_{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_{in} label separately denotes the *internal pc label*, specifying the integrity of the control flow when the method body begins execution. When pc_{in} is more trusted than pc_{ex}, the method explicitly endorses the control flow on entry. When pc_{in} is not specified, it defaults to pc_{ex}, so there is no endorsement. Finally, the *lock label* ℓ_{L}, adopted from Cecchetti et al. [17], specifies the *reentrancy lock* integrity this method respects: the key annotation used to prevent reentrancy attacks. If not specified, ℓ_{L} defaults to pc_{in}. If only pc_{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 [12, 74]. 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 requires an external *pc* label of `this`. These labels ensure the control flow and argument are sufficiently trusted to alter `owner`.

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

Fig. 5. Simplified Parity Wallet code in SCIF.

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, and signaling these potential vulnerabilities to anyone reading the code, while in Solidity the difference between insecure and secure implementations is 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 that CDAs follow a 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 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 [89] to minimize the complexity and cost of this check. Whereas Solidity uses a method's name and argument base types to perform method dispatch, 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 dangerous direct low-level calls that take a raw `bytes` argument to specify 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.

```
1  contract Dexible {
2    exception FailedSwap();
3
4    @public
5    void swap{sender}(IERC20 tokIn, IERC20 tokOut, IExchange router, uint amt)
6        throws (FailedSwap) {
7      atomic {
8          tokIn.approveFrom(sender, address(router), amt);
9          router.exchange(sender, tokIn, tokOut, amt);
10     } rescue * {
11         throw FailedSwap();
12 } } }
13
14 interface IExchange {
15   void exchange{user -> this}(final address user, IERC20{user} tokIn,
16                               IERC20{user} tokOut, uint{user} amt);
17 }
```

Fig. 6. Simplified Dexible implementation in SCIF.

Second, the `exchange` method of the `IExchange` interface includes explicit labels specifying that the router only needs the integrity of the user to execute. Using the approach described above, the SCIF compiler therefore automatically generates the dynamic type check as part of the call to `router.exchange` on line 9. 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 then compile. The call would require integrity `this`, 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, necessitating additional run-time 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 more subtle CDA attacks as well.

3.3 Reentrancy Attack Prevention

SCIF adopts and improves the mechanism from SeRIF [17] for preventing reentrancy attacks, combining static and dynamic *reentrancy locks* to prevent reentrant endorsement, so that reentrant calls do not enable 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.

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

Fig. 7. Simplified Uniswap implementation in SCIF.

SCIF maintains the security of SeRIF's reentrancy protection, while improving precision to allow 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 [96].

Following the ERC-20 standard [96], 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 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 Uniswap, `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` requires high integrity and is called after `transferFrom` returns, the dynamic lock on line 8 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, but that would preclude notifying 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 [29]. In SCIF, a failure at line 9 automatically reverts all state changes in `claimThrone` and propagates to the caller, preventing the original attack.

Example (Town Crier). Town Crier (TC) [103] 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 16) fails, TC needs to keep the fee, so it must *not* roll back the entire call. Hence, it wraps the line 16 in an `atomic–rescue` block to catch the failure, and intentionally discards it. Failure rolls back the user-supplied operation, but not fee delivery.

```
contract KoET {
  address monarch;
  uint claimPrice;

  @public
  void claimThrone() {
    assert value == claimPrice;
    uint retVal = calcReturn(claimPrice);
    send(monarch, retVal);
    monarch = sender;
    claimPrice = calcNewPrice(claimPrice);
  }
}
```

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.

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_{ex}, SCIF automatically inserts a dynamic check at the top of the method that is equivalent to `assert sender =>` pc_{ex}.

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.

```
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`

- The `trusts` method determines 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` as well as more complex implementations. However, programmers may freely provide their own implementations, 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).

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

Fig. 11. Syntax for Core SCIF

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 [17] 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 [55] augmented with standard mutable references [76, Chapter 13], information-flow labels, and reentrancy protection.

Figure 11 presents the syntax of Core SCIF. Integrity labels in SCIF may be the constants this, any, a contract address α, 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 similarly to SeRIF [17]. New features include exceptions and failures, and a more accurate treatment of contract addresses.

SCIF expressions are mostly standard, with a few notes. To simplify the language, expressions generally are (open) values lacking subexpressions. The exception is let-expressions, which encode sequential composition. Second, SCIF can condition on dynamical trust tests $v_1 \Rightarrow v_2$, 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 entirely reverts if a failure occurs, whether or not it is rescued.

4.1 Type System

The type system of Core SCIF also extends that of SeRIF [17] to support features including dynamic trust checks 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 Σ is a heap type, mapping heap locations and contract addresses to types, Γ 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. $\mathcal{T}$ consists of a set of these flows, and in typing rules we write $\mathcal{T} \vdash \ell_1 \Rightarrow \ell_2$ to check that the flow holds in the current environment.[1]

Expression judgments $\Sigma; \Gamma; \mathcal{T}; pc; \ell_{\mathrm{L}} \vdash e : \tau \dashv \Psi$ are a bit more complicated. Here Σ, Γ, and $\mathcal{T}$ are as above, and pc is the standard program-counter label. The lock label ℓ_{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 [69], the context Ψ tracks the integrity of these different possible termination paths. A path can either be normal termination ($\underline{\mathsf{n}}$), an

[1] 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.

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

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

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

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

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

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

Fig. 12. Selected typing rules for Core SCIF

exception (ex), or $\underline{\mathsf{fl}}$, denoting a failure. SCIF tracks both the integrity of the control flow and reentrancy locks, so Ψ maps possible termination paths to pairs of labels (pc, L).

Figure 12 shows selected typing rules for Core SCIF.[2] 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.

Rule Let defines sequential composition: a mostly standard let-binding that also enforces information flow and reentrancy security with greater precision than in SeRIF [17]. 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 ℓ_{L}, e_1 must maintain that lock. Formally, for any integrity level ℓ, if $pc' \Rightarrow \ell$ and $\ell_{\text{L}} \Rightarrow \ell$, then $\Psi_1[\underline{\mathsf{n}}].L \Rightarrow \ell$. The condition $\mathcal{T} \vdash \Psi_1[\underline{\mathsf{n}}].L \Rightarrow \ell_{\text{L}} \vee pc'$ 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: $\Psi_1[\underline{\mathsf{n}}].L \vee \ell_{\text{L}}$.

Rule IfTrust describes dynamic flows-to checks. To make this check practical, SCIF only checks 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 $v_1 \not\Rightarrow v_2$ when checking e_2.

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

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

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. Throw indicates that only exceptional termination (with the correct exception type) is possible. Since TryCatch handles exception *ex*, the possible return paths are the *other* return paths of the body, plus any possible return paths of the catch block. AtomicRescue is nearly identically to TryCatch, but it uses the single distinguished failure path $\underline{\mathsf{fl}}$. It also requires that the body can only terminate normally or with a failure, not with uncaught exceptions. It is unclear an unhandled exception in an atomic block should revert, as each option violates the expectations of either exceptions (commit) or atomic (revert). SCIF therefore disallows this situation, requiring the programmer to specify the desired behavior in each case.

4.2 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_{env}. The method signature specifies the integrity required to invoke the method in its external *pc* label pc_{ex}. Standard information flow checking demands that $pc_{\text{env}} \Rightarrow pc_{\text{ex}}$. A CDA occurs if a call violates this requirement.

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

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

The Call typing rule ensures that SCIF code is free from ℓ-CDA events at all labels ℓ when every contract type-checks. Unfortunately, smart contract environments are open systems with no guarantee that attacker-provided code is well-typed. To simplify the challenges of reasoning about ill-typed code, our core calculus introduces a special atk-cast term that empowers attackers to provide contracts of the wrong type as arguments to trusted functions:

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

The type system handles atk-cast like a regular cast, but with almost no semantic validation. The lack of validation adds power that seems 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 could 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 run-time check: the type of the called method must be the type expected. Executing $(\mathsf{atk\text{-}cast}\ C(\overline{v})\ \mathsf{as}\ D).m(\overline{w})$ thus requires the types of $D.m$ and $C.m$ to exactly match, and otherwise behaves like a normal call to $C.m$. By eliminating type confusion on method calls, SCIF ensures that $pc_{\text{env}} \Rightarrow pc_{\text{ex}}$ for every call, and therefore eliminates all ℓ-CDA events. A more permissive sound rule would be to require only a subtyping relationship between $D.m$ and $C.m$ (with pc contravariance), but this rule would be far harder to implement. So, we opt for the simpler requirement that the types match exactly.

5 Implementation

The SCIF compiler consists of just over 12,000 lines of Java code. It uses JFlex [60] and CUP [54] for parsing and does type checking by generating type constraints that it passes to the SHErrLoc constraint solver [102]. The compiler outputs Solidity code in which labels have been erased and run-time mechanisms have been inserted.

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

5.1 Run-time Mechanisms

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

5.1.1 CDA Prevention. As described in Section 3.2, our compiler prevents CDAs by leveraging Solidity's method dispatch mechanism to ensure the caller and callee agree on information flow labels. of the generated Solidity code. Any type confusion would cause a mismatch in dispatch hashes between the caller and the callee, and dispatch would fails, preventing attacks 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. The beginning of each public method confirms the trustworthiness of the caller by asserting that `trusts`(pc_{ex}, `sender`). If there may be an autoendorsement ($pc_{\text{ex}} \not\Rightarrow pc_{\text{in}}$ statically), we include a second assertion that `trusts`($pc_{\text{in}}, pc_{\text{ex}}$) || `bypassLocks`($pc_{\text{ex}}$), ensuring no dynamic reentrancy locks block this call. Compiling `lock(l) {...}` is simply `acquireLock(l)`, then the compiled contents, then `releaseLock(l)`. Finally, explicit trust relationship queries `l1 => l2` translate to `trusts(l2, l1)`.

5.2 Exception Handling

Solidity's exception handling is limited to a single external call and only reverts when there is a failure [91]. As a result, it cannot be directly used to implement SCIF exceptions, which do not roll back. Instead, non-failure termination results are encoded in method return values. A SCIF method that can 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 the exception arguments otherwise.

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. One important optimization is to avoid a trampoline when the `atomic` block holds a single external function call that throws no checked exceptions.

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	3759		26
ERC-20 2	88	0.55	9	19	2960	2097	20
ERC-20 3	83	0.59	8	18	2704		20
Uniswap	270	113	47	57	12,653	10,553	74
Dexible swap	29	0.13	0	2	3488	*	0
KoET	164	3.61	2	6	3563	2758	10
Poly Network	115	2.55	6	8	6142	*	8
HODLWallet	73	0.33	8	13	1669	1991	10
SysEscrow	138	0.62	3	6	2284	2439	13

Table 1. SCIF case studies.

5.3 Limitations

Our current SCIF implementation has some limitations:

(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. We evaluated the expressiveness and effectiveness of SCIF in two ways: first, by implementing and analyzing several challenging real-world smart contracts, and second, by applying SCIF to randomly chosen smart contracts from two large collections of insecure contracts.

6.1 Case studies

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

Compilation times range from under a second to nearly 2 minutes. Most time is spent in 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 run-time security enforcement (RSE) calls in the compiled code. While these metrics do not directly represent the run-time overhead, they illuminate the complexity introduced by 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 complex functionality unrelated to the core vulnerabilities. Our bytecode is shorter on HODL Wallet and SysEscrow because they require much older, less optimized, Solidity compiler versions than we use.

We also evaluated the overhead of SCIF's run-time security mechanisms by comparing the gas consumptions of security-critical operations of several contracts in both Solidity and SCIF. Table 2 presents the average of 100 runs of each operation.

Case	Operation	Solidity (version)		SCIF	Overhead
ERC-20 (1/2/3)	`approve`	1377	(0.8.28)	1516 / 1331 / 1331	10% / -5% / -5%
	`transfer`	2229		3442 / 3265 / 2740	54% / 46% / 23%
	`transferFrom`	3088		4148 / 3934 / 3605	35% / 27% / 18%
Uniswap (w/ ERC-20 2)	`tokenToExchangeSwapInput`	17,725	(0.5.17)	20,703	17%
	`ethToTokenSwapInput`	11,771		12,756	8%
	`tokenToTokenSwapInput`	18,010		20,430	13%
KoET	`claimThrone`	110,034	(0.4.26)	133,293	21%
HODLWallet	`withdrawTo`	9412	(0.4.26)	10,284	10%
	`depositTo`	8384		8652	3%
	`withdrawForTo`	9600		10,645	11%
SysEscrow	`createEscrow`	99,770	(0.4.26)	101,306	2%
	`cancelEscrow`	38,508		42,826	11%
	`approveEscrow`	28,228		28,360	0.4%
	`releaseEscrow`	41,550		45,392	9%

Table 2. Gas consumption (wei) of methods in Solidity and SCIF. The SCIF backend uses Solidity v0.8.28.

The gas consumption overhead due to SCIF's run-time security mechanisms may seem large for some operations such as `transfer`. However, the bill for compositional security must be paid somewhere. If ERC-20 does not enforce security, other contracts that interact with it must implement their own guards. Moving the overhead into ERC-20 is not inherently less efficient, and it simplifies securing larger applications. The cost of SCIF's checks is relatively small in absolute terms, as shown by the much lower overhead for larger applications like Uniswap, which uses ERC-20.

Case Study (ERC-20). Our ERC-20 implementation follows that of OpenZeppelin ERC-20 [73]. 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.

Table 2 shows run-time overhead of each implementation compared to Solidity, measured by averaging the gas consumption over 1,000 executions of each operation. The finer-grained ERC-20 2 has lower overheads because its use of a dependent map allows it to avoid auto-endorsements and their costly dynamic security checks. The overhead for `transfer` and `transferFrom` looks large because these methods do very little real work. It mostly stems from SCIF's exception-handling mechanism, which could be optimized significantly. ERC-20 3 is a modified version of ERC-20 2 that removes checked exceptions, reducing run-time overhead.

Case Study (Uniswap). As shown in Table 1, we implement Uniswap V1 [95] in SCIF. Our implementation addresses the vulnerability by only allowing interactions with ERC-20 tokens (no callbacks). We compared gas consumption of the original Uniswap V1 with our SCIF implementation, using Solidity and SCIF ERC-20 2 implementations, as shown in Table 2.

Case Study (Dexible Swap). For Dexible, we implemented only the core swap functionality, making direct comparison to the original implementation [41] 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 [39], 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 [77], 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 [82] 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 [82] 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.

6.2 Using SCIF on contracts in the wild

To understand the effectiveness of SCIF as a tool for making secure smart contracts, we reimplemented contracts from two large collections of insecure contracts: DAppScan [108] and CGT [36]. Reimplementing all of the contracts was not feasible, so we sampled a representative set of contracts randomly from the collections.

Reentrancy. Among 114 contracts in the DAppScan corpus labeled as having reentrancy vulnerabilities, we randomly chose 13. In each case, SCIF both detected the vulnerability and enabled a straightforward fix.

CDA. Only one contract from DAppScan was marked with a CDA vulnerability. SCIF detected it and allowed successfully reimplementing the contract securely.

Insecure contracts. We randomly selected 20 contracts by choosing the contracts with the smallest hash values from the CGT corpus, restricting to only real vulnerabilities rather than "code smells." Those contracts contained 35 identified vulnerabilities, of which SCIF successfully diagnose 16. The other 19 all fall into categories SCIF does not attempt to address: transaction order dependency (7), integer over/underflow (8), use of block values as a proxy for time (2), and airdrop Sybil attacks (2).

The list of the sampled contracts is included in the supplementary material.

7 Related Work

Confused Deputy Attacks. Rajani et al. [80] formally define CDA freedom as a security property and prove that information flow security is sufficient to enforce it. Jagadeesan et al. [57] 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 [49] analyzes EVM bytecode for CDAs using symbolic execution, but does not cover multiple-contract CDAs, and is incomplete by nature.

Reentrancy Security. Our reentrancy security mechanisms improve on those of SeRIF [17], which defines a formal notion of ℓ-reentrancy and an information flow type system to enforce reentrancy security. SCIF's flexible treatment of execution paths recognizes more code as secure.

Grossman et al. [50] and Albert et al. [3] 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 [15, 59, 65], 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 [83, 106], 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.

Secure Smart Contract Languages. The Scilla [86] language forces a programming style that separates pure computation, state changes, and method calls. Obsidian [22] and Flint [85] use resource types and typestate to aid reasoning about contract behaviors. The resource types guarantee that assets, such as tokens, cannot be arbitrarily created or destroyed. Nomos [35] introduces resource-aware session types, eliminating single-contract reentrancy. However, none of these languages can guard against CDAs or more sophisticated multi-contract reentrancy attacks.

Smart Contract Security Tools. Many stand-alone tools aim to find vulnerabilities in smart contracts. AI-based tools [2, 7, 88] provide no soundness or completeness guarantees. Bytecode modification tools that insert dynamic checks to prevent undesirable behaviors [71, 105] 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 [11, 13, 32, 47, 53, 64, 93, 94], sometimes using symbolic execution or model checking [37, 43, 44, 51, 58, 67, 72, 92]. 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, 48] provide high assurance but require significant user expertise and verification effort.

TxT [56] and Hoang et al. [52] are testing frameworks for smart contracts with additional support for testing security guarantees. FuzzDelSol [87] and EF\CF [82] apply fuzzing to smart contracts. Testing-based methodologies have low development overhead but are incomplete and have trouble dealing with the huge space of possible attacker contracts and transactions.

Sereum [81], ÆGIS [42], STING [107], SODA [19], and TxSpector [104] use run-time monitoring or verification to detect and defense against potential attacks, but they require cooperation from blockchain miner nodes, which are beyond the control of most smart-contract programmers.

Information Flow Control. Prior work uses IFC to secure decentralized systems. Fabric [66] provides a language to build distributed systems where nodes can securely share code and data despite mutual distrust. DStar [101] connects information flows within the operating system to secure distributed executions. These systems focus more traditional distributed computing rather than smart contracts, and fail to provide reentrancy security.

8 Conclusion

SCIF provides sorely needed assurance that smart contracts are not vulnerable to control-flow attacks including reentrancy, confused deputy attacks, and improper error handling. We introduce a more general, principled integrity-based 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. By applying SCIF to a wide variety of real-world examples, we see not only its effectiveness for improving security but also a harmonious integration of 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] [n. d.]. SCIF implementation. https://neroysq.github.io/SCIF/

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

[3] 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

[4] Anonymous. 2023. SCIF Reference Manual. https://neroysq.github.io/SCIF/

[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] K. J. Biba. 1977. *Integrity Considerations for Secure Computer Systems*. Technical Report ESD-TR-76-372. USAF Electronic Systems Division, Bedford, MA. https://ban.ai/multics/doc/a039324.pdf (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[9] 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.

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

[11] 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

[12] 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.

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

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

[15] 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

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

[17] 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
[18] Chainlink. 2024. How To Audit a Smart Contract. https://chain.link/education-hub/how-to-audit-smart-contract. Accessed March 2024.
[19] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, Yuxing Tang, Xiaodong Lin, and Xiaosong Zhang. 2020. SODA: A Generic Online Detection Framework for Smart Contracts. *Proceedings 2020 Network and Distributed System Security Symposium* (2020). https://doi.org/10.14722/ndss.2020.24449
[20] Stephen Chong, K. Vikram, and Andrew C. Myers. 2007. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *16th USENIX Security Symp.* http://www.cs.cornell.edu/andru/papers/sif.pdf
[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. 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.
[24] 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.
[25] 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.
[26] CoinDesk. 2024. Defi Protocol LI.FI Struck by $11M Exploit. https://www.coindesk.com/business/2024/07/16/defi-protocol-lifi-struck-by-8m-exploit/. Accessed October 2024.
[27] 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.
[28] 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.
[29] 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.
[30] 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.
[31] 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.
[32] 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
[33] 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.
[34] 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
[35] 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.
[36] Monika di Angelo and Gernot Salzer. 2024. Consolidation of Ground Truth Sets for Weakness Detection in Smart Contracts. In *Financial Cryptography and Data Security. FC 2023 International Workshops*, Aleksander Essex, Shin'ichiro Matsuo, Oksana Kulyk, Lewis Gudgeon, Ariah Klages-Mundt, Daniel Perez, Sam Werner, Andrea Bracciali, and Geoff Goodell (Eds.). Springer Nature Switzerland, Cham, 439–455.
[37] 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
[38] 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). http://dl.acm.org/citation.cfm?id=1095813
[39] 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.

[40] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. 2014. Collaborative Verification of Information Flow for a High-Assurance App Store. In *21st ACM Conf. on Computer and Communications Security (CCS)*. 1092–1104.

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

[42] Christof Ferreira Torres, Mathis Baden, Robert Norvill, Beltran Borja Fiz Pontiveros, Hugo Jonker, and Sjouke Mauw. 2020. ÆGIS: Shielding Vulnerable Smart Contracts Against Attacks. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20)*. Association for Computing Machinery, New York, NY, USA, 584–597. https://doi.org/10.1145/3320269.3384756

[43] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. EthBMC: A Bounded Model Checker for Smart Contracts. In *29th USENIX Security Symp.* https://www.usenix.org/conference/usenixsecurity20/presentation/frank

[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] Joseph A. Goguen and José Meseguer. 1984. Unwinding and Inference Control. In *IEEE Symp. on Security and Privacy*. 75–86. https://doi.org/10.1109/SP.1984.10019

[47] 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

[48] 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.

[49] 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.

[50] 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

[51] 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.

[52] Tram Hoang, Anton Trunov, Leonidas Lampropoulos, and Ilya Sergey. 2022. Random Testing of a Higher-Order Blockchain Language (Experience Report). *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 122:886–122:901. https://doi.org/10.1145/3547653

[53] 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

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

[55] 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.

[56] Nikolay Ivanov, Qiben Yan, and Anurag Kompalli. 2023. TxT: Real-Time Transaction Encapsulation for Ethereum Smart Contracts. *IEEE Transactions on Information Forensics and Security* 18 (2023), 1141–1155. https://doi.org/10.1109/TIFS.2023.3234895

[57] 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

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

[59] 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

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

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

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

[63] 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)*. http://dl.acm.org/citation.cfm?id=1294293

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

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

[66] 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

[67] 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

[68] Charalambos Mitropoulos, Maria Kechagia, Chrysostomos Maschas, Sotiris Ioannidis, Federica Sarro, and Dimitris Mitropoulos. 2024. Charting the Evolution of Solidity Error Handling. https://arxiv.org/abs/2402.03186. arXiv:2402.03186 [cs.SE] https://arxiv.org/abs/2402.03186

[69] 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

[70] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer.

[71] 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

[72] 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

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

[74] 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.

[75] 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.

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

[77] 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.

[78] 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).

[79] 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

[80] 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

[81] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. *Proceedings 2019 Network and Distributed System Security Symposium* (2019). https://doi.org/10.14722/ndss.2019.23413

[82] 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.

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

[84] 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

[85] 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.

[86] 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.

[87] Sven Smolka, Jens-Rene Giesen, Pascal Winkler, Oussama Draissi, Lucas Davi, Ghassan Karame, and Klaus Pohl. 2023. Fuzz on the Beach: Fuzzing Solana Smart Contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1197–1211. https://doi.org/10.1145/3576915.3623178

[88] 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.

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

[90] 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.

[91] 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.

[92] 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

[93] 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.

[94] 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

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

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

[97] 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.

[98] 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)*.

[99] 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

[100] 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. http://dl.acm.org/citation.cfm?id=2018419

[101] 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. http://dl.acm.org/citation.cfm?id=1387610

[102] 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. http://dl.acm.org/citation.cfm?id=3121137

[103] 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

[104] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. 2020. TXSPECTOR: Uncovering Attacks in Ethereum from Transactions. In *29th USENIX Security Symposium (USENIX Security 20)*. 2775–2792.

[105] 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

[106] 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. http://www.cs.cornell.edu/andru/papers/exceptions

[107] Zhuo Zhang, Zhiqiang Lin, Marcelo Morales, Xiangyu Zhang, and Kaiyuan Zhang. 2023. Your Exploit Is Mine: Instantly Synthesizing Counterattack Smart Contract. In *32nd USENIX Security Symposium (USENIX Security 23)*.

1757–1774.

[108] Zibin Zheng, Jianzhong Su, Jiachi Chen, David Lo, Zhijie Zhong, and Mingxi Ye. 2024. DAppSCAN: Building Large-Scale Datasets for Smart Contract Weaknesses in DApp Projects. *IEEE Trans. Softw. Eng.* 50, 6 (March 2024), 1360–1373. https://doi.org/10.1109/TSE.2024.3383422

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

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

$$\text{[E-IFTRUSTT]}\quad \frac{\alpha_1 \Rightarrow \alpha_2}{\langle \mathsf{if}_{pc}\ \alpha_1 \Rightarrow \alpha_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid C\rangle \longrightarrow \langle e_1\ \mathsf{at\text{-}pc}\ pc \mid C\rangle} \qquad \text{[E-IFTRUSTF]}\quad \frac{\alpha_1 \not\Rightarrow \alpha_2}{\langle \mathsf{if}_{pc}\ \alpha_1 \Rightarrow \alpha_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid C\rangle \longrightarrow \langle e_2\ \mathsf{at\text{-}pc}\ pc \mid C\rangle}$$

$$\text{[E-ATPC]}\quad \frac{}{\langle v\ \mathsf{at\text{-}pc}\ pc \mid C\rangle \longrightarrow \langle v \mid C\rangle} \qquad \text{[E-REF]}\quad \frac{\iota \notin \mathrm{dom}(\sigma)}{\langle \mathsf{ref}\ v\ \tau \mid C\rangle \longrightarrow \langle \iota \mid C[\sigma[\iota \mapsto v]/\sigma]\rangle} \qquad \text{[E-DEREF]}\quad \frac{}{\langle !\iota \mid C\rangle \longrightarrow \langle \sigma(\iota) \mid C\rangle}$$

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

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

(a) IFC Calculus Small-Step Operational Semantic Rules

A Full SCIF Rules

The full operational semantics for SCIF are given in Figure 13 and the full typing rules are given in Figures 14 and 15. We introduce the following syntactic forms as evaluation contexts to enable precise tracking of method boundaries, execution integrity, dynamic locks, and type confusions:

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

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

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

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

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

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

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

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

$$[\textsc{E-TryRet}]\,\frac{}{\langle \mathsf{try}\ v\ \mathsf{catch}\ x\!:\!ex\ e \mid C\rangle \longrightarrow \langle v \mid C\rangle} \qquad [\textsc{E-AtomicCommit}]\,\frac{S = S', \sigma'}{\langle \mathsf{trans}\ v\ \mathsf{rescue}\ x\ e \mid C\rangle \longrightarrow \langle v \mid C[S/S']\rangle}$$

(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. 13. Full small-step operational semantics for SCIF.

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

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

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

(a) Value typing

[Val]
$$\frac{\Sigma;\Gamma;\mathcal{T}\vdash v:\tau}{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\mathrm{L}}\vdash v:\tau\dashv\{\underline{\mathrm{n}}\mapsto(pc,\ell'_{\mathrm{L}})\}}$$

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

[Cast]
$$\frac{\Sigma;\Gamma;\mathcal{T}\vdash v:D^{\ell}}{\Sigma;\Gamma;\mathcal{T};pc;\ell_{\mathrm{L}}\vdash (C)v:C^{\ell}\dashv\{\underline{\mathrm{n}}\mapsto(pc,\ell'_{\mathrm{L}})\}}$$

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

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

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

[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]

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

[CALL]

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

[IF]

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

[IFTRUST]

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

[ASSIGN]

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

[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]

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

[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]

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

[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]

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

(c) Core expression typing

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

(d) Single Path Rule

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

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

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

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

(e) Tracking statement typing

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

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

(f) Attacker-model expression typing

Fig. 14. Full typing rules for SCIF values, expressions, and statements.

[Method-Ok]

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

[Class-Ok]

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

[CT-Ok]

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

(a) Class typing

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

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

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

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

(b) Lookup functions

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

(c) Subtyping

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

(d) Protection

Fig. 15. Typing rules for SCIF classes, auxiliary lookup functions, and relations.