Securing Smart Contracts with Information Flow

Ethan Cecchetti Siqiu Yao Haobin Ni Andrew C. Myers

Cornell University
{ethan,yaosiqiu,haobin,andru}@cs.cornell.edu

Research Short Paper

ABSTRACT

Securing blockchain smart contracts is difficult, especially when they interact with one another. Existing tools for reasoning about smart contract security are limited in one of two ways: they either cannot analyze cooperative interaction between contracts, or they require all interacting code to be written in a specific language. We propose an approach based on information flow control (IFC), which supports fine-grained, compositional security policies and rules out dangerous vulnerabilities. However, existing IFC systems provide few guarantees on interaction with legacy contracts and unknown code. We extend existing IFC constructs to support these important functionalities while retaining compositional security guarantees, including reentrancy control. We mix static and dynamic mechanisms to achieve these goals in a flexible manner while minimizing run-time costs.

1. INTRODUCTION

Smart contracts run in an environment of unprecedented hostility. Attackers have full access to the code and a direct financial incentive to find and exploit vulnerabilities, yet bugs cannot be fixed in deployed code. Consequently, there is a pressing need for better tools build and reason about secure contracts. Making the problem harder, many contracts need to interact with other contracts and off-chain functionality that they may not trust fully.

Existing languages designed to provide strong correctness or security in smart contracts [1, 3, 16] generally assume that contracts interact only with other contracts also written in the same language. This strong assumption may seem reasonable for permissioned blockchains, but even there, contracts might need to interact with off-chain legacy applications that do not respect the language rules.

Other analysis tools [5, 6, 8] assume little about interacting components, but focus on the security of the contract as a single unit. This focus interferes with reasoning about pieces of contracts or collaborative combinations of multiple contracts. Analyzing two contracts independently may not translate into meaningful guarantees about their combination.

Third International Symposium on Foundations and Applications of Blockchain (FAB '20) May 1, 2020, Santa Cruz, California, USA.

We aim to address both of these concerns by using an information flow control (IFC) type system to track the integrity (trustworthiness) of information. While various IFC tools and languages protect the *confidentiality* of data [14, 15, 17] and have proven highly effective [4], all blockchain data is public. We instead use IFC similarly to protect *integrity* by preventing untrustworthy data from unexpectedly influencing trusted computation. We extend existing decentralized IFC models [11] that we find particularly well-suited to the decentralized nature of smart contracts.

Unfortunately, prior IFC systems cannot provide provable guarantees for most contracts that allow calls from unknown sources—a critical smart contract functionality. The basic rules of IFC prohibit callers from invoking code unless that code trusts the caller. This restriction prevents common contract bugs like reentrancy, but also prevents most interesting smart contracts from operating at all. Existing systems [7, 10] address this limitation by allowing designated *entry points* where untrusted code can call into trusted code, but these entry-point mechanisms provide few security guarantees. In particular, they reopen reentrancy vulnerabilities.

We describe our work on adapting IFC to verification of smart contract security. Our core technical contribution is a design for trusted entry points that fits into an IFC system without requiring programmers to worry about reentrancy, even in the presence of non-IFC legacy systems and unknown code. We retain previous IFC benefits, like the ability to flexibly compose contracts in which trust is not constrained by contract boundaries. We are implementing these features in a new smart contract programming language.

2. NEED FOR IFC

Many high-profile contract vulnerabilities can be viewed as integrity failures; untrusted inputs improperly influence high-integrity state, leading to improper money transfers.

Example: Parity Wallet. In 2017, an Ethereum wallet created by Parity Technologies suffered two critical attacks exploiting the interaction of multiple contracts designed to work together. The second attack [12], which froze $100 million of Ether in place, is more famous, but the first attack [2], where attackers stole over $30 million, is more illustrative.

Listing 1 shows a simplified version of the vulnerable code. To reduce deployment costs, Parity split the contract into two pieces: a library contract that was deployed once and defined the wallet's available operations, and an instance contract which each user deployed separately. The instance contract delegated to the library using Ethereum's `delegatecall` operation, which, in this case, executes the library contract's

```
contract WalletLibrary {
    address owner;
    function __init__(address _owner) public {
        owner = _owner;
    }
    ...
}

contract Wallet {
    WalletLibrary walletLibrary;
    address owner;
    ...
    fallback () external payable {
        walletLibrary.delegatecall(msg.data);
    }
}
```

Listing 1: Simplified vulnerable Parity Wallet

code in the instance contract's state space.

Unfortunately, the interaction exposed a serious bug. The library's `__init__` function was marked as public and set the owner of the wallet with no validity checks, so the instance constructor could use it. The wallet's `fallback` function, however, delegated any unknown call to the library, allowing a carefully crafted attack to bypass the wallet's ownership check and change the owner using `__init__`.

Viewed from an integrity perspective, the bug is straightforward. The `owner` variable should be high-integrity, but the wallet blindly sets it to a low-integrity value provided by an unknown user. Information flow control (IFC) is designed to identify and eliminate exactly this type of bug.

IFC Labels. IFC type systems tag data with labels. Our labels specify the set of addresses that can influence a piece of data, allowing us to track trustworthiness. Blockchains provide an ideal setting for this technique, as any call can easily check the caller's trust level—its address.

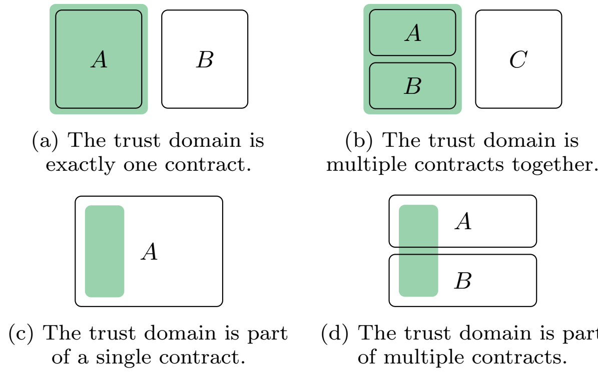
In our example, we may tag `owner` with the label ℓ_W, denoting that the wallet can influence it, and the argument to `fallback` as $\ell_{\mathcal{A}}$, denoting that an attacker may influence it. We include an *acts-for* relation between labels, written $\ell \Rightarrow \ell'$, to denote that ℓ is at least as trusted as ℓ'. That is, any address that can influence ℓ can also influence ℓ' [1] If the attacker trusts the wallet, we could include $\ell_W \Rightarrow \ell_{\mathcal{A}}$. As the wallet should not trust the attacker, we would not include a relation in the other direction.

Because `owner` has label ℓ_W, an IFC type system only allows an assignment `owner = x` if $\ell_{\texttt{x}} \Rightarrow \ell_W$, where $\ell_{\texttt{x}}$ is the label of `x`. In the above example, a new address provided by an attacker would have label $\ell_{\mathcal{A}}$, so an IFC system would reject the dangerous assignment.

IFC type systems also track the integrity of *computation* to prevent attackers from improperly influencing control flow. As we will see in Section 3, control-flow attacks can be just as damaging as data-flow attacks. To track the integrity of control flow, we associate each instruction with a program counter label *pc*. The signature of a function f also includes the integrity pc_f that f requires, and the type system requires a calling context's integrity to act for pc_f to invoke f.

These *pc* labels allow us to specify the trust level of different pieces of code within a contract. Unlike in existing tools, two pieces of code in the same contract can have different trust levels, while code in different contracts may have the same trust level. The Parity Wallet provides a perfect use case. The bug resulted from an unexpected interaction between two contracts that were *written to work together*, but only partially trust each other. In the instance contract, the `owner` variable must be high-integrity, but the `fallback` function must be low-integrity since the attacker can control it.

[1]Traditional IFC systems use *flows-to*, denoted $\ell \sqsubseteq \ell'$. As we only track integrity, flows-to and acts-for carry the same meaning and we find acts-for to be more intuitive.



(a) The trust domain is exactly one contract.

(b) The trust domain is multiple contracts together.

(c) The trust domain is part of a single contract.

(d) The trust domain is part of multiple contracts.

Figure 1: Different possible configurations of trust domains and contract boundaries we aim to express. In each example the trust domain is highlighted in green.

By labeling the trust level of individual instructions separately, IFC can say that some pieces of both the wallet and the library are trustworthy, but other pieces of both are not. Indeed, it can express trust domains entirely independently from contract boundaries. Figure 1 depicts several configurations of trust and contract boundaries. Prior smart contract analysis tools operate almost exclusively in the configuration of Figure 1a, but Figure 1d best represents the Parity Wallet.

3. ENTRY POINTS AND REENTRANCY

IFC systems constrain control flow using the *pc* label by requiring a caller to be at least as trusted as the function it is calling. The importance of this requirement is evident from the classic "reentrancy" attack. Consider a distributed bank with two contracts running identical code shown in Listing 2. A user can deposit money and withdraw money from either, and the two banks should keep their balances in sync. While this example may seem contrived, it simplifies the more realistic structure of an airline alliance or multi-company reward program where customers can earn and spend rewards with different alliance members.

As written, Listing 2 has two reentrancy vulnerabilities. One allows an attacker $\mathcal{A}$ to extract funds from a single contract. $\mathcal{A}$ deposits money and then calls `withdraw`. Line 8 returns $\mathcal{A}$'s money, but also passes control of execution to $\mathcal{A}$. Because the balance is not decreased until line 9, $\mathcal{A}$ can call `withdraw` again and extract double its original deposit. This bug resulted in the most famous smart contract hack to date when, in July 2016, an attacker drained \$50 million in tokens from Ethereum's Decentralized Autonomous Organization (DAO) [13].

Assuming sending money and modifying trustworthy state both require high integrity, IFC requires `withdraw` to operate with high integrity. The classic IFC restriction on invoking high-integrity functions then prevents a low-integrity attacker from invoking `withdraw` to execute this attack. Unfortunately, as stated, this constraint breaks correct functionality of the

```
1  contract DistributedBank {
2      DistributedBank otherBank;
3      mapping(address => uint) balances;
4
5      function withdraw(uint amount) {
6          if (balances[msg.sender] >= amount
7                  && this.balance >= amount) {
8              msg.sender.call{value: amount}("");
9              balances[msg.sender] -= amount;
10             otherBank.decreaseBal(msg.sender, amount);
11         }
12     }
13
14     function decreaseBal(address addr, uint amount) {
15         if (msg.sender == otherBank) {
16             balances[addr] -= amount;
17         }
18     }
19     ...
20 }
```

Listing 2: Multi-contract bank with two reentrancy bugs.

contract. The contract cannot distinguish honest users from attackers, so it must consider both untrustworthy, but then honest users would be unable to call `withdraw` at all!

Existing IFC systems [10] provide a mechanism to *endorse* control flow, thereby creating an entry point into high-integrity code. We introduce an operation `endorsepc`(ℓ), which endorses the *pc* label used to track the integrity of execution to the provided label ℓ. The `DistributedBank` contract then type-checks using `endorsepc` as follows:

```
function withdraw(uint amount) {
    if (balances[msg.sender] >= amount
            && this.balance >= amount) {
        endorsepc(to_label(this)) {
            msg.sender.call{value: amount}("");
            balances[msg.sender] -= amount;
            otherBank.decreaseBal(msg.sender, amount);
        }
    }
}
```

Listing 3: Well-typed `withdraw` function.

This endorsement allows the desired functionality, but without further restriction, reopens the reentrancy bug we are trying to prevent. The endorsement does, however, allow us to precisely *define* reentrancy and identify critical sections of the code. This definition is instrumental in developing appropriate restrictions on `endorsepc` to regain security.

3.1 Defining Reentrancy

Existing definitions of reentrancy [6, 9] rely on contract boundaries. The second vulnerability in the distributed bank demonstrates the need for a more general definition. Suppose we fixed the same-contract reentrancy bug by switching the order of lines 8 and 9 in Listing 2. When the attacker $\mathcal{A}$ acquires control after begin sent money in `withdraw`, it can no longer improperly extract money from the *same* contract instance. However, the other instance of `DistributedBank` still has the old value of $\mathcal{A}$'s balance. This allows $\mathcal{A}$ to call `withdraw` there and extract its original deposit twice, once from each instance.

Intuitively, the two attacks are the same; an attacker withdrew funds from the bank while the bank was waiting for a response. In classic reentrancy definitions [6, 9], however, the attacker must call the same contract that passed it control. In this view, the second attack would not even be reentrant. Our insight is to instead define reentrancy with respect to integrity levels. If trustworthy code retains control over execution, correctness is up to the programmer. If untrustworthy code or data gains influence over control flow, unexpected calls may ensue, such as dangerous withdrawals. The result is a definition parameterized by a label ℓ.

Definition 1 (Reentrancy)**.** An execution is *reentrant with respect to a label* ℓ if, at some point, the stack contains instructions with integrity levels pc_1, pc_2, and pc_3 in that order such that $pc_1 \Rightarrow \ell$, $pc_2 \not\Rightarrow \ell$, and $pc_3 \Rightarrow \ell$.

This definition generalizes reentrancy to match the fine-grained security policies from Section 2. When the high-integrity region is exactly a single contract, as in Figure 1a, our definition coincides with the classic one. Definition 1 remains sensible in other cases, like our distributed bank example that fits Figure 1b.

Definition 1 has interesting ramifications. First, it decouples the definition of reentrancy from contracts. This allows the same ideas to apply to non-blockchain systems with interactions across trust boundaries, like calls between trusted hardware and an untrusted operating system or interacting javascript code from different sources on the same webpage. Second, a single execution can be reentrant from the perspective of one trust level, but not another. This discrepancy makes sense. If contract A trusts contract B but not vice-versa, the contracts have different security concerns when B calls A and then A calls back into B. From A's perspective, both contracts are trustworthy and this interaction is expected. From B's perspective, however, A could be attacking, so this execution is reentrant and potentially dangerous.

3.2 Defining Security

This definition of reentrancy allows us to define *reentrancy security*. We aim to ensure that programmers can ignore reentrancy when reasoning about their code. Reentrant calls may not be bug-free, but reentrancy cannot be the cause of the bugs. We express this goal by requiring that trustworthy code not exhibit behavior in reentrant executions that it cannot exhibit elsewhere.

To express this notion more formally, we note that program correctness is often expressible as a set of invariants. A property of the system is a *transaction invariant* if, whenever it is true before a transaction, it is also true after. A contract is reentrancy-secure if allowing reentrancy does not change which properties are invariants.

To correspond to our label-based definition of reentrancy, we parameterize the notion of invariant in terms of labels. An ℓ-integrity transaction invariant is one that depends only on data of integrity at least ℓ. An adversary not trusted by ℓ should be unable to affect an ℓ-integrity invariant.

Definition 2 (Reentrancy Security)**.** A program is *ℓ-reentrancy-secure* if, for all I, I is an ℓ-integrity transaction invariant whenever I is invariant for *non-reentrant* transactions.

To see how Definition 2 aligns with our intuition, we look to the distributed bank. There are two transaction invariants that a correct distributed bank should maintain: (i) `balances` contains the same value at both contracts, and (ii) the combined funds of both `DistributedBank` contracts is at least the sum of the values in any copy of the `balances` map. Because Listing 2 is insecure, it should not satisfy

Definition 2. Indeed, using Definition 1, these two properties hold at the beginning and end of each transaction for all non-reentrant executions. However, if an attacker overdraws their balance through either above-described reentrancy attack their balance will underflow, violating (*ii*).

3.3 Enforcing Security

We enforce reentrancy security by modifying the IFC typing rules to track the integrity entry points have granted. Within the scope of `endorsepc`, we lock the new, higher integrity level, and prevent the program from re-granting that integrity while it remains locked. In the modified bank code above, a reentrancy attack re-grants the bank's integrity to an attacker in the second call to `withdraw` while that integrity remains locked in the first.

To ensure compositional security and allow static checking, we track *locked integrity* as part of the type system. An `endorsepc` statement endorsing label pc_{from} to pc_{to} *respects a lock* if any integrity available at pc_{to} but not pc_{from} is unlocked. In other words, if pc_{to} can perform an operation and that operation's required integrity may already be locked, then the lower integrity pc_{from} must be sufficient. Formally if β is the locked integrity, we check that, for any ℓ, if $pc_{\text{to}} \Rightarrow \ell$ and $\beta \Rightarrow \ell$ then $pc_{\text{from}} \Rightarrow \ell$. More succinctly, we respect lock β if $pc_{\text{from}} \Rightarrow pc_{\text{to}} \vee \beta$, where $pc_{\text{to}} \vee \beta$ denotes the most trusted label that both pc_{to} and β act for.

We enforce locks across function calls with the same technique that standard IFC uses to maintain control-flow integrity. Function signatures specify not only the integrity required to execute the function, but also the locks that function respects. Function calls require the calling context to be at least as high integrity as the function—the standard IFC rule—and the function to respect all locks in place in the calling context.

Returning to Listing 3, the `endorsepc` statement in `withdraw` grants high integrity, so it only respects low locks. The type system then disallows calls to `withdraw` from contexts with the bank's integrity already locked, and only allows calls out of the `endorsepc` block to functions known to respect this lock. Reentrant calls to `withdraw` are therefore impossible.

4. OPEN-WORLD SECURITY

This purely type-based solution successfully prevents reentrancy attacks while allowing required entry points, but it is still extremely restrictive. For example, `withdraw` must ensure that the code invoked when the user receives money respects high locks. While this may be possible in some limited cases or in a system where every contract is written with these IFC types, it prevents most interesting interaction with legacy code or arbitrary unknown contracts. For example, the `call` instruction on line 8 of Listing 2 would not type check even if we included an `endorsepc` block.

We enable such interaction via dynamic locks, enforced at the same place as the static locks: `endorsepc` entry points. Before, `endorsepc` had no run-time effect, but it now checks dynamic locks. Though locks do add some performance cost, they increase flexibility. Code can statically respect a lock either by performing only operations that respect that lock or by converting it to a dynamic lock before executing operations that might violate it. We cannot assume legacy systems or unknown code will respect *any* locks, but dynamic locks allow secure interaction with them from high-integrity contexts.

For example, we can finish securing our distributed bank by including `lock(to_label(this)) {...}` around the `call` instruction to acquire a dynamic lock. This lock removes the need to assume that the unknown user does not attempt a reentrant call; it ensures that any such call will fail.

Developers can also assign a trusted label and IFC signature to legacy or off-chain systems. This trust allows simple and efficient interaction with the specified legacy code. Such trust is also risky, as the entire system may be insecure if the legacy code does not properly enforce the claimed guarantees.

5. IN PROGRESS

In addition to providing proofs of security, two key components remain incomplete.

Removing Unneeded Locks. Astute readers may notice that dynamic locks are unnecessary to secure the distributed bank. Moving the `call` instruction that sends money after both the balance update and contract-synchronization operations in `withdraw` is sufficient. Control flow would pass to untrusted code only after high-integrity code has completed all other operations. A reentrant call would then produce the same result as two sequential calls, meaning it cannot cause a bug, and is therefore safe. We are working to allow such secure calls without dynamic locks.

Implementation. We are designing a language and associated compiler that includes the features we have described. The dynamic locks poses two challenges. First, we need to record locks in a manner that spans multiple calls to a contract, but not multiple transactions. Using persistent storage for locks is extremely expensive on systems like Ethereum, but we do not need truly persistent storage. A session variable that remains in scope across multiple calls but resets each transaction would be ideal.

Second, because we are locking labels, we need to represent and compare them dynamically. Other systems implement dynamic label checking [7, 10], but they are not always efficient in space or computation. The high cost of blockchain computation makes it important to keep these checks fast. We are optimistic that blockchain applications will, in practice, use simple labels that are easy to represent and compare.

6. CONCLUSION

Securing smart contracts is a difficult but important challenge. Information flow control (IFC) provides a means to express and enforce powerful fine-grained, compositional security policies. We adapted these techniques to guarantee smart contract security at a fine-grained level. In the process we generalized the concept of reentrancy to describe a wider range of vulnerabilities. We described how to eliminate these vulnerabilities while retaining the assurance of IFC by restricting designated entry points. Finally, we expanded our approach to support an open world with legacy systems and contracts. This work forms a strong foundation for building a smart contract language with strong security guarantees.

Acknowledgments

We would like to thank our anonymous reviewers for their insightful and comments as well as Maximilian Algehed, Tom Magrino, and Drew Zagieboylo for help editing. Funding for this work was provided by a National Defense Science and Engineering Graduate Fellowship, NSF grants 1704615 and 1704788, and a gift from Ripple.

References

[1] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, Rain, D. Russi, S. Sezer, T. Zakian, and R. Zhou. Move: A language with programmable resources. https://developers.libra.org/docs/move-paper, Sept. 2019. Accessed February 2020.

[2] L. Breidenbach, P. Daian, A. Juels, and E. G. Sirer. An in-depth look at the parity multi-sig bug. http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/, 22 July 2017. Accessed February 2020.

[3] M. Coblenz. Obsidian: a safer blockchain programming language. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 97–99. IEEE, 2017.

[4] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. Collaborative verification of information flow for a high-assurance app store. In *21st ACM Conf. on Computer and Communications Security (CCS)*, pages 1092–1104, Nov. 2014.

[5] I. Grishchenko, M. Maffei, and C. Schneidewind. Foundations and tools for the static analysis of ethereum smart contracts. In *International Conference on Computer Aided Verification (CAV)*, pages 51–78. Springer, 2018.

[6] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM on Programming Languages*, 2(POPL):1–28, Dec. 2017.

[7] J. Liu, O. Arden, M. D. George, and A. C. Myers. Fabric: Building open distributed systems securely by construction. *J. Computer Security*, 25(4–5):319–321, May 2017.

[8] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *ACM Conf. on Computer and Communications Security (CCS)*, pages 254–269, 2016.

[9] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.

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

[11] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *16th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Oct. 1997.

[12] Parity Technologies. A postmortem on the parity multi-sig library self-destruct. https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/, 15 Nov. 2017. Accessed February 2020.

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

[14] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *IEEE Symp. on Security and Privacy*, pages 655–670, May 2014.

[15] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[16] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao. Safer smart contract programming with Scilla. *Proc. ACM on Programming Languages*, 3(OOPSLA):1–30, Oct. 2019.

[17] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *39th ACM Symp. on Principles of Programming Languages (POPL)*, pages 85–96, 2012.