Writing Safe Smart Contracts in Flint

Franklin Schrans
Imperial College London
United Kingdom
fr@nklinschrans.com

Susan Eisenbach
Imperial College London
United Kingdom
susan@ic.ac.uk

Sophia Drossopoulou
Imperial College London
United Kingdom
sd@ic.ac.uk

ABSTRACT

Blockchain-based platforms such as Ethereum support the execution of versatile decentralized applications, known as *smart contracts*. These typically hold and transfer digital currency (e.g., *Ether*) to other parties on the platform. Contracts have been subject to numerous attacks, losing hundreds of millions of dollars (in Ether).

We propose **Flint**, a new type-safe, capabilities-secure, contract-oriented programming language specifically designed for writing robust smart contracts. To help programmers reason about access control of functions, Flint programmers use *caller capabilities*. To prevent vulnerabilities relating to the unintentional loss of currency, transfers of assets in Flint are performed through safe atomic operations, inspired by linear type theory.

CCS CONCEPTS

• **Software and its engineering** → **Context specific languages**; Distributed programming languages; • **Computer systems organization** → *Peer-to-peer architectures*;

KEYWORDS

Smart Contracts, Ethereum, Programming Language

ACM Reference Format:
Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. 2018. Writing Safe Smart Contracts in Flint. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3191697.3213790

1 BACKGROUND

Smart contract development needs to ensure correctness of the contract's behavior before deployment, as contracts cannot be updated easily[1]. Correct behavior requires protection against unauthorized calls[2]. Contracts may remain active over a long period of time, thus reasoning needs to cover all possible states.

Programming languages have been developed for writing smart contracts on Ethereum. Solidity [1] is the most widely used of these languages. It uses static typing, and brings features specifically designed to write smart contracts, such as function *modifiers*[3] [2].

Smart contracts have been targeted by numerous **attacks** [3, 4, 7, 8], leading to double-spending, or unauthorized function calls [7, 8]. Analysis tools [9, 10] aim to detect vulnerabilities in smart contracts before their deployment.

Flint aims to make it easy to write *inherently safer* contracts, rather than analyze the contract after it has been written. On a similar vein, Viper [5] does not support Solidity modifiers, recursion, or infinite loops, while Bamboo [6] considers smart contracts as state machines. Flint proposes a different approach and is, we claim, a better fit for writing smart contracts.

2 FLINT

2.1 Language Design

Flint is a statically-typed programming language with novel contract-oriented features.

Caller capabilities require programmers to think about which Ethereum accounts have the right to call the contract's sensitive functions, which need to be declared in *caller capability blocks*. Capabilities are checked statically for internal function calls, and at runtime for calls originating from an Ethereum user or another smart contract.

Assets, such as Ether, are often at the center of smart contracts. Flint puts assets at the forefront through the special *Asset* trait. Assets in Flint cannot be accidentally created, duplicated, or destroyed, but they can be atomically split, merged, and transferred to other Asset variables. Asset types avoid conversions between numbers and Ether, and ensures that the contract's Ether is always consistent with its local state, preventing attacks such as TheDAO [3].

Figure 1: Flint Assets cannot be created, duplicated, or destroyed accidentally. They can be transferred atomically.

Restricting writes to state in functions helps programmers more easily reason about their contract. A function which writes to the contract's state is annotated with the `mutating` keyword. A non-mutating function cannot call a mutating function. This is important as smart contracts can be active over long periods of time and need to be reasoned about given any state.

[1] Updating a contract involves deploying a new version of the contract at a new Ethereum address, then manually transferring the state of the old contract—an expensive operation.
[2] A smart contract is more akin to a web service presenting API endpoints than a traditional computer program, which runs sequentially.
[3] These specify preconditions to protect against unauthorized calls.

<Programming'18> Companion, April 9–12, 2018, Nice, France

ACM ISBN 978-1-4503-5513-1/18/04.
https://doi.org/10.1145/3191697.3213790

2.2 Example Flint Contract

We look at a `FlightManager` contract, which an airline might create to allow users to book flights by sending Ether. If the airline needs to cancel a flight, an administrator should be able to call a function to refund all the passengers.

1. Declaring the Contract's State. In Flint, the contract's state is defined in isolation from its functions to help ensure no unnecessary state properties are declared. The `Address` type represents an Ethereum address (a user or another contract), and `Seat` is a struct we have defined.

```
contract FlightManager {
 var flightID: String
 var admin: Address
 var allocations: [Address: Seat]
 var ticketPrice = 1.12 // in Ether.

 var totalFunds: Ether
}
```

2. Declaring the Functions Anyone Can Call. Functions are declared in caller capability blocks. The function `buy` can be called by any user. The caller's address is bound to the `caller` local variable.

```
FlightManager :: caller <- (any) {
 @payable
 mutating public func buy(implicit value: Ether) {
  assert(value.getRawValue() == ticketPrice)
  let seat = findAvailableSeat()
  allocations[caller] = seat

  // Record the received Ether in the state.
  totalFunds.transfer(&ticketPrice)
 }
}
```

Similarly, we define `cancelFlight`, which refunds all the passengers, and can only be called by the flight administrator.

```
FlightManager :: (admin) {
 mutating public func cancelFlight() {
  for (passenger: allocations.keys) {
   refund(passenger)
  }
 }
}
```

3. Declaring the Functions Passengers Can Call. The `keys` property of `allocations` contains the addresses of all passengers. The `cancelBooking` function needs to be annotated **`mutating`** as it modifies the contract's state (`refund` is a mutating function).

```
FlightManager :: passenger <- (allocations.keys) {
 public func getSeatAllocation() -> Seat {
  return allocations[passenger]
 }

 mutating public func cancelBooking() {
  refund(passenger)
}
```

4. Refunding Users. The `Ether` type is a Flint Asset and therefore supports a set of safe atomic transfer operations. In line 3, we transfer `ticketPrice` amount from the `totalFunds` state property to the local variable `r`. We then transfer the contents of `r` to the Ethereum address `passenger`.

```
FlightManager :: (admin, allocations.keys) {
 mutating func refund(passenger: Address) {
  let r = Ether(from: &totalFunds, ticketPrice)
  allocations[passenger] = nil
  send(passenger, &r)
 }
}
```

2.3 The Flint Compiler

Our compiler `flintc` is under active development, and produces valid EVM bytecode. The open-source compiler and the Flint Language Guide are available on flintlang.org.

REFERENCES

[1] 2014. Solidity Documentation. http://solidity.readthedocs.io/en/latest/. (2014).
[2] 2014. Solidity Modifiers. http://solidity.readthedocs.io/en/develop/contracts.html#function-modifiers. (2014).
[3] 2016. Chasing the DAO Attacker's Wake. https://pdaian.com/blog/chasing-the-dao-attackers-wake/. (2016).
[4] 2016. King of the Ether Throne: A Post-Mortem investigation. https://www.kingoftheether.com/postmortem.html. (2016).
[5] 2016. The Viper programming language. https://github.com/ethereum/vyper. (2016).
[6] 2017. Bamboo: a language for morphing smart contracts. https://github.com/pirapira/bamboo. (2017).
[7] 2017. The Multi-sig Hack: A Postmortem. http://paritytech.io/the-multi-sig-hack-a-postmortem/. (2017).
[8] 2017. A Postmortem on the Parity Multi-Sig Library Self-Destruct. http://paritytech.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/. (2017).
[9] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, A Rastogi, T Sibut-Pinote, N Swamy, and S Zanella-Beguelin. 2016. Formal verification of smart contracts. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security-PLAS16*. 91–96.
[10] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269.