Writing Safe Smart Contracts in Flint

Extended Abstract

Franklin Schrans
Imperial College London
fs2014@ic.ac.uk

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

Sophia Drossopoulou
Imperial College London
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 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. To help programmers reason about access control of functions, Flint programmers use *caller capabilities*.

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 to make it easy to write *inherently safer* contracts, rather than analyse the contract after it has been written. On a similar vein, Viper [5] does not support Solidity modifiers, nor recursion and 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.

[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.

2 FLINT

2.1 Language design

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

Assets, such as Ether, are often at the center of smart contracts. Flint puts assets at the forefront through the special *Asset* trait. A restricted set of atomic operations can be performed on assets, ensuring a contract's state is always consistent. It is impossible to lose or create Ether in unprivileged code. This prevents attacks relating to double-spending and re-entrancy. The Asset trait contains one associated type, *RawType*, representing the backing representation (*Int* for *Ether*). The trait defines which operations each asset should implement: *transfer* to move contents between assets, *value* to retrieve the raw representation, and *init* to create a new asset from a raw value (a privileged operation).

Caller capabilities require programmers to think about who should be able to call the contract's sensitive functions. Capabilities are checked statically for internal calls (unlike Solidity modifiers), and at runtime for calls originating from external contracts.

Restricting writes to state in functions helps programmers reason. A function which writes to the contract's state is annotated with the `mutating` keyword.

2.2 Example Flint contract

This example defines a smart contract emulating a bank. Users can send and transfer currency. The `Address` type represents an Ethereum address (a user or another contract). The `Ether` type implements *Asset*: lines 18 and 23 atomically transfer the contents of `value` and `balances[account]` to another variable. We also demonstrate the use of caller capabilities (l. 8, 15), capability binding (l. 15), and **`mutating`** and **`@payable`** functions (which can receive currency from the Ethereum network).

```
contract Bank {
 var manager: Address
 var balances: [Address: Ether]
 var accounts: [Address]
}
// manager is used as a caller capability.
// Only manager can call numAccounts.
Bank :: (manager) {
 func numAccounts() -> Int {
   return accounts.count
 }
}
// Can be called by any registered account.
// The matching account is bound to account.
Bank :: account <- (accounts) {
 @payable mutating
 func deposit(implicit value: Ether) {
   balances[account].transfer(value)
   // value has been atomically set to 0.
   // no re-rentrancy, nor double-spending issues.
 }
 mutating func transfer(dest: Address) {
  balances[dest].transfer(balances[account])
  // balances[account] == 0
 }
}
```

2.3 `flintc`, the Flint Compiler

Our compiler `flintc` currently implements most of Flint's features, producing valid EVM bytecode. We plan to make the compiler open-source and accept contributions from the community.

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.