Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Flint

Writing Safe Smart Contracts in Flint

Franklin Schrans

June 2018

Supervisor:

Prof. Susan Eisenbach

Second Marker:

Prof. Sophia Drossopoulou

Abstract

Blockchain-based platforms such as Ethereum support the execution of versatile decentralised applications, known as smart contracts. These typically hold and transfer digital currency (e.g., Ether) to other parties on the platform. Smart contracts have, however, been subject to numerous attacks due to the unintentional introduction of bugs. In total, over a billion dollars worth of Ether has been stolen. As smart contracts cannot be updated after deployment, it is imperative to ensure their correctness during development. Current program analysers cannot accurately find all vulnerabilities in smart contracts, as the main programming language used to write smart contracts allows many unsafe patterns. For this reason, program analysers are often not part of the development cycle.

We propose Flint, a new statically-typed programming language specifically designed for writing robust smart contracts. Flint's features enforce the writing of safe and predictable code. To help programmers reason about access control of functions, we introduce *caller capabilities*. To prevent vulnerabilities relating to the unintentional loss of currency, Flint *Asset types* provide safe atomic operations, ensuring the state of contracts is always consistent. Writes to state are restricted, and simplify reasoning about smart contracts.

Acknowledgements

I would like to thank:

- Professor Susan Eisenbach and Professor Sophia Drossopoulou, for introducing me to Ethereum, for their amazing continuous support, and for the invaluable insight they have shared with me on designing a programming language and on writing academic papers,
- the Department of Computing, for funding my trip to the *2nd International Conference on the Art, Science, and Engineering of Programming*,
- Paul Liétar, for the great discussions about linear type systems, which led to the development of Flint's Asset types,
- Malina, Saurav, Daniel, and Amin, for being exceptional friends,
- and of course, my parents, who have always given me the opportunity to pursue my dreams.

Contents

Chapter 1

Introduction

The Ethereum [1] network supports decentralised execution of programs, known as *smart contracts*. A smart contract is similar to a stateful web service, but is not executed by computers controlled by a specific organisation. Instead, it is deployed to the nodes of Ethereum's open network, known as *miners*. The Ethereum Virtual Machine [2] (EVM) is a global virtual machine operated by miners supporting the execution of smart contracts. It provides a versatile instruction set allowing the creation of Turing-complete [3] programs.

Users can interact with a smart contract by calling the functions it exposes. Function calls are executed by miners, which maintain the state of each smart contract. Similarly to Bitcoin [4] users, Ethereum users and smart contracts can exchange a digital currency known as *Ether*. Miners are rewarded for processing transactions and function calls using this currency.

Smart contracts implement self-managed agreements enforced autonomously. The source code of a smart contract is publicly available, and cannot be changed after deployment. The execution of function calls cannot be tampered with. Individuals who interact with smart contracts trust the correct execution of the code rather than reprogrammable machines controlled by a single provenance. In recent years, smart contracts have been used to implement auctions, votes [5], and sub-currencies [6] for crowdfunding purposes. Implementing a vote using a smart contract allows voters to not have to trust an organisation to count the votes correctly—they would be counted by the smart contract.

Not being able to update a smart contract's code after deployment requires the finding of all bugs before. Otherwise, a smart contract might exhibit unintended behaviour. Recently, attackers have found vulnerabilities in smart contracts allowing to redirect Ether funds held by the smart contract to their personal Ethereum account. Attacks against TheDAO [7] and the Parity smart contracts [8, 9] have accumulated losses of over a billion dollars worth of Ether. The primary programming language used to write smart contracts, *Solidity* [5], is expressive and introduces features designed for smart contract programming. However, Solidity supports a variety of unsafe patterns [10] which makes it difficult for analysis tools [11, 12] and programmers to find all the vulnerabilities of a smart contract. Solidity has few built-in security mechanisms and does not require programmers to write safe smart contracts by default. Often, vulnerabilities are introduced because of simple programming mistakes, such as forgetting to write a keyword in a function signature. Others are harder to notice, such as implicit integer overflows, or discarding the return value of sensitive functions.

For traditional computer architectures, languages such as Java [13], Haskell [14], Swift [15], Rust [16], and Pony [17] leverage years of research in programming languages to prevent the writing of unsafe code and allowing efficient optimisations. In contrast, multiple programming

languages [5, 18, 19, 20, 21] for writing smart contracts, including Solidity, have attempted to mimic languages for traditional architectures such as JavaScript [22] and Python [23], without providing additional safety mechanisms for Ethereum's unique programming model.

This project introduces a intuitive statically-typed programming language, specifically designed for writing Ethereum smart contracts. By identifying challenges specific to developing smart contracts and learning from past vulnerabilities, we attempt to design language features which make it difficult to write unsafe code. In particular, we focus on four main points:

1. **Protecting against unauthorised function calls.** Smart contract often carry out sensitive operations which need to be protected against unauthorised calls. It is easy to forget to write assertions which check whether the caller of a function is authorised to perform the call, and such vulnerabilities have been exploited numerous times.
2. **Safe operations for handling Ether.** As many smart contracts handle Ether (such as crowdfunding smart contracts), we believe the programming language should provide safe operations to receive and manage Ether. Ether is usually represented as an integer rather than a dedicated type, allowing accidental conversions between numbers and Ether. This leads to inconsistent states, in which the actual Ether balance of the smart contract is not accurately recorded.
3. **Limiting writes to state.** Since the lifetime of a smart contract can span multiple years, it is important for programmers to be able to reason about functions given any state. A programming language could help achieve this goal by preventing spurious writes to state, and supporting the declaration of immutable variables.
4. **Full interoperability with Solidity.** As a large set of smart contracts and libraries are written in Solidity, we aim to make our programming language interoperable with this language. This would entail having an Application Binary Interface (ABI) which is compatible with Solidity's, thus allowing Solidity smart contracts to call functions on Flint smart contracts and vice-versa. In addition, Ethereum client applications which support the preparation of Solidity function calls would also support Flint smart contracts without modification.

1.1 Challenges

1. **Developing an entirely new programming language.** The development of a programming language involves significant design and implementation work. Determining which features should be included requires to understand the challenges specific to programming immutable decentralised applications. Furthermore, we wanted to make our language fit into the existing Ethereum ecosystem. To allow for existing Ethereum developers to easily transition to Flint, we needed to make our safety features approachable and easy to use. To maintain efficient interoperability between Flint and Solidity contracts, we adopted similar calling conventions.
2. **Implementing an extensible compiler.** Implementing a compiler for a language in development requires designing a code architecture allowing for frequent changes and extensions. We were developing for a new platform with limited tools for debugging, deploying, and testing EVM bytecode. We enabled Solidity tools to work with Flint programs by embedding Flint IR code in a Solidity file.
3. **Evaluating a language.** It is difficult to immediately assess the impact of our language on the Ethereum community. Nonetheless, we compare Flint programs and their equivalent

in current languages in terms of code quality, safety (by running program analysers), and performance (by comparing execution costs). We also received feedback from presenting our work at a conference, and from the Ethereum community.

1.2 Contributions

1. **The Flint Programming Language.** We present Flint, a new language specifically designed for writing robust smart contracts with the following main features.

   (a) **Caller Capabilities.** We introduce a capabilities-based system to protect functions against unauthorised calls. Ethereum users must have the appropriate *caller capability* to call functions. We leverage Ethereum's existing cryptographic schemes to use Ethereum user addresses to power the caller capabilities feature. Flint requires programmers to systematically think about which Ethereum users are allowed to call the smart contract's functions before defining them. The novel capabilities system statically verifies the validity of internal calls to avoid unintentional bugs and increase runtime performance. The design and implementation of caller capabilities is described in detail in chapter 4.

   (b) **Safe Asset transfer operations.** Flint supports special operations for handling *Assets* such as Ether in smart contracts. Transfer operations are performed atomically, and ensure that the state of contract is always consistent. In particular, Assets in Flint cannot be accidentally created, duplicated, or destroyed, but they can be atomically split, merged, and transferred to other Asset variables. Using Asset types avoids a class of vulnerabilities in which smart contracts' internal state does not accurately represent their true Ether balance. We describe the design and implementation of Flint Asset types in chapter 5.

   (c) **Immutability by default.** To aid with reasoning, functions in Flint cannot mutate the state of the smart contract by default. Modifying the state requires the function signature to include the `mutating` keyword. This avoids accidental writes to state, and allows users of the smart contract to easily notice which functions need to be read carefully. This feature is described in subsection 3.3.1.

   (d) **ABI Parity.** We use the same Application Binary Interface (ABI) as Solidity's. This allows Solidity contracts to call Flint functions, and vice-versa, and allows existing Ethereum client applications to interact with Flint smart contracts without additional modifications. Caller capabilities are checked at runtime for external calls. The compiler also embeds Flint's IR code into a Solidity file, allowing us to leverage debugging and testing tools for Solidity. We show how we achieved ABI parity in subsection 6.7.3.

2. **The Flint Compiler.** We implement a compiler for Flint which verifies the validity of Flint programs and produces correct EVM bytecode. The implementation is written in 25 000 lines of Swift [15] code, and is tested using our robust testing infrastructure. The extensible code architecture of the compiler allowed us to support newly designed language features.

3. **Evaluation.** We compare the implementation of selected smart contracts in Flint and other Solidity smart contracts. In our evaluation, we find that Flint programs are much safer with

low performance overheads, and sometimes significantly faster thanks to more compile-time guarantees.

1.3 Community Feedback

The Flint project was made open source on GitHub [24] in April 2018 under the MIT license. Since then, we have presented the language at the *2nd International Conference on the Art, Science, and Engineering of Programming* in Nice, France, for which we published a paper [25]. The feedback was very positive. Flint won the First Prize in the Undergraduate track of the ACM Student Research Competition, and was selected to participate in the ACM Grand Finals. We were also honoured to present Flint at the Imperial Blockchain Forum alongside blockchain experts.

Flint's safety focused features and ease of use were praised by the Ethereum Community, with articles describing Flint as being "on its way to filling a sorely needed gap in the developer tooling space." Our Medium article [26] accumulated over a thousand "claps", and among the 6700 smart contract programming GitHub projects, Flint [24] is among the thirty most popular, with over 140 "stars". We have also seen developers write articles about implementing their smart contracts in Flint [27].

Chapter 2

Background

In this chapter, we introduce concepts required for writing a programming language for Ethereum smart contracts. We present an overview of how Ethereum and its blockchain operate, introduce the Solidity programming language, study significant attacks which were conducted against smart contracts, and current approaches to prevent vulnerabilities.

2.1 Ethereum, a Smart Contracts Platform

Ethereum is a platform which supports the decentralised execution of programs, known as *smart contracts*. Smart contracts are executed by Ethereum's open network of nodes (*miners*), and are interacted with through function calls, similarly to traditional web services. In addition, users and smart contracts on Ethereum can hold and transfer a currency called *Ether*, which we describe in 2.1.1. Users perform Ether transfers and call functions by creating Ethereum *transactions*, as explained in 2.1.5, which are processed by all the miners in the network. The state of Ethereum records the processed transactions, the balances of each user account, and the state of each smart contract. State is maintained by each miner in the network using a blockchain data structure for synchronising changes. This is described in more detail in 2.1.6.

Once deployed, the source code of a smart contract is available for any user to read. Smart contracts run autonomously as no party can alter their behaviour by updating the code. This allows users of a smart contract to inspect and understand the behaviour of the code, while being guaranteed it will persistently behave the same way. A smart contract can thus be thought of as the implementation of an agreement which is enforced by an autonomous entity. Clients of smart contracts do not have to trust the computers of a single provenance to execute the smart contract correctly. This eliminates the need for users to trust an organisation and is beneficial for multiple applications, such as running auctions or votes [5], and implementing sub-currencies [6] for crowdfunding purposes.

For instance, organising a vote, in which users can vote for an outcome among others, can be organised using a smart contract. Voters would cast their vote by calling a function on the smart contract. Clients would be guaranteed votes are counted correctly, as the code of the smart contract can be read and verified, and the execution of the contract cannot be tampered with due to the decentralised nature of its execution. Other applications of smart contracts include crowdfunding, or creating sub-currencies.

Record	Account types	Description
`balance`	All	The number of Wei owned by the account.
`storageRoot`	Smart contracts	Keccak-256 hash of Merkle Patricia's tree root used to encode the contract's internal storage.
`codeHash`	Smart contracts	Keccak-256 hash of the account's bytecode.

Figure 2.1: Ethereum Account States (Omitting Certain Fields)

2.1.1 Currency and Gas

The Ethereum platform provides its own currency, *Ether*, which can be used to reflect the value of goods and services. There are a variety of denominations of Ether, the smallest one being a *Wei*. 1 Wei is 10^{-18} Ether. When referring to Ether in Ethereum transactions, the Wei denomination is used.

Users also use Ether to purchase *gas*, required to pay for computational costs when executing transactions (see subsection 2.1.5). The amount of gas required to execute a function depends on its computational cost.

2.1.2 Cryptography and Addresses

The Keccak-256 [28] cryptographic hashing algorithm is used throughout the Ethereum platform. Keccak-256 was a proposed implementation of the Secure Hash Algorithm 3 [29] (SHA-3), but was not the final implementation of SHA-3 (FIPS-202), which differs slightly.

Users and smart contracts are identified by their Ethereum address, which is a 160-bit integer. Addresses are usually represented in hexadecimal format.

2.1.3 Accounts and Account States

Information about accounts is stored by every miner in the Ethereum network. There are two types of accounts on Ethereum:

Externally owned accounts (EOAs). These are controlled by private keys. Creating an EOA involves generating a private key (a 256-bit integer) and computing the corresponding address by using a combination of the Elliptic Curve Digital Signature Algorithm [30] (ECSDA) and Keccak-256. Users create EOAs to send Ether to other users and call contract functions, by creating transactions (described in 2.1.5).

Smart contract accounts. These are controlled by their bytecode, which miners run when users call their functions, and accept calls from other accounts.

Figure 2.1 describes the contents of an account state. A contract's internal storage is maintained by miners using a Merkle Patricia tree [31] mapping Keccak-256 hashes to 256-bit integers.

Field name	Description
`to`	The recipient's address.
`value`	An optional Wei value to transfer to the recipient.
`data`	An optional input data payload for transactions targeting contracts.
`init`	An optional payload of bytecode, used to initialise contracts.
`gasPrice`	Number of Wei to be paid per unit of gas.
`gasLimit`	The maximum amount of gas which should be used to execute the transaction.
`v`, `r`, `s`	Values corresponding to the sender's signature.

Figure 2.2: Contents of an Ethereum Transaction (Omitting Certain Fields)

2.1.4 World State

The world state σ is the mapping between account *addresses* and *account states*. World state is maintained using a Merkle Patricia tree, which maps Keccak-256 addresses to the Recursive Length Prefix–encoded [32] value of an account state.

2.1.5 Transactions and Calls

Transactions. The Ethereum platform can be seen as a finite-state machine, where state transitions are performed through transactions. An Ethereum transaction T brings a state $\boldsymbol{\sigma}_t$ to a state $\boldsymbol{\sigma}_{t+1}$:

$$\boldsymbol{\sigma}_{t+1} \equiv \Upsilon(\boldsymbol{\sigma}_t, T)$$

where Υ is the Ethereum state transition function (we use the same notation as the Ethereum Yellow Paper [2]). Practically, it is a signed data packet originating from an EAO, which can update the state (Ether balance, or contract storage) of an account. Smart contracts cannot create transactions. Figure 2.2 describes the contents of an Ethereum transaction. Transactions are aggregated into blocks of Ethereum's blockchain after having been mined. Newly-created transactions are broadcasted to the rest of the network.

EAOs decide which *gas price*, i.e., the number of Wei they are paying per unit of gas, is attached to their transaction. Usually, a higher gas price increases the probability for the transaction to be chosen by a miner. A transaction involves executing a variety of instructions (more details in 2.1.7). The amount of gas which will be used is not always deterministic (if the contract uses loops or recursion, for example), and thus a *gas limit* is also specified in a transaction. The execution of a transaction is terminated with an exception if the gas limit is reached.

Miners record which transactions they have processed by inserting them into a Merkle Patrica tree. When a transaction is processed, the client receives a *transaction receipt*, which indicates whether the transaction was successful, how much gas was used, etc.

Figure 2.3 presents the lifecycle of an Ethereum transaction. Transactions are created and signed by EAOs using an Ethereum client, and sent to miners in the Ethereum network. Each miner selects which transaction to execute, and write state changes to their local blockchain, which is then propagated to the other miners in the network.

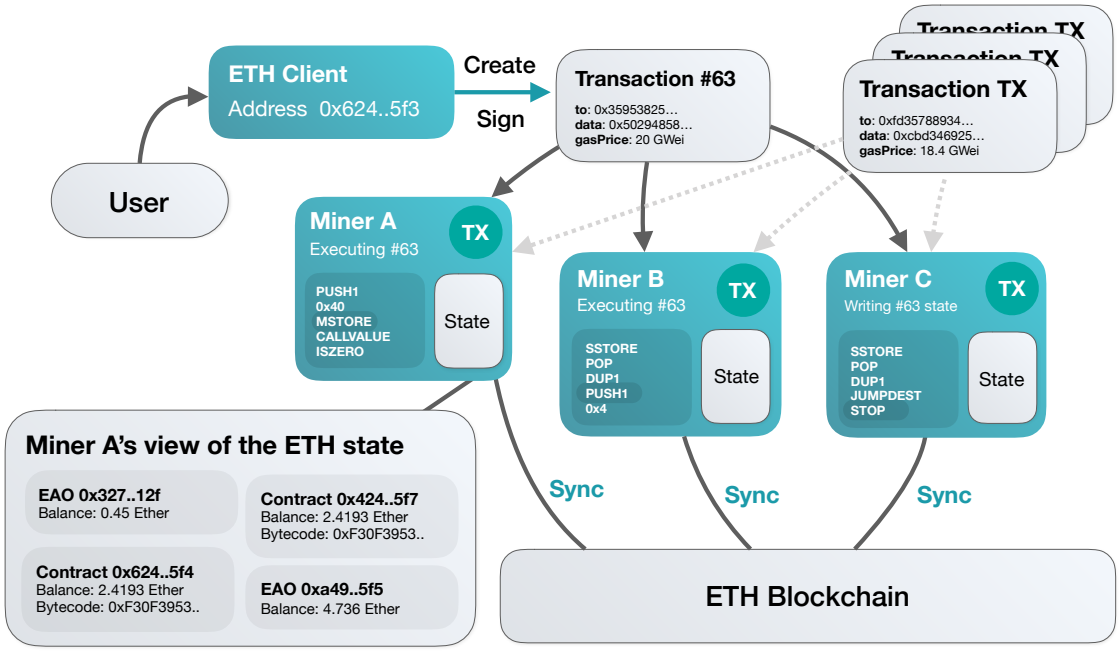


Figure 2.3: Lifecycle of an Ethereum Transaction

2.1.6 Blockchain

A blockchain is an append-only data structure composed of *blocks*, allowing miners to maintain a consistent view of the network's state. Cryptographic schemes ensure old blocks in a blockchain cannot be modified.

An Ethereum block is a data structure which is appended to the end of a blockchain, used by miners to share the result of the transactions they have processed. Miners have the freedom to select which transaction to process from their transaction pool (typically, the one with the highest gas price). A block contains a *block header* and the list of transactions which were processed by the miner (stored in a Merkle Patricia tree). Figure 2.4 describes the content of an Ethereum block header.

Miners each have a local version of the Ethereum blockchain. Miners append newly computed blocks to their local version of the blockchain, and broadcast their new blockchain to the rest of the network. Computing a block requires the miner to solve a cryptographic problem which limits the rate at which they can broadcast blocks. Miners thus continuously receive blockchains, and replace their local version with the longest blockchain they have received. As this process is the same for each miner, miners work off of the same blockchain.

Hard Forks

A fork in a blockchain refers to the existence of diverging paths in the sequence of blocks. A hard fork is an intentional fork which requires the network's nodes to adopt new backwards-incompatible rules for generating blocks. That is, blocks constructed by miners under the old rules are rejected by miners who follow the new rules. Hard forks usually occur when the Ethereum protocol is updated.

Field name	Description
`header`	The block header, used as an identifier, includes the following fields: `parentHash`: The Keccak-256 hash of the parent block header. Hashing previous blocks prevents attackers from altering the history of the blockchain. `beneficiary`: The address to which the reward gas is sent to. `stateRoot`: The Keccak-256 hash of the root node of the state trie after the transactions have been executed. `transactionsRoot`: The Keccak-256 hash of the root node of the transactions trie for the block.
`transactions`	A list of the transactions processed in the block, recorded from the transactions Merkle Patricia tree.

Figure 2.4: Contents of an Ethereum Block Header (Omitting Certain Fields)

2.1.7 Ethereum Virtual Machine and Bytecode

When processing transactions, miners execute bytecode. The execution model of Ethereum bytecode is specified through a stack-based virtual machine, the Ethereum Virtual Machine [2] (EVM). Like traditional computer architectures, EVM supports a variety of opcodes, allowing the development of versatile applications. EVM words are 256-bit long.

The Ethereum Virtual Machine uses four data stores.

1. **Stack.** The EVM stack has a maximum size of 1024. Accessing the stack is inexpensive, regarding gas costs.
2. **Memory.** A volatile byte array. Memory gets cleared after each transaction and is not written to the blockchain. All locations are initialised to zero. Its size grows at runtime if required, and its theoretical maximum size is 2^{256}. Accesses to memory are cheaper than storage accesses, and become more expensive when its size is increased.
3. **Storage.** A non-volatile word array. Storage persists across transactions and is written to the blockchain. All locations are initialised to zero. Its size is fixed at 2^{256} bytes. The cost of writing to storage is high, and is the same for all locations.
4. **Virtual ROM.** A read-only byte array containing the instructions of the smart contract.

Let $\boldsymbol{\mu_s}$ denote the EVM stack, and $\boldsymbol{\mu_s}[i]$ the ith element from the top of the stack. Let $\boldsymbol{\mu'_s}$ denote the resulting stack after executing an opcode. When we write $\boldsymbol{\mu'_s}[0] = c$, we mean $\boldsymbol{\mu'_s}[0] = c \wedge \forall i \in [1..k).\boldsymbol{\mu'_s}[i] = \boldsymbol{\mu_s}[i-1]$ where k denotes the length of $\boldsymbol{\mu_s}$'s stack. Let $\boldsymbol{\mu_{pc}}$ denote the value of the program counter.

Recall $\boldsymbol{\sigma}$ denotes the Ethereum global state. Let $\boldsymbol{\sigma'}$ denote the global state after executing an opcode. We denote I_a the address of the executed contract, and $\boldsymbol{\sigma}[I_a]$ its storage.

Figure 2.1 presents a selection of EVM opcodes. In addition to the traditional computer architecture instructions, EVM supports Ethereum-specific instructions, for example `SHA3` to perform a Keccak-256 hash, `BALANCE` to obtain the number of Wei attached to the

transaction, `CALL` to send a message call to another address, and `REVERT` to halt execution and revert the storage operations which occurred during the transaction.

Opcode	Description
Stop and Arithmetic Operations	
`STOP`	Halt execution.
`ADD`	$\boldsymbol{\mu_s'}[0] = \boldsymbol{\mu_s}[0] + \boldsymbol{\mu_s}[1]$
`MUL`	$\boldsymbol{\mu_s'}[0] = \boldsymbol{\mu_s}[0] * \boldsymbol{\mu_s}[1]$
...	
Comparison & Bitwise Logic Operations	
`EQ`	$\boldsymbol{\mu_s'}[0] = \begin{cases} 1 & \text{if } \boldsymbol{\mu_s}[0] = \boldsymbol{\mu_s}[1] \\ 0 & \text{otherwise} \end{cases}$
`GT`	$\boldsymbol{\mu_s'}[0] = \begin{cases} 1 & \text{if } \boldsymbol{\mu_s}[0] > \boldsymbol{\mu_s}[1] \\ 0 & \text{otherwise} \end{cases}$
`AND`	$\forall i \in [0..256).\boldsymbol{\mu_s'}[0]_i = \boldsymbol{\mu_s}[0]_i \wedge \boldsymbol{\mu_s}[1]_i$ where $\boldsymbol{\mu_s}[i]_j$ the jth bit of $\boldsymbol{\mu_s}[i]$.
...	
SHA3	
`SHA3`	$\boldsymbol{\mu_s'}[0]$ = Keccak-256 hash of the value in memory in the range $[\boldsymbol{\mu_s}[0]..\boldsymbol{\mu_s}[0] + \boldsymbol{\mu_s}[1])$
Environmental Information	
`ADDRESS`	$\boldsymbol{\mu_s'}[0]$ = address of the current executed account
`BALANCE`	$\boldsymbol{\mu_s'}[0]$ = balance in Wei of the account at address $\boldsymbol{\mu_s}[0]$
`CALLER`	$\boldsymbol{\mu_s'}[0]$ = address of the caller (account responsible for the current transaction)
`CALLVALUE`	$\boldsymbol{\mu_s'}[0]$ = number of Wei attached to this transaction
`GASPRICE`	$\boldsymbol{\mu_s'}[0]$ = gas price in the current environment
...	
Block Information	
`COINBASE`	$\boldsymbol{\mu_s'}[0]$ = address of the current block's beneficiary (gas recipient)
`TIMESTAMP`	$\boldsymbol{\mu_s'}[0]$ = current block's timestamp
`NUMBER`	$\boldsymbol{\mu_s'}[0]$ = current block's number
...	
Stack, Memory, Storage and Flow Operations	
`POP`	Remove items from the stack.
`MLOAD`	Load word from memory at $[\boldsymbol{\mu_s}[0]..\boldsymbol{\mu_s}[0] + 32)$.
`MSTORE`	Store word $\boldsymbol{\mu_s}[1]$ to memory at $[\boldsymbol{\mu_s}[0]..\boldsymbol{\mu_s}[0] + 32)$.

SLOAD	Load word from storage. $\boldsymbol{\mu'_s}[0] = \boldsymbol{\sigma}[I_a][\boldsymbol{\mu_s}[0]]$
SSTORE	Save word to storage. $\boldsymbol{\sigma'}[I_a][\boldsymbol{\mu_s}[0]] = \boldsymbol{\mu_s}[1]$
JUMP	Set the program counter to $\boldsymbol{\mu_s}[0]$.
PC	$\boldsymbol{\mu'_s}[0]$ = value of program counter prior to executing the opcode.
...	
Push Operations	
PUSH1	Push the 1 byte value at $\boldsymbol{\mu_{pc}} + 1$, extended with zeros to fit 32 bytes.
PUSH2	Push the 2 byte value from $[\boldsymbol{\mu_{pc}} + 1..\boldsymbol{\mu_{pc}} + 2]$, extended with zeros to fit 32 bytes.
...	
PUSH32	Push a full word value from $[\boldsymbol{\mu_{pc}} + 1..\boldsymbol{\mu_{pc}} + 32]$.
Logging Operations	
LOG0..LOG4	Append to log record with values (details omitted).
System Operations	
CALL	Message-call to another account (details omitted).
REVERT	Halt execution, reverting state changes.
SELFDESTRUCT	Halt execution and register account for later deletion. Transfers the account's balance to $\boldsymbol{\mu_s}[0]$.

Table 2.1: EVM Opcodes from the Yellow Paper [2]

Exceptions. Execution can be interrupted due to exceptions. These can occur for various reasons, including as the outcome of a REVERT instruction, executing an invalid instruction, a stack underflow, or a gas limit reached.

2.1.8 EVM Events

EVM supports special opcodes to log events triggered in a smart contract. Logs are sent back to the creator of the transaction, as part of the transaction receipt. JavaScript libraries such as web3.js [33] allow JavaScript applications to connect to the Ethereum blockchain and initiate transactions. web3.js also allows listening for EVM events returned as part of the transaction receipt and trigger a user-defined function when specific events occur.

2.1.9 Secret-Sharing Example

We now go through a practical example of how the Ethereum platform can be used. Consider three individuals, A, B, and C. A and B both have a secret to share to C. However, A is only willing to share its secret M_A if B will share its M_B secret, and vice-versa. To ensure both A and B have provided their secrets before delivering them to C, C sets up a smart

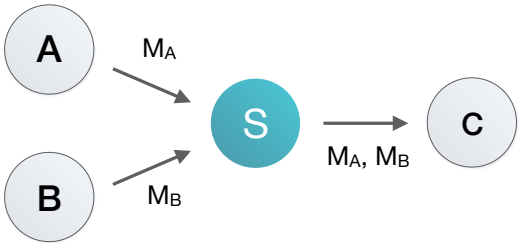


Figure 2.5: The Interactions Between the Ethereum Accounts A, B, C, and S. A sends its secret message M_A to S, B sends its secret message M_B to S, and S delivers both messages to C.

contract S which will receive A and B's secrets, and only deliver the results to C whenever both secrets are received.

1. **Joining the network.** A, B, and C launch their Ethereum clients, and each create an EOA. They are given their Ethereum address, and a private key.

2. **Creating a smart contract.** C writes a smart contract S using a high-level programming language, such as Solidity, and compiles the code to EVM bytecode. C then deploys the contract to Ethereum's blockchain by creating a transaction and signing it using its private key. C sets the `init` field (Figure 2.2) to the Keccak-256 hash of the bytecode. C estimates the amount of gas which will be needed for the transaction to be processed, and sets the `gas` field accordingly. C also creates a small JavaScript application using `web3js` to listen to EVM events triggered by S. When the "messages received" event is triggered, the application extracts from the payload the messages of A and B and sends them to C by email.

3. **Deploying the smart contract.** A miner of the Ethereum network selects C's transaction and waits for miners to process it. Eventually, miners run the transaction's bytecode, and create a new block. The block gets accepted to the blockchain and S is deployed at address A_S.

4. **A sends its secret to S.** A would now like to send its message to C. A creates a transaction, and since they are familiar with Solidity's Application Binary Interface (ABI), they know how to populate the `data` field of their transaction to call a specific function in S, passing in their secret message as a parameter. The transactions get broadcast to all the miners.

5. **S receives a call from A.** A's transaction gets mined: nodes execute S's bytecode and the function A's transaction specified gets called with the secret message as a parameter. Part of the internal state is modified, and the smart contract now acknowledges the receipt of the message by A, and stores the message. The execution of S stops, as it has not received B's message yet.

6. **B send its message.** B is also experienced with Solidity's ABI and send its message to S.

7. **S receives B's message.** Now that both messages have been received, S sends an EVM event, which triggers a callback function in C's JavaScript application. C receives the email with A and B's secrets.

8. *Optional: C deletes S.* C then wants to prevent S accepting any calls to its functions by deleting its associated account. C thus creates a new transaction calling a function in

S which executes the `SELFDESTRUCT` opcode. The contract is now marked as *destroyed*.

Note: All the data on the Ethereum blockchain is public and is not encrypted out-of-the-box. A and B secrets are readable by any miner. To protect their privacy, one could imagine they could securely send their secrets to S using a public/private key scheme.

An example of how S could be implemented in the Solidity programming language is shown in Listing 2.11.

2.2 Solidity

Solidity [5] is a high-level programming language for developing Ethereum smart contracts. It is statically-typed, imperative, and its syntax is inspired by JavaScript. A Solidity contract is similar to an object-oriented class, which can inherit functionality from other classes. Built-in types in Solidity include integers, addresses, fixed-size arrays, dynamic arrays, dictionaries (`mapping`). It is also possible for programmers to define their own types (`struct`). A contract declaration contains storage fields, event declarations, and function declarations.

2.2.1 Coin Example

Listing 2.1, from the Solidity documentation, contains the declaration of a sub-currency contract, `Coin`. `Coin` contains two fields, `minter` and `balances`, both declared public[1]. The fields constitute the contract's internal storage, and their values are persisted across transactions. The event `Sent` takes three arguments, and when called from a function, gets appended to the contract's log. The constructor on line 13 and other functions make use of the global `msg.sender` variable, which is bound to the Ethereum address of the caller of the function being executed. They are also marked `public` and mutate the contract state.

```
1  contract Coin {
2      // The keyword "public" makes those variables
3      // readable from outside.
4      address public minter;
5      mapping (address => uint) public balances;
6
7      // Events allow light clients to react on
8      // changes efficiently.
9      event Sent(address from, address to, uint amount);
10
11     // This is the constructor whose code is
12     // run only when the contract is created.
13     function Coin() public {
14         minter = msg.sender;
15     }
16
17     // Functions can return a value.
18     function getMinter() public returns(address) {
19         return minter;
20     }
21
22     function mint(address receiver, uint amount) public {
```

[1] All data on the blockchain can be read from any miner. When declaring a field `public`, Solidity synthesises getter functions for those fields.

```
23          if (msg.sender != minter) return;
24          balances[receiver] += amount;
25      }
26
27      function send(address receiver, uint amount) public {
28          if (balances[msg.sender] < amount) return;
29          balances[msg.sender] -= amount;
30          balances[receiver] += amount;
31          Sent(msg.sender, receiver, amount);
32      }
33  }
```

Listing 2.1: Solidity Sub-Currency Sample, from the Solidity Documentation[1]

2.2.2 Functions

Function Modifiers

Solidity *function modifiers* can be used to check preconditions before entering a function's body. Most of the time, they are used to insert a condition check before a function body. In Listing 2.2, we create the `onlyManager` modifier on line 4. The `require` statement enforces the condition `msg.sender == manager` to be true. If it is not, execution of the contract stops, and the sender receives an exception as part of the transaction receipt. The `_;` statement is used to indicate where the body of the target function should be inserted. A modifier can be used in the declaration signatures of multiple functions.

```
1   contract Bank {
2       address manager;
3
4       modifier onlyManager {
5         require(msg.sender == manager);
6         _;
7       }
8
9       function destroy() public onlyManager {
10        // Sends the SELFDESTRUCT opcode to the contract,
11        // i.e. deletes it.
12        selfdestruct(owner);
13      }
14
15      // Equivalent:
16      function destroyNoModifiers() public {
17        require(msg.sender == manager);
18        selfdestruct(owner);
19      }
20  }
```

Listing 2.2: An Example Use of Modifiers in Solidity. `destroy()`'s body can only be executed if the caller's address is the one stored in `manager`.

Modifiers may also take arguments and mutate the contract's state. Let's consider a more complex example in Listing 2.3. In this scenario, we would like a function to execute only

[1] `http://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html#subcurrency-example`

when all accounts a in some set of addresses S have called it. We use state variables to record which addresses have called each function.

`atLeastManagers` ensures that the target function's body will only be executed if all the addresses in the local `managers` array have called the target function. The `functionID` parameter is needed in the case where the modifier is being used for different functions. The `managerApprovals` state field maps a function identifier to another mapping which indicates which addresses have called the function.

In Solidity, the keys of a mapping cannot be retrieved. We use another mapping, `approvalKeysForFunctionID`, which stores the keys (addresses) in `managerApprovals`' domain mapping. When all the managers have called the target function, we need to reset the appropriate state fields (lines 20–25). We use the addresses from `approvalKeysForFunctionID` to reset the domain mapping in `managerApprovals`. We then clear `approvalKeysForFunctionID` at the appropriate entry using the `delete` operation.

```
1  contract Bank {
2      address managerA;
3      address managerB;
4      address[] importantManagers = [managerA, managerB];
5
6      mapping(string => mapping(address => bool)) managerApprovals;
7      mapping(string => address[]) approvalKeysForFunctionID;
8
9      modifier atLeastManagers(string functionID, address[] managers) {
10         managerApprovals[functionID][msg.sender] = true;
11         approvalKeysForFunctionID[functionID].push(msg.sender);
12
13         for (uint i = 0; i < managers.length; i++) {
14             if (!managerApprovals[functionID][managers[i]]) {
15                 revert();
16             }
17         }
18
19         // Reset helper state fields.
20         for (uint j = 0; j < approvalKeysForFunctionID[functionID].length; j++) {
21             address a = approvalKeysForFunctionID[functionID][j];
22             managerApprovals[functionID][a] = false;
23         }
24
25         delete approvalKeysForFunctionID[functionID];
26
27         // Execute function body.
28         _;
29     }
30
31     function destroy() atLeastManagers("destroy", importantManagers) {
32         selfdestruct(msg.sender);
33     }
34 }
```

Listing 2.3: An Example Use of Modifiers Taking Arguments, and Mutating the Contract's State in Solidity.

Attribute	Description
`pure`	The function cannot read or write to the contract's storage.
`constant`	The function cannot write to the contract's storage.
None	The function can read and write to the contract's storage.
`payable`	The function can read and write to the contract's storage, and accept a Wei value. The Wei value is implicitly added to the account's balance.

Figure 2.6: Function Attribute Modifiers

Function Return Values

Functions in Solidity return a specified number of values. For example, `function foo() returns(uint, address)` declares a function which returns two values. For functions which return values, a `return` statement does not explicitly have to appear in their body: the default values (0 for numbers and addresses) for the return types will be returned instead, as shown in listing 2.4. It is also possible to name return parameters, and use them as local variables. Their final value is returned from the call.

```
1  contract A {
2    function foo() returns(uint) {
3      // implicitly returns 0.
4    }
5
6    function bar() returns(uint, address) {
7      // return 0: type error
8      return (0, 0)
9    }
10
11   function bar() returns(uint a, uint b) {
12     a = 2
13     b = 10 + a
14   }
15 }
```

Listing 2.4: Returning Functions in Solidity

State Mutation and Visibility

Functions signatures can contain attributes to specify how they are allowed to access the contract's state, and their visibility by other contracts.

State mutation. Figure 2.6 presents the different function attributes which can be used in Solidity. A function may only have one mutation attribute.

```
1  contract A {
2    function foo() constant {}
3  }
```

Listing 2.5: A Constant Function in Solidity.

Modifier	Description
`external`	The function can only be called by external contracts through message calls (more details in section 2.2.3).
`public`	The function can be called internally or externally.
`private`	The function can only be called internally.

Figure 2.7: Function Mutation Modifiers (Omitting `internal`)

Visibility. Figure 2.7 presents the different visibility modifiers which can be used in Solidity. A function may only have one visibility modifier. Functions declared without visibility modifiers are implicitly `public`.

```
1 contract A {
2   function foo() constant public {}
3 }
```

Listing 2.6: A Constant Public Function in Solidity.

Note: declaring a field as `public` will synthesise a getter function for it.

Fallback Functions

A Solidity contract can define a *fallback function*, an unnamed function. When an account calls into a function of a contract for which the signature is not defined, the fallback function is called. Listing 2.7 presents a fallback function.

```
1  contract A {
2    // Fallback function.
3    function () public payable {
4      // body
5    }
6  }
7
8  contract B {
9    function foo() {
10     new A().call("foo") // calls A's fallback function, as "foo" is not defined
11     new A().call()      // calls A's fallback function
12   }
13 }
```

Listing 2.7: Fallback Function in Solidity.

2.2.3 Calls to Functions

Solidity functions can perform calls to other functions defined in the same contract or another contract.

Internal Calls

Calls to functions defined in the same contract are simply executing through `JUMP` instructions.

External Calls

A Solidity contract C can call functions from an external contract D, via a *message call*. The most common way to perform a message call is through expressions such as `d.foo()`, where `d` is bound to the address of a contract D, and `foo()` is declared in D. When executing a function in D, the caller's address (`msg.sender`) is bound to C's address. It is also possible to attach a Wei value to an external call, through an expression such as `d.foo.value(100)()` to send 100 Wei.

In Listing 2.8, contract `A` instantiates a contract of type `B` on line 2. This is allowed since the source code of `B` is included when compiling `A`. The field `B` is internally represented by an `address`.

```
1  contract A {
2      B public b = new B();
3
4      // Fallback function.
5      function A() public payable {}
6
7      function bGetCaller() public constant returns(address) {
8        // Returns A's address.
9        return b.getCaller();
10     }
11
12     function send(uint amount) public {
13         b.receive.value(amount)();
14     }
15 }
16
17 contract B {
18     function receive() public payable {}
19
20     function getCaller() public constant returns(address) {
21         return msg.sender;
22     }
23 }
```

Listing 2.8: External Function Calls in Solidity

Note: The special `this` variable is bound to the contract's address. Therefore, a call to `this.foo()` performs a *message call* to itself, and `msg.sender` in `foo` is now bound to the contract's address. This is different to performing the `foo()` call, which is internal and does not modify `msg.sender`. External calls to `this` are disallowed in the contract constructor's body.

Low-level Calls

External function calls as described above are translated to low-level function calls. The `call` function can be executed with an address as the receiver, and takes as the first parameter the first four bytes of the Keccak-256 hash of the target function's canonicalised function signature[1].

[1]The canonicalised identifier of a function's signature is of the form $f(t_1, t_2, t_3...)$ where f is the function's identifier, and $t_1, t_2, t_3, ...$ are the types of the function parameters. This is explained in more detail in 6.7.3.

Another variant of low-level function calls is supported, namely *delegate calls*. Delegate calls are similar to regular calls, except that any mutation of state in the target function is performed in the **caller**'s storage.

Both `call` and `delegatecall` return a boolean value denoting whether or not the call succeed with exceptions. Listing 2.9 demonstrates both types of calls. A call to `bCall` will set `B`'s `b` to 1, and a call to `bDelegateCall` will set `A`'s `a` to 2. Notice how delegate calls access a contract's storage through *offsets*, which is why the name of `B`'s first field is irrelevant.

```
1  contract A {
2      uint256 public a;
3      B public b = new B();
4
5      function bCall() public {
6        // Sets B's first field to 1.
7        b.setB(1);
8      }
9
10     function bDelegateCall() public {
11       // Sets A's first field to 2.
12         if (!b.delegatecall(bytes4(keccak256("setB(uint256)")), 2)) {
13             revert();
14         }
15     }
16 }
17
18 contract B {
19     uint256 public b;
20     function setB(uint256 value) public {
21         b = value;
22     }
23 }
```

Listing 2.9: Calls and Delegate Calls in Solidity

2.2.4 Interfaces

Solidity contracts can inherit from other contracts, with the semantics being similar to traditional object-oriented programming languages. *Interfaces* are contracts which can only declare function signatures, as shown in Listing 2.10.

```
1 interface I {
2   function foo() returns(uint256);
3 }
```

Listing 2.10: A Solidity Interface

2.2.5 Application Binary Interface (ABI)

The Solidity Application Binary Interface [34] specifies how to encode calls to functions to populate the `data` field of a transaction. At a high-level, the first four bytes of the `data` field represent the first four bytes of the Keccak-256 hash of the target function's signature. The rest of the `data` bytes represent the arguments' values given to the function. The ABI specification [34] describes the encoding for dynamic types (arrays and dictionaries) as well.

2.2.6 Contract S from subsection 2.1.9

Listing 2.1.9 provides an example implementation of the S contract from 2.1.9. From an Ethereum client, one can monitor an instance of S using its address, and listen for the `ReceivedSecrets` event.

```
contract Secrets {
    address addressA; string secretA; bool receivedA;
    address addressB; string secretB; bool receivedB;

    event ReceivedSecrets(string messageA, string messageB);

    function Secrets(address a, address b) public {
        addressA = a;
        addressB = b;
    }

    modifier onlyAorB {
        if (msg.sender != addressA && msg.sender != addressB) {
            revert();
        }
        _;
    }

    function receive(string secret) onlyAorB public {
        if (msg.sender == addressA) {
            secretA = secret;
            receivedA = true;
        } else if (msg.sender == addressB) {
            secretB = secret;
            receivedB = true;
        }

        if (receivedA && receivedB) {
            ReceivedSecrets(secretA, secretB);
        }
    }
}
```

Listing 2.11: An Example Implementation of the S Contract from 2.1.9 in Solidity.

2.3 Attacks Against Solidity Contracts

In this section, we explore different attacks which affected contracts written in Solidity.

2.3.1 Call Reentrancy: TheDAO Attack

TheDAO was a *decentralised autonomous organisation* (DAO) which operated on Ethereum's network. One of TheDAO's smart contracts contained a vulnerability which allowed an attacker to gain possession of approximately 3.6 million Ether, equivalent to approximately 880 million dollars at the time of writing. The incident resulted in a *hard fork*[1] of Ethereum's

[1] A hard fork is a process which results in a blockchain being split (see section 2.1.6). Concretely, this alters the state of the blockchain, breaking its *append-only* property. A hard fork was performed on Ethereum's

blockchain. Newspapers [35] and researchers [7], such as Philip Daian from Cornell University, attributed the vulnerability to poor design of Solidity: *"This was actually not a flaw or exploit in the DAO contract itself: technically the EVM was operating as intended, but Solidity was introducing security flaws into contracts that were not only missed by the community, but missed by the designers of the language themselves."*. Peter Daian mainly blamed [36] the unintuitive semantics of the `call` method: *"you cannot assume anything about the state of your contract after the external call is executed."*

TheDAO's attack was due to an exploit around *call reentrancy*. In EVM, when a function executes an external function call, it pauses execution until the call has completed. Call reentrancy occurs when the external function calls back into the original call. Control flow thus *reenters* the original call.

Listing 2.12 presents a contract, `Vulnerable`, which is vulnerable to call reentrancy. Users send Wei to this contract through the `deposit` function. The `withdraw` function retrieves the balance of the given account, transfers the Ether to the caller's Ethereum account, then sets the caller's balance to zero. On line 13, an external call is performed using the low-level `call` function, attaching a Wei value. As no function signature is given to `call`, the target's fallback function is called. The vulnerability is enacted when the target's fallback function calls back into `withdraw(address)`. Control flow executes lines 11–13 again, without having set the recipient's balance to 0. `Vulnerable` thus sends `balance` again, and the process repeats itself until the transaction's gas is exhausted. If enough gas is provided, the entire smart contract's Ether balance is sent to the attacker. The `balances` mapping remains unchanged except for the attacker's balance, which becomes 0. This leads to the state being inconsistent with the smart contract's actual Ether balance.

```
1  contract Vulnerable {
2      mapping(address => uint256) public balances;
3
4      function Vulnerable() public payable {}
5
6      function deposit(address recipient) public payable {
7          balances[recipient] += msg.value;
8      }
9
10     function withdraw(address recipient) public {
11         uint256 balance = balances[recipient];
12         recipient.call.value(balance)();
13         balances[recipient] = 0;
14     }
15 }
16
17 contract Attacker {
18     uint256 public total;
19     function () public payable {
20         total += msg.value;
21         msg.sender.call(bytes4(keccak256("withdraw(address)")), this);
22     }
23 }
```

Listing 2.12: Code Reentrancy Vulnerability in Solidity

TheDAO attack could have been prevented if line 12 and 13 were swapped. This would

blockchain for transactions involving the theft of money due to TheDAO attack to be "forgotten". This raised controversy as it violated the initial motivations behind running a decentralised blockchain.

have enforced the update to the contract's state to happen before performing the dangerous external call.

2.3.2 Function Visibility Modifiers Semantics: the First Parity Multi-sig Wallet Hack

The Parity Multi-signature (*multi-sig*) Wallet is a smart contract which allows accounts to control a common wallet. Due to a bug, an attacker managed to exploit the wallet to steal over 82 millions dollars worth of Ether [8].

The wallet ensures that, for example, all owners of a wallet need to accept a transaction before it can get executed (similarly to Listing 2.3). The library code Parity (the organisation behind the wallet) provided was written as a contract. Users would create their own wallet contract, and delegate all the functions to the library instance.

A simplified version is shown in Listing 2.13. `Wallet`'s constructor takes as arguments the owner of the wallet, and Parity's library contract address. It performs a `delegatecall` to `initWallet(address)`, which sets the owner in **Wallet**. `initWallet(address)` was intended to be called only once, in the constructor.

A transaction to `Wallet` with the `data` field containing `initWallet(address)`'s hashed signature would trigger the contract's function fallback to be called (`Wallet` doesn't define a function of that signature). Line 34 simply delegates the call to `WalletLibrary` with the `data` field from the transaction left intact. Because of the semantics of `delegatecall`, the function signature in `data` would in fact match the `initWallet` function in `WalletLibrary`, allowing the caller to set themeselves as the owner of the contract.

```
1  contract WalletLibrary {
2      address owner;
3
4      // Called by constructor. Visibility modifier defaults to 'public'.
5      function initWallet(address _owner) {
6          owner = _owner;
7          // More setup.
8      }
9
10     function () public payable {}
11
12     function withdraw(uint amount) external returns (bool success) {
13         if (msg.sender == owner) {
14             // body
15         }
16     }
17 }
18
19 contract Wallet {
20     address owner;
21     address walletLibrary;
22
23     function Wallet(address _owner, address _library) public {
24         walletLibrary = _library;
25         walletLibrary.delegatecall(bytes4(keccak256("initWallet(address)")), _owner);
26     }
27
```

```
28      function withdraw(uint amount) public returns (bool success) {
29          return walletLibrary.delegatecall(bytes4(keccak256("withdraw(uint)")), amount);
30      }
31
32      // Fallback function.
33      function () public payable {
34          walletLibrary.delegatecall(msg.data);
35      }
36  }
```

Listing 2.13: WalletLibrary and Wallet in Solidity

This attack could have been prevented by adding a modifier to `initWallet` ensuring the function is only called during its Initialisation phase.

2.3.3 Using a Contract as a Global Library: the Second Parity Multi-sig Wallet Hack

The same Parity Multi-sig contract was affected by another attack, which this time caused the loss of 260 million dollars worth of Ether. Consider Listing 2.14, the same `WalletLibrary` code from Listing 2.13, except that `initWallet` now has a modifier (`only_uninitialised`) which ensures the function can only be called if the contract has not be initialised yet. The addition of this modifier followed the attack mentioned previously. As a reminder, Parity runs a single instance of `WalletLibrary`, and each user deploys their own `Wallet` contract, which delegates its calls to the `WalletLibrary`.

The assumption from the Parity developers was that `WalletLibrary` would only be interacted with through delegate calls (mutating the caller's state rather than the library's state). However, when deploying `WalletLibrary`, `initWallet` was in fact *not* called. A user from the Internet then decided to call `initWallet`, which succeeded since the condition from `only_uninitialised` was met. At that point, the user was set as the owner of the *library* itself, and executed the `SELFDESTRUCT` opcode on it, which terminated the contract. As a consequence, all of the wallets delegating their calls to `WalletLibrary` were frozen: the instance of `WalletLibrary` they were delegating was destroyed.

```
1   contract WalletLibrary {
2       address owner;
3
4       // Can only be called once.
5       function initWallet(address _owner) only_uninitialised {
6           owner = _owner;
7           // More setup.
8       }
9
10      function () public payable {}
11
12      function withdraw(uint amount) external returns (bool success) {
13          if (msg.sender == owner) {
14              // body
15          }
16      }
17  }
```

Listing 2.14: WalletLibrary and Wallet in Solidity

The attack could have been prevented by using a Solidity `library` construct, which has been introduced following this attack. A Solidity library is a contract which is meant to be interacted with only using `delegatecall`s, and does not have any associated contract state. This would ensure that `initWallet` could not have been called directly on the library contract.

Alternatively, adding a constructor to `WalletLibrary` which called `initWallet` would have prevented the issue, as the `only_uninitialised` modifier would prevent further calls to it.

2.3.4 Unchecked Calls: King of the Ether Throne

The `KingOfTheEtherThrone` smart contract keeps track of a current `king`. An account needs to pay more Wei than the king paid in order to dethrone him. We consider a simplified version of the problem in Listing 2.15, in which the `king` and `claimPrice` keep track of the king and the number of Wei which allowed him to become king. When a king is dethroned, his `claimPrice` value is sent back to him. The `claim` function is called by an account which wishes to dethrone the king.

A vulnerability allows a king to be dethroned without receiving his `claimPrice` back. The issue relies on line 8: the Solidity `send` function returns a boolean indicating whether the transaction was successful. The transaction might fail due to an out-of-gas exception: the king's address may host a smart contract containing an expensive fallback function. When an out-of-gas exception occurs, any Wei value is returned to the caller, and thus the king's account is not credited back and the `King of the Ether Throne` keeps the Wei.

```
1  contract KingOfTheEtherThrone {
2      address public king;
3      uint256 public claimPrice;
4
5      function claim() payable public {
6          if (msg.value <= claimPrice) { revert(); }
7
8          king.send(claimPrice);
9          king = msg.sender;
10         claimPrice = msg.value;
11     }
12 }
```

Listing 2.15: WalletLibrary and Wallet in Solidity

The Solidity compiler now warns the developer when the return value of a sensitive function is not checked. A simple fix in this case is to check the return value of `send` and call the `REVERT` operation in the negative case.

2.3.5 Arithmetic Overflows: Proof of Weak Hands Coin

The `PoWH Coin` [37] smart contract implements a currency. 866 Ether (about $476K) was lost due to an arithmetic overflow following an addition. Arithmetic overflows are a common bug in smart contracts, as EVM's arithmetic operators have wrap-around semantics by default. The Solidity developers are looking into implementing operators which crash when overflows occur[1]. The third-party Solidity `SafeMath` library[2] implements methods to perform arithmetic

[1] https://github.com/ethereum/solidity/issues/796
[2] https://github.com/Openzeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol

operations safely.

```
contract PoWHCoin {
    function sell(uint256 amount) internal {
        var numEthers = getEtherForTokens(amount);
        // remove tokens
        totalSupply -= amount;

        // If balanceOfOld[msg.sender] < amount, the value will overflow and be close
        // to the largest 256-bit number, awarding a very large amount of tokens to
        // msg.sender.
        balanceOfOld[msg.sender] -= amount;

        // fix payouts and put the ethers in payout
        var payoutDiff = (int256) (earningsPerShare * amount + (numEthers * PRECISION));
        payouts[msg.sender] -= payoutDiff;
        totalPayouts -= payoutDiff;
    }
}
```

Listing 2.16: Arithmetic Overflows

2.3.6 Transaction-Ordering Dependencies

Miners have the freedom of choosing any transaction they wish from the Ethereum transaction pool. Consider two transactions T_i and T_j targeting the same smart contract. The order in which the transactions are processed can lead to different final storage states. In Listing 2.17, we consider a straightforward contract, `Puzzle`, in which an account is rewarded if it finds a secret string. If an account A calls `checkResult` and `receiveReward`, the correct account would be rewarded only if `checkResult` was executed before `receiveReward`. In the case where the transactions would be executed in the other order, the previous winner would have been rewarded. Transaction-ordering dependencies can be thought of as data-races between transactions on the blockchain.

```
contract Puzzle {
  address winner;
  string secret = "dQw4w9WgXcQ";
  uint256 reward = 100;

  function checkResult(string guess) {
    if (isCorrect(guess)) {
      winner = msg.sender;
    }
  }

  function receiveReward() {
    uint256 winnerReward = reward;
    reward = 0;
    winner.send(winnerReward);
  }

  function addReward() payable {
    reward += msg.value;
  }
}
```

Listing 2.17: Transaction-Ordering Dependency in Solidity

This issue can be mitigated by sending the reward directly in `checkResult`, ensuring the `winner` variable would be set correctly.

2.4 Current Attempts to Prevent Vulnerabilities

A vast amount of Ether has been lost due to the attacks mentioned previously. Previous work has been focused on finding these vulnerabilities early in the development cycle.

2.4.1 Analysis Tools for Solidity and EVM bytecode

There has been research around building analysis tools to detect vulnerabilities in smart contracts.

Oyente Dynamic Analysis. The Oyente [11] symbolic execution tool aims to find vulnerabilities in smart contracts by analysing their EVM bytecode. Symbolic execution is a technique which represents variable in a program by symbolic expressions. Paths are generated by accumulating constraints which the symbolic expressions form. From the type of bugs Oyente aims to find, such as reentrancy call issues, the authors of the tool claim they have found that 50% of smart contracts on Ethereum's network has at least one vulnerability. Oyente finds issues such as timestamp dependencies (when a smart contract relies on the time of the operating system of the miner which executes the contract), mishandled exceptions, and detects reentrancy calls.

Mythril Dynamic Analysis. The Mythril [12] tool is another popular analyser for Solidity contracts. It uses concolic analysis, which uses symbolic analysis to determine execution paths. Mythril tries to find a variety of vulnerabilities, such as timestamp dependencies, integer overflows, and reentrancy issues.

The Remix IDE. The online Remix [38] Integrated Development Environment (IDE) for Solidity, maintained by the Ethereum organisation, has a built-in code analyser which aims to find bugs. The analyser will, among other tasks, attempt to find reentrancy bugs and check for correct usage of low-level calls. In our testing, when running on the examples given in section 2.3, the Remix Analyser warned when low-level calls' return value was not checked, but was not able to find any reentrancy call issues.

Converting to F*. There has been research about formally verifying smart contracts using F* [39]. The approach involves translating (a supported subset of) Solidity code to F*, decompiling EVM bytecode to F*, then checking equivalence between the two translations. This aims to verify whether the produced bytecode matches the Solidity code's intended behaviour. The goal of the research was to show how F*'s type system is flexible enough to verify developer-encoded properties.

Scilla. Scilla [40] is a continuous passing style–based intermediate representation language for smart contracts. The goal is to convert Scilla code to the Coq theorem prover to produce formal proofs of certain properties of the contract.

2.4.2 Improving upon Solidity and EVM bytecode

Since the creation of Ethereum, multiple programming languages have attempted to improve the smart contract programming experience. Lisp Like Language [18] (LLL), developed by the Ethereum Foundation (the creators of Ethereum), is a low-level language which provides a Lisp-like syntax to program smart contracts. The project was abandoned in favour of higher level languages. Serpent [20] is a high-level programming language with a syntax similar to Python's. The project has been deprecated by the Ethereum Foundation due to its numerous security issues [41]. Solidity, which attempted to solve the issues these programming languages present, still allows writing unsafe code. Since Solidity, newer programming languages have been developed:

Vyper. Vyper [21] is an experimental programming language inspired by Python, aiming to provide developers with better security and more intuitive semantics than Solidity. However, even though Vyper is safer to use, it does not present unique features tailored for contract security.

Bamboo. The Bamboo [42] programming language allows reasoning about smart contracts as state machines. Developers define which functions can be called in each state, and the language provides constructs to specify changes of state explicitly. However, Bamboo does not present any additional features geared towards the safety of programs.

IULIA. IULIA [43] is a low-level programming language developed by the developers behind Solidity. The goal of IULIA is not to replace Solidity, but to be used as inline assembly within Solidity code. The Solidity compiler developers are also aiming to use IULIA as an Intermediate Representation (IR) of Solidity, to facilitate analysis and optimisations.

2.5 Remarks

The Ethereum platform allows the execution of versatile decentralised applications. The EVM allows smart contract programmers to focus on programming and ignore the complexities of blockchains and coordinating nodes in a very large network. Multiple programming languages have attempted to facilitate the creation of smart contracts, but many allow writing unsafe code. While program analysers for Solidity programs have attempted to find vulnerabilities, they cannot pinpoint all the issues in programs. Newer languages aim to bring the experience of developing traditional programs to Ethereum smart contracts. However, they are not designed to address the challenges programmers must face when developing for Ethereum, such as handling large quantities of money and not being updatable.

The attacks described in section 2.3 occur frequently, and can mostly be attributed to a mix of human error and unintuitive language semantics. For most of the vulnerabilities we have seen, we think a new language could help prevent them.

Call Reentrancy: TheDAO attack. This could have been avoided if Solidity provided safe library functions for Ether transfers. For example, the implementation of safe transfer operations would help ensure the contract's state is updated before Ether leaves the smart contract. For TheDAO, the runtime could execute lines 12 and 13 atomically in Listing 2.12.

Function visibility modifiers semantics: the first Parity Multi-sig wallet hack. Due to the programmer forgetting to specify the visibility of a function, Solidity chose to make the function 'public' by default. If Solidity did not expose functions to the public by default, this issue would not have occurred. Furthermore, a language could have helped ensure that the `initWallet` is only called during contract initialisation.

Using a contract as a global library: the second Parity Multi-sig wallet hack. The problem is that a user was able to modify the state of a smart contract which was only meant to be used as a library. This could be prevented by the use of a Solidity `library`, which is a contract which does not have any associated state and is meant to be interacted with through `delegatecall`s exclusively. The library contract would therefore not have an `owner` field in its state, and its functions could not be called directly.

Unchecked calls: King of the Ether Throne. This vulnerability can be prevented by the programming language disallowing the developer to discard the result of a function call by default. This would enforce them to check for the return value of the `send` call.

Arithmetic overflows: Proof of Weak Hands Coin. This attack was possible due to an integer value overflowing unexpectedly. We can prevent overflows by making arithmetic operators cause an exception whenever overflows occur. We could also introduce a type representing Ether which would provide safer transfer operations.

Transaction-ordering dependencies. This concurrency issue is difficult to fix at the programming language–level. Traditional locking mechanisms do not apply in the smart-contract-development environment, as the control flow of a contract cannot be paused and resumed arbitrarily. Analysis tools could help find data-race issues, however.

Chapter 3

The Flint Programming Language

Ensuring a smart contract behaves correctly before deploying it is crucial, as Ethereum does not allow updating them. Smart contracts handling millions of dollars worth of Ether have been exploited, resulting in the loss of all of their funds. For TheDAO attack (see subsection 2.3.1), the history of the blockchain was rewritten to revert the malicious transactions, violating Ethereum's core decentralisation principle.

For traditional computer architectures, languages such as Java [13], Rust [16], and Pony [17] have been designed to prevent writing unsafe code and introduced efficient runtime optimisations for the platform they target. For instance, Java prevents direct access to memory, Rust uses *ownership types* to efficiently free memory, and Pony uses *reference capabilities* to prevent data races in concurrent programs. In contrast, even though the Ethereum platform requires smart contract programmers to ensure the correct behaviour of their program before deployment, it has not seen a language designed with safety in mind. Solidity and others do not tailor for Ethereum's unique programming model and instead, mimic existing popular languages like JavaScript, Python, or C, without providing additional safety mechanisms.

We present Flint, a new programming language built for easily writing safe Ethereum smart contracts. Flint is approachable to existing and new Ethereum developers, and presents a variety of security features. The language requires programmers to protect their functions from unauthorised accesses by using *caller capabilities*. Functions in Flint cannot mutate the contract's state by default, hence simplifying reasoning. The standard library offers *Asset types*, which provide safe atomic operations to handle currency, ensuring the state of smart contracts is always consistent.

The remainder of this chapter presents an overview of Flint. Chapters 4 and 5 detail the design and implementation of caller capabilities and Asset types. Chapter 7 presents future work for features which we have designed but not fully implemented yet, such as support for type states, bounded loops, and the Flint Package Manager.

3.1 Programming in Flint

A smart contract's state is represented by its fields, or *state properties*. Its behaviour is characterised by its functions, which can mutate the contract's state. *Public* functions can be called by external Ethereum users.

Flint's syntax is focused on allowing programmers to write and reason about smart contracts easily. Providing an intuitive and familiar syntax is essential for programmers to express their smart contract naturally. As the source code of a smart contract is publicly available, it should be easily understandable by its readers. The syntax is inspired by the Swift Programming Language's [15] syntax.

When developing Flint, we focused on the novel features aimed at smart contract security. For this reason, some features which developers might expect from a programming language and its compiler such as recursive data types or type inference have not been implemented in the compiler. The code listings containing Flint code can be compiled using the latest version of the Flint compiler, unless stated otherwise.

3.1.1 Declaring Contracts

The contents of this subsection are partly taken from Writing Safe Smart Contracts in Flint [25], a paper we have written for the <Programming> 2018 conference.

When declaring a Flint smart contract, state properties are declared in isolation from the functions. Programmers can ensure that no unnecessary state properties are declared. This is important as state properties are stored in the smart contract's persistent memory (*storage*), which have high access costs.

Functions in Flint are not declared at the top level of a contract declaration. They must be enclosed in a *behaviour declaration block*, which protects its functions from unauthorised calls using caller capabilities. This forces programmers to first think about which parties should be able to call their function before defining it. A user must have the caller capability associated with the behaviour declaration block to call the block's functions. The caller capabilities feature is described in detail in Chapter 4.

We look at a `FlightManager` contract, which an airline might create to allow users to book flights using Ether. In the event the airline needs to cancel a flight, an administrator should be able to call a function to refund all the passengers. Since this contract handles currency, we use the safe transfer operations provided by the Wei Asset type, which represents the smallest denomination of Ether. For more details about Wei types and other Asset types, see chapter 5.

When declaring the contract, we observe how Flint's syntax requires programmers to write their smart contact in a specific sequence of steps.

1. Declaring the contract's state. The `Address` type represents an Ethereum address (a user or another contract), `Wei` an Ethereum Wei (the smallest denomination of Ether) and `Seat` a struct which we will define in the next section. We assign *default values* to the `allocations` and `amountPaid` state properties.

```
1 contract FlightManager {
2   let flightID: String
```

```
  let admin: Address
  var ticketPrice: Int
  var allocations: [Address: Seat] = [:]
  var amountPaid: [Address: Wei] = [:] // Records how much each passenger has paid.
}
```

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. The initialiser can only be called when the contract is created. Rules for Initialisation are explained in 3.1.3. `buy` is public and can be called from outside the contract. `allocateSeat` can only be called within the contract. `buy` is marked `@payable` as it the caller to attach Ether when calling the function. The amount of Ether is part of the Ethereum transaction and is not a function argument passed when calling `buy`. Flint thus binds the value to the `value` parameter, which is marked `implicit`.

```
FlightManager :: caller <- (any) {
  public init(flightID: String, admin: Address, ticketPrice: Int) {
    self.flightID = flightID
    self.admin = admin
    self.ticketPrice = ticketPrice
  }

  @payable
  mutating public func buy(implicit value: Wei) {
    assert(value.getRawValue() == ticketPrice)
    let seat: Seat = allocateSeat()
    allocations[caller] = seat

    // Record the received Ether in the contract's state.
    // 'value' is passed by reference, and its contents are set to 0 after the transfer.
    amountPaid[caller].transfer(&value)
  }

  mutating func allocateSeat() -> Seat {} // Definition omitted.
}
```

Similarly, we define functions which can only be called by the administrator, i.e., `setTicketPrice`, and `cancelFlight`, which refunds all the passengers. When referring to state properties, the `self.` prefix can be used. When it is unambiguous whether an identifier refers to a state property or a local variable, the `self.` prefix can be omitted.

```
FlightManager :: (admin) {
  mutating public func setTicketPrice(ticketPrice: Int) {
    self.ticketPrice = ticketPrice
  }

  mutating public func cancelFlight() {
    for passenger in 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).

```
39 FlightManager :: passenger <- (allocations.keys) {
40   public func getSeatAllocation() -> Int {
41     return allocations[passenger].getSeatNumber()
42   }
43
44   mutating public func cancelBooking() {
45     refund(passenger)
46   }
47 }
```

4. Refunding users. The `Wei` type is a Flint Asset and therefore supports a set of safe atomic transfer operations. These are described in more detail in chapter 5. On line 51, we read the state to determine how much Wei the passenger has paid, and transfers that amounting to a new local variable, `refund`. `amountPaid[passenger]` in the contract's state is cleared as part of the same *atomic* operation. We then transfer the contents of `refund` to the Ethereum address `passenger`. The value of `refund` is 0 after `send` is called.

```
48 FlightManager :: (admin, allocations.keys) {
49   mutating func refund(passenger: Address) {
50     // Transfer Wei from the state to a local var to reflect the contract losing Ether.
51     let refund = Wei(&amountPaid[passenger])
52     allocations[passenger] = nil
53     send(passenger, &refund)
54   }
55 }
```

The full example is available on Flint's GitHub repository [24]. We have not implemented loops yet in the compiler yet, as they require design work to ensure Flint code can still be formally verified easily. We propose a solution in subsection 7.1.2. We also do not support accessing the `keys` properties of dictionaries yet, due to performance concerns.

3.1.2 Declaring Structs

Programmers can declare structs in Flint to group state and functions. Struct functions are not protected by caller capabilities as they can only be called by (protected) contract functions, and are required to be annotated `mutating` if they mutate the struct's state.

We define the `Seat` struct:

```
1  struct Seat {
2    var number: Int
3
4    init(number: Int) {
5      self.number = number
6    }
7
8    func getNumber() -> Int {
9      return number
10   }
11 }
```

Struct values can be declared as state properties or local variables, and are initialised through their initialiser (e.g., `let seat = Seat(8)`). When stored as a state property, the struct's data resides in EVM storage. When stored as a local variable, it resides in EVM memory, and a pointer is allocated on the EVM stack.

Passing Structs as Function Arguments

When struct values are passed as function arguments, they are passed *by value* by default. That is, on function entry, a struct is copied in storage or memory (depending on whether it is a state property or a local variable) before executing the function's body. It is also possible to pass struct values *by reference*, using the `inout` keyword. The struct is then treated as an implicit reference to the value in the caller. We have implemented support for passing structs by reference in the compiler (see subsection 6.8.1), and have not implemented pass by value yet. Structs which contain properties of the same type, `recursive` structs, are not supported yet.

Listing 3.1 demonstrates the semantic differences between pass by value and pass by reference.

```
1  func foo() {
2    let s = S(8)
3    byValue(s) // s.x == 8
4    byReference(s) // s.x == 10
5  }
6
7  func byValue(s: Seat) {
8    s.x = 10
9  }
10
11 func byValue(s: inout Seat) { // s is an implicit reference to 's', from line 2.
12   s.x = 10
13 }
```

Listing 3.1: Pass by Value and Pass by Reference

3.1.3 Initialisation

Each smart contract and struct must define a public initialiser. All of the state properties must be initialised before the initialiser returns. State properties can be declared with a default value, e.g., for `allocations` and `amountPaid`, which are assigned dictionary empty literals (`[:]`), or in the initialiser, for `flightID`, `admin`, and `ticketPrice`. `let` constants must be assigned exactly once. Functions cannot be called in initialisers, as calling them might result in accessing uninitialised state properties.

Property: State Property Initialisation. Each state property v in a type T (contract or struct) must be initialised before the initialiser returns.

$$\text{IsInitialised(T)} \triangleq \forall \text{v} \in \text{StateProperties(T)} \ . \ \text{IsInitialised(v)}$$

where IsInitialised(v) indicates whether the property v has been assigned a default value when declared, or has been assigned in the initialiser of the type T.

For structs, if all state properties are assigned a default value at the declaration site, Flint synthesises an empty initialiser.

3.2 Type System

Flint is a statically-typed language with a simple type system, without support for subtyping. We are planning to implement type inference in the compiler. We are also planning to support Rust-style traits (or Java-style interfaces) with default implementation of functions as a mechanism to share code between structs and contracts.

Flint supports basic types (Figure 6.3) and dynamic types (Figure 3.2). Dynamic types can be passed by value or by reference.

Type	Description
`Address`	160-bit Ethereum address
`Int`	256-bit unsigned integer
`Bool`	Boolean value
`String`	String value
`Void`	Void value

Figure 3.1: Flint Basic Types

Type	Description
Array	Dynamically-sized array. `[T]` refers to an array of element type `T`.
Fixed-size Array	Fixed-size memory block containing elements of the same type. `T[n]` refers to an array of size `n`, of element type `T`.
Dictionary	Dynamically-size mappings from one key type to a value type. `[K: V]` is a dictionary of key type `K` and value type `V`.
Structs	Struct values, including `Wei`, are considered to be of dynamic type.

Figure 3.2: Flint Dynamic Types

3.2.1 Events

Type	Description
`Event<T...>`	An Ethereum event to notify clients listening to a transaction. Takes an arbitrary number of type arguments.

Figure 3.3: Flint Event Type

3.3 Mutation and Constants

Smart contracts can remain in activity for a large number of years, during which a large number of state mutations can occur. To aid with reasoning, Flint functions cannot mutate smart contracts' state by default. This helps avoid accidental state mutations when writing

the code, and allows readers to easily draw their attention to the mutating functions of the smart contract.

Let-constants ensure state properties and local variables are not accidentally mutated.

3.3.1 Mutating Functions

The Flint compiler requires functions which mutate the state of a function to be annotated with the `mutating` keyword. Functions which make calls to mutating functions are also considered to be mutating.

Property: Mutating Functions. A function f has to be declared `mutating` if it assigns a value to a state property of the type T (contract or struct) it is declared in, or if it calls a `mutating` function g.

$$\text{IsMutating(f, T)} \triangleq \forall s \in \text{Body(f)}\ (\text{IsAssignmentToState(s, T)}) \vee \exists c \in \text{FunctionCalls(s)}\ \begin{pmatrix} \neg\text{IsDeclaredLocally(Receiver(c))} \wedge \\ \text{IsMutating(MatchingDecl(c), ReceiverType(c))} \end{pmatrix}$$

where
IsAssignmentToState(s, T) indicates whether the statement s assigns to a property of T,
IsDeclaredLocally(v) indicates whether v is declared as a local variable,
FunctionCalls(s) is the set of function calls in statement s,
Receiver(c) is the receiver of the function call c,
ReceiverType(c) is the type of the receiver of the function call c,
MatchingDecl(c) is the matching declaration for a function call c.

In addition, functions which do not mutate state but are marked `mutating` trigger a warning.

In Listing 3.2, the function `increment()` mutates the state of the smart contract, but is not declared `mutating`. This triggers an error, shown in Listing 3.3. `getValue()` is marked `mutating` even though it does not mutate the state of the contract, which triggers a warning.

```
1  contract Counter {
2    var value: Int = 0
3  }
4
5  Counter :: (any) {
6    // This is a mutating function.
7    mutating public func set(value: Int) {
8      self.value = value
9    }
10
11   func increment() {
12     value += 1 // We are mutating state, as 'value' refers to a state property.
13   }
14
15   // This function is declared mutating but does not mutate any state.
16   mutating func getValue() -> Int {
17     return value
18   }
19
```

```
20    // Initialiser omitted.
21  }
```

Listing 3.2: Mutating Functions

The compiler produces the following output:

```
Warning in counter.flint:
Function does not have to be declared mutating: none of its statements are mutating at line
     16, column 3:
  mutating public func getValue() -> Int {
  ^^^^^^^^
Error in counter.flint:
Use of mutating statement in a nonmutating function at line 12, column 5:
    value += 1
    ^^^^^
Failed to compile.
```

Listing 3.3: Compiler Output for Invalid Mutation Declarations, for the code in Listing 3.2.

The full example is available at `https://github.com/franklinsch/flint/blob/master/examples/valid/counter.flint`.

3.3.2 Let-constants

The `var` keyword can be used to declare state properties or local variables which can be mutated. The `let` keyword declares a constant, as shown in Listing 3.4. `airline` is not initialised at the declaration site, thus needs to be assigned in the initialiser. It cannot be reassigned on line 14. `maxPassengers` cannot be assigned a value in the initialiser, as a value has already been assigned to it on line 3.

```
1   contract Flight {
2     let airline: String
3     let maxPassengers: Int = 250
4     var numPassengers: Int = 0
5   }
6
7   Flight :: (any) {
8     public init(airline: String) {
9       self.airline = airline
10      self.maxPassengers = 10 // Invalid as 'maxPassengers' is assigned on line 3.
11    }
12
13    mutating func setAirline(name: String) {
14      self.airline = name // Invalid as 'airline' is a let-constant.
15
16      let x: Int = 1
17      x = 2 // Invalid as 'x' is a let-constant.
18    }
19  }
```

Listing 3.4: Let-constants

The compiler produces the following output:

```
Error in test.flint:
Cannot reassign to value: 'maxPassengers' is a 'let' constant at line 10, column 10:
```

```
    self.maxPassengers = 10
         ^^^^^^^^^^^^^
  Note in test.flint:
  'maxPassengers' is declared here at line 3, column 3:
    let maxPassengers: Int = 250
    ^^^^^^^^^^^^^^^^^^^^^^
Error in test.flint:
Cannot reassign to value: 'airline' is a 'let' constant at line 14, column 10:
    self.airline = name // error
         ^^^^^^^
  Note in test.flint:
  'airline' is declared here at line 2, column 3:
    let airline: String
    ^^^^^^^^^^^^^^^^^^^
Error in test.flint:
Cannot reassign to value: 'x' is a 'let' constant at line 17, column 5:
    x = 2
    ^
  Note in test.flint:
  'x' is declared here at line 16, column 5:
      let x: Int = 1
      ^^^^^^^^^^
Failed to compile.
```

Listing 3.5: Compiler Output for Invalid Reassignments to Let-constants, for the Code in Listing 3.4.

3.4 Standard Library

3.4.1 Global Functions and Structs

The Flint Standard Library is available under the `stdlib/` directory of the Flint GitHub repository [24].

The standard library defines the `Wei` type. More information about its safe transfer operations is available in chapter 5.

We also define a set of global functions, shown in Figure 3.4. Global functions are defined in the special `Flint$Global` struct in `stdlib/Global.flint` and are imported globally by the compiler.

Function	Description
`send(address: Address, value: inout Wei)`	Sends `value` Wei to the Ethereum address `address`, and clears the contents of `value`.
`fatalError()`	Terminates the transactions with an exception, and revert any state changes.
`assert(condition: Bool)`	Ensures `condition` holds, cause a `fatalError()`.

Figure 3.4: Flint Global Functions

3.4.2 Safe Arithmetic Operators

Safe arithmetic operators are also provided. The `+`, `-`, and `*` operators throw an exception and abort execution of the smart contract when an overflow occurs. The `/` operator implements integer division. No underflows can occur as we do not support floating point types yet. The performance overhead of our safe operators are low, as described in our evaluation (see chapter 8).

Property: Safe Arithmetics. Let $\mathbb{Z}/2^{256}$ be the set of integers between 0 and $2^{256}-1$. Let $\underline{+}$, $\underline{-}$, $\underline{*}$, $\underline{/}$ denote the arithmetic operators of Flint, $+, -, *, /$ refer to the mathematical operators, and $\rightsquigarrow$ denotes the evaluation of an expression. If a computation does not follow the following rules (e.g., the evaluation causes an overflow), an exception is thrown and the Ethereum transaction is aborted with an exception.

$$\begin{aligned}
\forall a, b, c \in \mathbb{Z}/2^{256}\ a \underline{+} b \rightsquigarrow c &\implies a + b = c \\
\forall a, b, c \in \mathbb{Z}/2^{256}\ a \underline{-} b \rightsquigarrow c &\implies a - b = c \\
\forall a, b, c \in \mathbb{Z}/2^{256}\ a \underline{*} b \rightsquigarrow c &\implies a * b = c \\
\forall a, b, c \in \mathbb{Z}/2^{256}\ a \underline{/} b \rightsquigarrow c &\implies a / b = c
\end{aligned}$$

In rare cases, allowing overflows is the programmer's intended behaviour. Flint also supports overflowing operators, `&+`, `&-`, and `&*`, which will not crash on overflows.

3.4.3 Payable Functions

When a user creates a transaction to call a function, they can attach Ether to send to the contract. Functions which expect Ether to be attached when called must be annotated with the `@payable` annotation. When adding the annotation, a parameter marked `implicit` of type `Wei` must be declared. `implicit` parameters are a mechanism to expose information from the Ethereum transaction to the developer of the smart contract, without using globally accessible variables defined by the language, such as `msg.value` in Solidity. This mechanism allows developers to name `implicit` variables themselves, and do not need to remember the name of a global variable.

In Listing 3.6, the number of Wei attached to the Ethereum transaction performing the `receiveMoney` call is bound to the implicit variable `value`.

```
1 @payable
2 public func receiveMoney(implicit value: Wei) {
3   doSomething(value)
4 }
```

Listing 3.6: A Payable Function

Payable functions may have an arbitrary number of parameters, but exactly one can to be `implicit` of currency type.

3.4.4 Events

JavaScript applications can listen to events emitted by an Ethereum smart contract. When emitting an event, smart contracts can attach additional information. In the case of a money

transfer for instance, the event `didTransferComplete(caller, destination, amount)` can be emitted to notify JavaScript clients of a transfer of 30 Wei.

Events are declared in contract declarations. They are declared as special state properties which cannot be assigned. The `Event` type takes generic arguments, corresponding to the types of values attached to the event. An example is shown in Listing 3.7.

```
1  contract Bank {
2    var balances: [Address: Int]
3    let didCompleteTransfer: Event<Address, Address, Int> // (origin, destination, amount)
4  }
5
6  Bank :: caller <- (any) {
7    mutating func transfer(destination: Address, amount: Int) {
8      // Omitting the code which performs the transfer.
9
10     // A JavaScript client could listen for this event.
11     didCompleteTransfer(caller, destination, amount)
12   }
13 }
```

Listing 3.7: Using EVM Events

3.5 Remarks

3.5.1 Safety Properties

We summarise the safety properties which Flint guarantees. **No Unauthorised Internal Calls** (with the notion of Compatibility) and **Asset Type Operations** are defined in chapter 4 and chapter 5.

Property: Mutating Functions. A function f has to be declared `mutating` if it assigns a value to a state property of the type T (contract or struct) it is declared in, or if it calls a `mutating` function g.

$$\text{IsMutating(f, T)} \triangleq \forall s \in \text{Body(f)}\ (\text{IsAssignmentToState(s, T)}) \vee \exists c \in \text{FunctionCalls(s)} \left(\begin{array}{l} \neg\text{IsDeclaredLocally(Receiver(c))} \wedge \\ \text{IsMutating(MatchingDecl(c), ReceiverType(c))} \end{array} \right)$$

where
IsAssignmentToState(s, T) indicates whether the statement s assigns to a property of T,
IsDeclaredLocally(v) indicates whether v is declared as a local variable,
FunctionCalls(s) is the set of function calls in statement s,
Receiver(c) is the receiver of the function call c,
ReceiverType(c) is the type of the receiver of the function call c,
MatchingDecl(c) is the matching declaration for a function call c.

Property: Safe Arithmetics. Let $\mathbb{Z}/2^{256}$ be the set of integers between 0 and $2^{256}-1$. Let $\underline{+}$, $\underline{-}$, $\underline{*}$, $\underline{/}$ denote the arithmetic operators of Flint, $+, -, *, /$ refer to the mathematical operators, and $\rightsquigarrow$ denotes the evaluation of an expression. If a computation does not follow the following rules (e.g., the evaluation causes an overflow), an exception is thrown and the Ethereum transaction is aborted with an exception.

$$\begin{aligned}
\forall a, b, c \in \mathbb{Z}/2^{256}\ a \underline{+} b \rightsquigarrow c &\implies a + b = c \\
\forall a, b, c \in \mathbb{Z}/2^{256}\ a \underline{-} b \rightsquigarrow c &\implies a - b = c \\
\forall a, b, c \in \mathbb{Z}/2^{256}\ a \underline{*} b \rightsquigarrow c &\implies a * b = c \\
\forall a, b, c \in \mathbb{Z}/2^{256}\ a \underline{/} b \rightsquigarrow c &\implies a / b = c
\end{aligned}$$

Property: State Property Initialisation. Each state property v in a type T (contract or struct) must be initialised before the initialiser returns.

$$\text{IsInitialised(T)} \triangleq \forall \text{v} \in \text{StateProperties(T)}\ .\ \text{IsInitialised(v)}$$

where IsInitialised(v) indicates whether the property v has been assigned a default value when declared, or has been assigned in the initialiser of the type T.

Property: No Unauthorised Internal Calls. In a function f, the caller capabilities for performing each function call c must be compatible with the caller capabilities of f.

$$\begin{aligned}
&\forall s \in \text{Body(f)}.\ \forall c \in \text{ContractFunctionCalls(s)}. \\
&\quad \text{Compatible(CallerCaps(f), CallerCaps(MatchingDecl(c)))}
\end{aligned}$$

where
CallerCaps(f) returns the caller capabilities required to call a function f,
MatchingDecl(c) returns the matching declaration for a function call c,
Compatible(s, s') is described in section 4.3.

Properties: Asset Types Operations

No Unprivileged Creation. It is not possible to create an asset of non-zero quantity without transferring it from another asset.

No Unprivileged Destruction. It is not possible to decrease the quantity of an asset without transferring it to another asset.

Safe Internal Transfers. Transferring a quantity of an asset from one variable to another within the same smart contract does not change the smart contract's total quantity of the asset.

Safe External Transfers. Transferring a quantity q of an asset A from a smart contract S to an external Ethereum address decreases S's representation of the total quantity of A by q. Sending a quantity q' of an asset A to S increases S's representation of the total quantity of A by q'.

3.5.2 Towards Formal Verification

We designed Flint's features with verifiability in mind. The restricted set of operations (such as forbidding infinite loops) helps the translation of Flint programs to proof assistant programs such as Coq [44] programs, allowing the formal verification of user-defined properties about the smart contract. Furthermore, automated analysers could encode Flint's semantics to provide meaningful information about the safety of the smart contract.

3.5.3 Other Blockchains

The issues that Flint attempts to solve are not specific to the Ethereum platform. In principle, Flint's features are applicable to future blockchain based decentralised platforms. Restricting unauthorised calls is a problem inherent to any web service, and we believe exchange of assets is predominant in this domain. While immutability of a smart contract's code might not be required in other platforms, we believe separating state mutating code enhances reasoning in general. We have decoupled the Flint compiler's frontend from the EVM backend, allowing us to write new code generation libraries for other bytecode specifications in the future.

Chapter 4

Caller Capabilities

In this chapter, we present the design and implementation of Flint's *caller capabilities*, which protect functions from unauthorised accesses. When external users call a function in a Flint contract, they must have the correct caller capability. Otherwise, the call is rejected. Internal functions calls within a Flint contract are only checked at compile time, allowing great runtime performance.

4.1 Motivation

Both smart contracts and web services present a set of functions which can be called by users, i.e., the API. Controlling access to API functions is important. Unintentionally allowing an unauthorised third-party to call a privileged function can be catastrophic, such as in the first Parity attack (see subsection 2.3.2). Solidity uses function modifiers to insert dynamic checks in functions, which can for instance abort unauthorised calls. However, it is easy to forget to specify these checks, as the language does not require programmers to write them. Having a language construct which protects functions from unauthorised calls could require programmers to systematically think about which parties should be able to call the functions they are about to define.

4.2 Design

Flint's approach to controlling access to functions is performed through a feature we call *caller capabilities*. A caller capability grants an Ethereum user the permission to perform certain operations in a smart contract. When defining functions, smart contract developers need to specify explicitly which caller capability is required to call each function. Functions protected by the special caller capability `any` can be called by any user. An example is shown in Listing 4.1. Lines 1 and 5 each declare function blocks each protected by a *caller capability group*, namely `(any)` and `(manager)`. Ethereum users which hold *any* of the addresses in the caller capability group can call the functions in the block. These semantics imply a caller capability group forms a union of its members.

```
1 Bank :: (any) {
2   // The functions in this block can be called by any user.
3 }
4
```

```
Bank :: (manager) {
  // The functions in the block can be called by the Ethereum users which hold
  // the address stored in the 'manager' state property.
}
```

Listing 4.1: Flint Caller Capabilities

A caller capability represents a single or an array of Ethereum addresses. In the case of arrays, the caller needs to hold *any* of the addresses in the array in order to perform a function call. Addresses backing caller capabilities can be changed at runtime. This allows, for example, to change the address of an `admin` caller capability, or add an address to a `customers` array caller capability.

The Ethereum address of the caller of a function is unforgeable. It is not possible to impersonate another user, as a consequence of Ethereum's mechanism which generates public addresses from private keys. Transactions are signed using a private key, and determine the public key of the caller. Stealing a caller capability would hence require stealing a private key. The recommended way for Ethereum users to transfer their ability to call functions is to either change the backing address of the caller capability they have (the smart contract must have a function which allows this), or to securely send their private key to the new owner, outside of the Ethereum platform.

In addition to checking at runtime whether the caller of a function is in possession of a capability allowing them to call this function, the compiler finds invalid internal function calls statically. This prevents the programmer from, for example, calling a privileged function from a function which can be called by any user.

In the example below, we implement a smart contract a parent would deploy to set up a college fund for their child. The parent can deposit Ether gradually. Once the child is ready to start college, the parent calls `allowWithdrawal`, which enables the child to withdraw the Ether into their Ethereum account. It is important for the functions in this smart contract to be protected from unauthorised accesses.

1. Contract state.

```
contract ChildCollegeFund {
  var parent: Address
  var child: Address
  var canWithdraw: Bool = false
  var tuitionFee: Int

  var contents: Wei = Wei(0)
}
```

2. Anyone can initialise the contract. The contract can be deployed by anyone, and is initialised during contract deployment. The Ethereum user which deploys the contract is considered to be the parent.

```
ChildCollegeFund :: caller <- (any) {
  // Anyone can initialise this contract. The caller's address is bound to
  // the 'caller' variable on the line above.
  public init(child: Address, tuitionFee: Int) {
    self.parent = caller
    self.child = child
    self.tuitionFee = tuitionFee
  }
}
```

3. Functions that only the parent can call. The parent can deposit money, allow the child to withdraw the funds, get the number of Ether which has been deposited, and how much more is required to meet the tuition fee.

```
18 ChildCollegeFund :: (parent) {
19   // Only the parent can call the functions in this block.
20
21   @payable
22   public mutating func deposit(implicit value: Wei) {
23     contents.transfer(&value)
24   }
25
26   mutating public func allowWithdrawal() {
27     self.canWithdraw = true
28   }
29
30   public func getContents() -> Int {
31     return contents.getRawValue()
32   }
33   invalidWithdraw
34   public func getDistanceFromGoal() -> Int {
35     // The caller of this function is statically known to be 'parent'.
36     // Therefore, the calls to 'getTuitionFee' and 'getContents' can be performed.
37     return getTuitionFee() - getContents() // OK.
38   }
39 }
```

4. Functions the parent and the child can call. The parent and the child can see the total tuition fee. The call to `withdraw` is invalid, as it cannot be called by the parent. When calling `invalidWithdraw`, it is not statically known whether the caller is the child (it could be the parent).

```
40 ChildCollegeFund :: (parent, child) {
41   // The parent or the child can call these functions.
42
43   public func getTuitionFee() -> Int {
44     return tuitionFee
45   }
46
47   mutating public func invalidWithdraw() {
48     // The call to 'withdraw' is invalid, as it requires the caller to have the
49     // 'child' capability statically. At runtime, we only know the caller has
50     // one of the capabilities 'parent' or 'child'.
51     withdraw() // Invalid.
52   }
53 }
```

5. Functions the child can call. The child can withdraw the funds after the parent's approval. The child cannot know how much more funds are required to meet the tuition fee goal.

```
54 ChildCollegeFund :: (child) {
55   // The child can call these functions.
56
57   mutating public func withdraw() {
58     // We have not implemented the unary not (!) operator yet.
59     require(canWithdraw == false)
60     send(child, &contents)
```

```
61    }
62
63    public func getDistanceFromGoal2() -> Int {
64      // The call to 'getContents' is invalid, as it requires the caller
65      // to have the caller capability 'parent'. The caller is statically known
66      // to only have the capability 'child'.
67      return getTuitionFee() - getContents() // Invalid.
68    }
69  }
```

The compiler output for this smart contract is shown in 4.2.

4.3 Safety Property

We introduce the notion of *compatibility* between caller capabilities.

Definition: Compatibility of Caller Capabilities. A function f can call a function g if their caller capabilities are compatible. That is, either g has the special capability `any`, or any of the caller capabilities required to call f should be sufficient to call g.

$$\begin{aligned}&\text{Compatible(CallerCaps, CalleeCaps)} \triangleq \\ &\quad (\texttt{any} \in \text{CalleeCaps}) \lor (\forall c \in \text{CallerCaps}.\ c \in \text{CalleeCaps})\end{aligned}$$

Caller capabilities ensure no unauthorised internal function calls are performed. We formalise this property.

Property: No Unauthorised Internal Calls. In a function f, the caller capabilities for performing each function call c must be compatible with the caller capabilities of f.

$$\begin{aligned}&\forall s \in \text{Body(f)}.\ \forall c \in \text{ContractFunctionCalls(s)}. \\ &\quad \text{Compatible(CallerCaps(f), CallerCaps(MatchingDecl(c)))}\end{aligned}$$

where
CallerCaps(f) returns the caller capabilities required to call a function f,
MatchingDecl(c) returns the matching declaration for a function call c,
Compatible(s, s') is described in section 4.3.

4.4 Implementation

Caller capability checks are performed at compile-time for internal function calls (when a function calls another function defined in the same contract), and thus allows for finding bugs early in the development cycle. This also allows for better runtime performance by omitting runtime checks, as shown in our evaluation in chapter 8.

The caller capability of foreign contracts and Ethereum users calling into Flint programs are verified at runtime, hence protecting from unauthorised attempts to call protected functions.

4.4.1 Static Checking

During parsing, each function declaration and their associated caller capabilities are recorded in an Environment struct. The semantic analyser then verifies the validity of each function call with respect to its caller capabilities.

If a function call can be matched to a declaration but the caller capabilities required to perform the call are not sufficient, the compiler provides an additional note to help the programmer fix the issue.

```
Error in ChildCollegeFund.flint:
Function 'withdraw' cannot be called using the caller capabilities 'parent, child' at line
    48, column 5:
    withdraw()
    ^^^^^^^^^^
  Note in ChildCollegeFund.flint:
  Perhaps you meant this function, which requires the caller capability 'child' at line 53,
       column 19:
    mutating public func withdraw() {
                     ^^^^^^^^^^^^^^^
Error in ChildCollegeFund.flint:
Function 'getContents' cannot be called using the caller capability 'child' at line 62,
    column 30:
    return getTuitionFee() - getContents()
                             ^^^^^^^^^^^^^
  Note in ChildCollegeFund.flint:
  Perhaps you meant this function, which requires the caller capability 'parent' at line
      28, column 10:
    public func getContents() -> Int {
           ^^^^^^^^^^^^^^^^^^^^^^^^^^
Failed to compile.
```

Listing 4.2: Compiler Ouptut for Invalid Caller Capabilities, for the Code in Listing 4.2.

4.4.2 Dynamic Checking

To prevent external Ethereum users and smart contracts from performing unauthorised function calls, runtime checks are inserted in the function selector portion of the code, immediately before running the function's body. If the caller's address is not in the set of caller capabilities required to call the function, an exception is thrown and the call is aborted (the `REVERT` opcode is executed).

Caller capability checks are omitted for calls to functions declared in the same contract. This provides better runtime performance without compromising safety, as the checks for internal calls are verified at compile-time.

4.5 Remarks and Related Work

4.5.1 Solidity Modifiers Are More Fine-Grained

Solidity modifiers allow checking for *any* type of assertion before the body of a function is entered. However, Solidity does not enforce programmers to write them. As we believe

protecting privileged functions in smart contracts is a basic requirement, Flint requires[1] programmers to do so using caller capabilities. Our implementation is also more efficient for internal calls, as shown in section G.1. Flint users can use the `assert` function to perform other types of checks at runtime.

4.5.2 Overloading on Capability Groups

A natural extension to the caller capabilities system could be to allow overloading of functions using caller capabilities. We provide an example in Listing 4.3. We would call the `withdraw` function from line 8 if the caller has the `manager` capability, and `withdraw` from line 2 otherwise.

```
1  Bank :: (any) {
2    mutating func withdraw() {
3      // Body omitted.
4    }
5  }
6
7  Bank :: (manager) {
8    mutating func withdraw() {
9      // Body omitted.
10   }
11 }
```

Listing 4.3: Overloading Using Capabilities

We believe this presents confusing semantics. The caller capability groups can be overlapping (such as in the above example) and the overload chosen runtime might not be the one the user expected.

4.5.3 The Term "Capability"

Flint's caller capabilities system draws similarities to role-based access control systems[2] [47], wherein operations can only be performed if the user has been assigned a given role, rather than a specific set of users. A caller capability then encapsulates such a role, as the user which has been assigned the role can change after contract initialisation.

[1] A programmer could decide, however, to write all functions in an `any` block. This is discussed in section 8.2.3.

[2] After the Flint project was open-sourced, the term "caller capabilities" has been criticised to not be capabilities in the traditional sense of the term. Mark Miller et al. [45] describe four security models which make the distinction between Access Control Lists (ACLs) and different types of capabilities. In particular, it describes "object capabilities" systems as systems in which authority over an object can be transferred.

The original definition of a capability by Jack Dennis et al. [46] regard a capability as an unforgeable token (a number) which when possessed by a user, allows access to a resource. In Flint, a capability is linked with an Ethereum address, which can be regarded as an unforgeable token, as explained in 4.2. An individual has the authority to call a function if it possesses the appropriate private key, and transferring authority is done by simply sharing the private key.

?? is a proposal which was written on Flint's GitHub repository to motivate the change of the term, and suggests the replacement name "caller identities."

4.5.4 Caller Capabilities as Types

Some programming languages, such as Pony, reference capabilities associated with object references are part of the object's type. Flint's caller capabilities are not encoded in the functions' type. We could encode caller capabilities in the type system by making each function take the capability as an argument. A function of type `Int -> Int` that is only callable by the `manager` address could have type `manager -> Int -> Int`. Then, the function would not be callable if caller does not have the `manager` address. To encode caller capability groups (the caller must have one of the addresses in the group), we could use a union type: `(manager | admin) -> Int -> Int`, where `(manager | admin)` is an anonymous union type. This mechanism would however require a dependently-typed system, as caller capabilities are state properties. Implementing a dependently-typed system which would have proved to be significantly more difficult. If Flint's type system were dependently typed to support other language features, we would have encoded caller capabilities as such.

Chapter 5

Currency and Assets

In this chapter, we look at how Flint smart contracts can handle currency and other assets in a type safe manner. We introduce the concept of *Assets*. Assets in Flint describe values which cannot be accidentally created, duplicated, or destroyed. They can only be atomically transferred to other variables. These properties help ensure smart contracts are always in a consistent state. The most prevalent Asset in smart contracts is Ether. We implement the Asset operations for the `Wei` type, representing the smallest denomination of Ether. In the future, we plan on allowing the definition of user-defined Assets, which would inherit our Asset properties (see section 5.5.

5.1 Motivations

Numerous attacks targeting smart contracts, such as ones relating to reentrancy calls (see TheDAO, subsection 2.3.1), allow hackers to steal a contract's Ether. These happen because smart contracts encode Ether values as integers, making it is easy to make mistakes when performing Ether transfers between variables, or to forget to record Ether arriving or leaving the smart contract.

The `Bank` contract in Listing 5.1 records the `balances` of its customers, and implicitly assumes that the sum of all the balances reflects exactly the total amount of Wei the bank received. When supporting `withdraw` and `deposit` operations, the programmer needs to manually update the `balances` dictionary to reflect the changes.

In the following example, if either of the lines a or b were omitted, the contract's state would not be accurately representing the total amount it has. Omitting line b is more dangerous: the contract would be sending Wei without recording it in the state. A customer could withdraw the same amount until the bank's balance is completely exhausted.

```
1  contract Bank {
2    var balances: [Address: Wei]
3  }
4
5  Bank :: account <- (balances.keys) {
6    @payable
7    mutating func deposit(implicit value: inout Wei) {
8      balances[account] += value // a
9    }
10
```

```
11    mutating func withdraw() {
12      send(account, balances[account])
13      balances[account] = 0 // b
14    }
15  }
```

Inconsistencies between the smart contract's recorded balance and its actual Ether balance can also occur with reentrant calls. In TheDAO attack (see subsection 2.3.1), an attacker was able to retrieve more Ether than they had sent to the contract, due to the contract not updating its state before performing subsequent transfers.

5.2 Properties

In order to prevent state inconsistency issues such as the ones highlighted above, we introduce the concept of *Assets*. We encode currency (such as Ether) as Assets, but also other items of value, such as theatre tickets. Assets are meant to keep the state of a smart contract consistent. We formalise by defining the following properties.

Properties: Asset Types Operations

No Unprivileged Creation. It is not possible to create an asset of non-zero quantity without transferring it from another asset.

No Unprivileged Destruction. It is not possible to decrease the quantity of an asset without transferring it to another asset.

Safe Internal Transfers. Transferring a quantity of an asset from one variable to another within the same smart contract does not change the smart contract's total quantity of the asset.

Safe External Transfers. Transferring a quantity q of an asset A from a smart contract S to an external Ethereum address decreases S's representation of the total quantity of A by q. Sending a quantity q' of an asset A to S increases S's representation of the total quantity of A by q'.

`Wei` and `Ether` are Assets in Flint. The sum of the state properties of type `Wei` or `Ether` should always be equal to the actual value of the contract, as seen by miners. Assets can also be used to create sub-currencies, or *coins*, which can be minted during contract initialisation using a privileged operation.

5.3 Design and Implementation

In Flint, developers can perform asset transfers safely. The standard library implements the `Wei` type, which provides safe transfer operations.

Listing 5.1 shows how we implement the Wei type in the standard library.

```
1  struct Wei {
2    // The Wei amount as an integer.
3    var rawValue: Int
```

```
4
5   // Creates Wei directly from an integer. This is a privileged operation.
6   init(unsafeRawValue: Int) {
7     self.rawValue = unsafeRawValue
8   }
9
10  // Creates Wei by transferring a specific quantity of another Wei.
11  // Causes a fatalError() if the quantity of source is smaller than amount.
12    init(source: inout Wei, amount: Int) {
13    if source.getRawValue() < amount { fatalError() }
14    source.rawValue -= amount
15    rawValue = amount
16  }
17
18  // Creates Wei by transferring the entire quantity of another Wei.
19  init(source: inout Wei) {
20    init(&source, source.getRawValue())
21  }
22
23  // Transfers a specific quantity of another Wei into the receiver.
24  // Causes a fatalError() if the quantity of source is smaller than amount.
25  mutating func transfer(source: inout Wei, amount: Int) {
26    if source.getRawValue() < amount { fatalError() }
27    source.rawValue -= amount
28    rawValue += amount
29  }
30
31  // Transfers the entire quantity of another Wei into the receiver.
32  mutating func transfer(source: inout Wei) {
33    transfer(&source, source.getRawValue())
34  }
35
36  // Returns the quantity of Wei, as an integer.
37  func getRawValue() -> Int {
38    return rawValue
39  }
40 }
```

Listing 5.1: Wei Type Declaration

We also aim to support user-defined assets, which would inherent safe transfer operations from the standard library (see section 5.5).

5.3.1 Properties

The properties highlighted in the previous section hold intuitively:

No Unprivileged Creation. The only way to create an Asset is through the `init(unsafeRawValue:)` initialiser, which is considered to be a privileged operation. An `@privileged` function annotation is discussed in 5.6.2. An `Int` cannot be directly assigned to a variable of type `Wei`.

No Unprivileged Destruction. There is no function which allows explicitly decreasing the number of Wei using an integer. When an Asset variable goes out of scope, its contents are destroyed. This is considered to be a privileged operation, and Flint will emit a compiler warning when the contents of an Asset going out of scope are not transferred.

Safe Internal Transfers. The `init(source:amount:)` and `init(source:)` initialisers remove an amount a from a variable and adds the same amount a to another. The total amount of the Asset thus remains unchanged.

Safe External Transfers. For sending assets, we use the `send` function, which takes a Wei parameter rather than an integer value, and clears its contents when transferring. This prevents attacks such as TheDAO. Smart contracts receive Assets as parameters to functions. From **No Unprivileged Destruction**, it follows the parameter's value must be written to state before the function returns.

5.4 Example use of Assets

The illustrate how Flint's Asset operations can be used, we define a `Bank` smart contracts which makes use of the `Wei` asset.

1. Declaring the contract. The `Bank` contract will emulate a bank. Customers can send Ether to the bank, and retrieve it at a later stage. We also allow the Bank to accept donations, in order to showcase simple Asset transfers. We declare the state properties of the contract. The `balances` dictionary holds the balances of each customer, and the `accounts` array stores the addresses of each customer. `totalDonations` represents the total amount of Ether which was donated to the bank.

```
1  contract Bank {
2    var manager: Address
3    var balances: [Address: Wei] = [:]
4    var totalDonations: Wei = Wei(0)
5  }
```

2. Anyone can donate. Customers can donate Ether by calling `donate`. The Wei value attached to the Ethereum transaction is transferred to the state of the contract. The function `donate` receives an Ether amount from Ethereum, which is bound to the implicit parameter `value`. On line 8, the `transfer` function is called on `totalDonations`, which has type `Wei`. It takes as a parameter a reference to `value`, and transfers its contents to `totalDonations` in a single atomic operation. After line 8, `totalDonations` has been increased, and `value` has 0 Wei.

```
6  // The functions in this block can be called by any user.
7  Bank :: account <- (any) {
8    // Omitting the contract initialiser.
9
10   @payable
11   public mutating func donate(implicit value: Wei) {
12     // This will transfer the funds into totalDonations.
13     totalDonations.transfer(&value)
14   }
15 }
```

3. Customer operations. Customers get the balance of their account by calling `getBalance`, which returns the value as an integer. Internal transfers between accounts can be performed using `transfer`. The balances of the originator and the destination are updated atomically. The operation crash if `balances[account]` doesn't have enough Wei. The `withdraw` operation sends Wei back to the Ethereum account of the customer. The required amount is first transferred from their bank account to the local variable `w`, then sent to the Ethereum address.

```
Bank :: account <- (balances.keys) {
  public func getBalance() -> Int {
    return balances[account].getRawValue()
  }

  public mutating func transfer(amount: Int, destination: Address) {
    balances[destination].transfer(&balances[account], amount)
  }

  @payable
  public mutating func deposit(implicit value: Wei) {
    balances[account].transfer(&value)
  }

  public mutating func withdraw(amount: Int) {
    // The operation crashes if the user's bank account has less than amount Wei.
    let w: Wei = Wei(&balances[account], amount)

    // Send the amount back to the Ethereum user.
    send(account, &w)
  }
}
```

The full example is available at `https://github.com/franklinsch/flint/blob/master/examples/valid/bank.flint`.

5.4.1 Distributing Money Among Peers

We provide a more complex example use of Assets, which we cannot compile yet using the latest version of the Flint compiler. This smart contracts splits Ether among peers according to a *weights* mapping. We also split a bonus equally.

```
contract Wallet {
  var balance: Wei
  var bonus: Wei

  var beneficiaries: [Address]
  var weights: [Address: Int]
}

Wallet :: (owner) {
  mutating func distribute(amount: Int) {
    let beneficiaryBonus = bonus.getRawValue() / beneficiaries.count
    for i in (0..<beneficiaries.count) {
      var allocation = Wei(&balance, amount * weights[i])
      allocation.transfer(&bonus, beneficiaryBonus)
      send(beneficiaries[i], &allocation)
    }
  }
}
```

5.5 Generalised Assets

We also aim to implementing an Asset trait (similar to a typeclass with default implementations for some methods), to allow developers to define their own assets. This would be useful for representing sub-currencies as Assets, which would enable the use of the same safe transfer operations. This feature has not been implemented as we do not support traits yet.

5.5.1 Trait Definition

In Listing 5.5.1, we implement the Asset trait. A Flint trait is similar to a Rust trait[1], a protocol in Swift, a typeclass in Haskell, or an interface in Java. A trait cannot be instantiated, and defines which functions the structs which conform to it must implement. Developers can also specify default implementations of functions for traits, and any conforming class inherits from the functionality. As we have not implemented traits yet, generalised assets are not yet supported.

```
1  trait Asset {
2    // Create the asset by transferring a given amount of asset's contents.
3    init(source: inout Self, amount: Int)
4
5    // Unsafely create the Asset using the given raw value.
6    init(unsafeValue: Int)
7
8    // Return the raw value held by the receiver.
9    func getRawValue() -> Int
10
11   // Transfer a given amount from source into the receiver.
12   mutating func transfer(source: inout Self, amount: Int)
13 }
```

5.5.2 Default Implementation of Functions

We provide default implementations, which any type conforming to Asset inherits.

```
1  extension Asset {
2    // Create the asset by transferring another asset's contents.
3    init(from other: inout Self) {
4      self.init(from: &other, amount: other.getRawValue())
5    }
6
7    // Transfer the value held by another Asset of the same concrete type.
8    mutating func transfer(source: inout Self) {
9      transfer(from: &source, amount: source.getRawValue())
10   }
11
12   // Transfer a subset of another Asset of the same concrete type.
13   mutating func transfer(source: inout Self, amount: Int) {
14     if amount > source.getRawValue() { fatalError() }
15
16     source.rawValue -= amount
17     rawValue += amount
18   }
```

[1] Traits in Rust: `https://doc.rust-lang.org/stable/rust-by-example/trait.html`

```
}
```

5.5.3 Implementing a Plane Ticket Asset

We implement an alternate version of the `FlightManager` from subsection 3.1.1, which sells tickets without assigning seats immediately. We first declare a new `PlaneTickets` asset, backed by an integer value.

```
struct PlaneTickets: Asset {
  var rawValue: Int

  init(unsafeValue: Int) {
    rawValue = unsafeValue
  }

  init(from other: inout Asset, amount: Int) {
    rawValue = 0
    transfer(from: &other, amount: amount)
  }

  func getRawValue() -> Int {
    return rawValue
  }
}
```

Listing 5.2: Implementing Wei Using the Asset Trait

We then update our `FlightManager` declaration as follows. We do not use unsafe operations, except for initialising the initial number of tickets.

```
contract FlightManager {
  // State properties as before.

  // The plane has 140 tickets.
  var tickets = PlaneTickets(140)
  var ticketAllocations: [Address: PlaneTickets] = [:]
}

FlightManager :: caller <- (any) {
  // Initialiser and 'allocateSeat' as before.

  @payable
  mutating public func buy(implicit value: Wei) {
    assert(value.getValue == ticketPrice)
    amountPaid[caller].transfer(&value)

    // Transfer one ticket. An exception is thrown if there are no more available tickets.
    ticketAllocations[caller].transfer(&tickets, 1)
  }
}

FlightManager :: passenger <- (ticketAllocations.keys) {
  // As before.
}

FlightManager :: (admin, ticketAllocations.keys) {
```

```
27    mutating func refund(passenger: Address) {
28      let refund = Wei(&amountPaid[passenger])
29      allocations[passenger] = nil
30
31      // Transfer the ticket back to the ticket pool.
32      tickets.transfer(&ticketAllocations[passenger])
33
34      send(passenger, &refund)
35    }
36  }
```

5.5.4 Compiler Warnings for Misusing Assets

To implement **No Unprivileged Destruction**, each local variable and parameter of an Asset type should be transferred to another local variable or state property. This implies that Assets are never implicitly destroyed when a function exists—they are transferred to state. Assets transferred from state to local variables need to eventually be transferred back. An Asset received through a parameter (including the `implicit` value of an `@payable` function) must eventually be recorded to state. We plan to implement a compiler warning to alert the programmer when this rule is not respected. In subsection 5.6.1, we discuss why we chose to produce a warning rather than an error.

```
1  @payable
2  public mutating func deposit1(implicit value: Wei) {
3    // Warning: local variable 'value' has not been transferred.
4  }
5
6  @payable
7  public mutating func deposit2(implicit value: Wei) {
8      let w: Wei = Wei(&value) // OK
9      // Warning: local variable 'w' has not been transferred.
10 }
11
12 @payable
13 public mutating func deposit(implicit value: Wei) {
14   balances[account].transfer(&value) // OK
15 }
```

Listing 5.3: Errors for Not Using Linear Values

5.6 Remarks

5.6.1 Linear Types

Substructural type systems [48], in particular *linear* type systems, aid the implementation of Asset types. Linear type systems are generally used to determine when heap-allocated objects can be deallocated. Specifically, linear objects need to be used exactly once in the scope where they are defined. In Flint, we use a similar approach to check if all local variables of an Asset type are transferred exactly once to ensure receiving an Asset always results in transferring it to state.

In 5.5.4, we discussed a linear type approach to producing warnings when Asset local variables are implicitly destroyed when exiting their declaration scope. Due to aliasing difficulties and partial transfers, we cannot always determine if an Asset has been entirely transferred.

Listing 5.4 highlights two cases where we cannot guarantee that an Asset has been transferred entirely exactly once. In `transfer`, if `indexA` and `indexB` are equal (which cannot be checked at compile-time, the second transfer operation should be prohibited as the value of `arr[indexB]` would have been transferred already. In `deposit`, we cannot know at compile-time whether all of the contents of `value` are transferred when we transfer 10 Wei.

Due to our inability to statically check whether all local variables are used exactly once, we do not produce errors, but warnings which can be ignored. Producing errors would produce an inconsistent experience for the developer, who would expect *all* instances of unsafe code would be rejected.

```
1  public mutating func transfer(arr: inout [Wei], indexA: Int, indexB: Int) {
2    wallet.transfer(&arr[indexA])
3    wallet.transfer(&arr[indexB]) // Does indexA == indexB?
4  }
5
6  @payable
7  public mutating func deposit(implicit value: Wei) {
8    balances[account].transfer(&value, 10)
9    // Has 'value' been transferred completely?
10 }
```

Listing 5.4: Linear Values Aliasing

We described our approach for requiring the transfer of local variables, but not for state properties. State properties do not have to be transferred in each scope. The compiler could, however, check they are used *at most* once in each scope. This would be an implementation of *affine* type theory.

5.6.2 Protecting Privileged Operations

It is still possible to write unsafe code using Flint's Asset types, through the use of the `init(unsafeValue:)` initialiser, which allows any currency to be created from an integer. The use of unsafe operations can easily be caught by the semantic analyser, and reported to users of the smart contract. We cannot disallow developers the use of this initialiser, as there are valid uses cases for it, such as the minting of a currency.

In the future, however, we'd like to add an `@privileged` function annotation. The compiler would require the annotation to be present if the function uses privileged operations. A user-defined function initialising a Wei directly from an integer value would need to be annotated `@privileged`. Similarly, a function which implicitly destroys an Asset (by not transferring the contents of a local Asset variable to a state property), could be annotated `@privileged` to silence warnings.

The `Wei` declaration would be updated to use the `@privileged` keyword, as shown in Listing 5.5. We also show how implicit Asset destruction could be allowed using using the keyword.

```
1  struct Wei {
2    var rawValue: Int
3
```

```
4   @privileged
5   init(unsafeRawValue: Int) {
6     self.rawValue = unsafeRawValue
7   }
8
9   // Same declarations as before.
10 }
11
12 Fund :: (any) {
13   @privileged @payable
14   func receiveMoney(implicit value: Wei) {
15     // No warning is produced.
16   }
17 }
```

Listing 5.5: Implementing Wei Using the Asset Trait

5.6.3 Conversion between Assets

It is sometimes useful to convert a type of Asset into another type of Asset. For example, it should be possible to safely perform currency conversions between Ether and a sub-currency. In our `PlaneTickets` example, we could have created an initialiser which takes Wei as a parameter, and initialises a certain number of plane tickets. We would however have needed to unsafely create plane tickets. To sell tickets, we could have created another initialiser for `Wei`, to convert a ticket's value back to Wei. In the future, we would like to provide a better scheme for converting between Asset types.

Chapter 6

Compiler and Code Generation

We implement `flintc`, a compiler for Flint written in about 25 000 lines of Swift [15] code, and can be used on the Linux and macOS operating systems. The compiler's source code is open source and available on GitHub [24]. We choose Swift as it is a modern programming language for writing well-designed, performant code. We were able to implement the compiler while designing the language, using a highly extensible code architecture allowing us to quickly implement new features and analyses (see section 6.3).

The compiler stages are illustrated in Figure 6.1 and described in this chapter. Input Flint programs are analysed, compiled to the IULIA [43] intermediate representation, and finally to EVM bytecode. By embedding IULIA in a Solidity file, tools built for Solidity work with Flint. We compile IULIA code using the Solc [5] compiler[1]. We also describe Flint's Application Binary Interface (ABI) which allows interoperability with Solidity, and present the runtime organisation of Flint programs.

We implement a full test suite to verify the validity of our compiler (see section 6.13).

6.1 Tokeniser

The tokeniser converts raw source text of a Flint input program into a sequence of tokens. We elected to implement a tokeniser ourselves, rather than using a tokeniser library. This allowed better flexibility in the language's syntax, which evolved significantly since its inception.

We implement the tokeniser in `Parser/Tokenizer.swift`. The input text is first split into words, and each word is matched to a token. The different types of Flint tokens are defined in `AST/Token.swift`. We organise tokens into different groups: keywords (such as `contract`, `struct`, or `func`), punctuation (such as `{`, `->`, or `+`), function attributes (such as `@payable`), literals (boolean, string, and number literals), and `identifiers` (such as `MyContract`).

Operators, such as `+`, `&+` (overflowing plus), `+=`, and `||`, have an associated precedence. We adopt the precedence rules programmers expect.

When tokenising, we maintain the source location (line and column number) of each token in the source file. This helps provide better error messages in the later compiler stages.

[1] When developing the compiler, `solc` could not compile IULIA code directly—it had to be embedded in a Solidity file

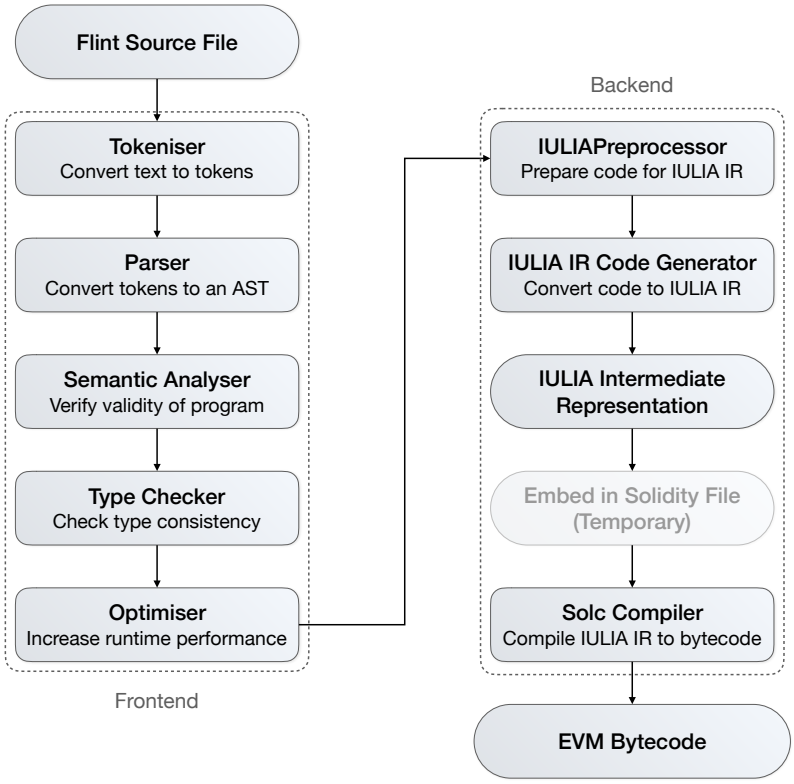


Figure 6.1: Compiler Stages

6.2 Parser

The parser processes the sequence of tokens into an Abstract Syntax Tree (AST). We also developed the parser ourselves, rather than using a third-party library, which made it easier to update Flint's grammar rules and have better control over what we store in each node. Our parser uses the Flint grammar, which can be read in Appendix A. We implement a custom recursive descent parser with variable lookahead. If the program cannot be parsed, the compiler attempts to find the invalidly placed token and produces an error message.

In most cases, the parser only looks at the next token to determine which rule of the grammar to match. The exception is for expressions—the location of the last token of the expression is searched before parsing. We do this by implementing a `indexOfFirstAtCurrentDepth` function, which finds the first occurrence of a token at a given syntactic depth (opening or closing a bracket increases or decreases depth). For example, when parsing a function call argument, we determine the location of the end of the expression by using `indexOfFirstAtCurrentDepth` to find a comma or a close bracket token.

The parser also creates an Environment, which records various information about the contracts and structs defined in the Flint program.

The AST's nodes are defined in `AST/AST.swift`. A node's children are stored in state properties. A child can either be a terminal (a token), or another node. In some cases, not all properties need to be set. For instance, a FunctionDeclaration node can have its `resultType` property set to `nil` if the function does not explicitly return a value. Nodes also have a `sourceLocation` computed property[1], which describes their span in the original source file.

[1]A computed property does not have any associated memory storage—its value is computed in terms of

Using this mechanism, we can highlight entire nodes in error messages, rather than single tokens.

6.3 The AST Pass Mechanism for Better Extensibility

6.3.1 AST Visitor and AST Passes

The Semantic Analyser, Type Checker, Optimiser, and IULIA Preprocessor are stages of the compiler which all require traversing the AST. We implement a code architecture which decouples AST traversal and node processing. By *processing*, we mean modifying a node, updating contextual information such as the Environment, and produce errors or warnings.

The traditional visitor pattern [49] leverages dynamic method dispatch mechanisms to separate node processing logic from tree traversal logic. However, it requires for visited nodes to invoke the visitor on their children nodes manually. In our design, nodes do not have references to visitors. We create an architecture which uses a single *AST Visitor*, and multiple *AST Passes*. Figure 6.2 illustrates this process. The AST Visitor is initialised with an AST Pass, and visits each of the AST's nodes. The AST Visitor passes a copy of the node to appropriate *process* the current AST Pass. Each AST Pass implements a process function for each type of AST node. The AST Visitor updates the tree using the (potentially modified) nodes returned by the AST Pass, collects diagnostics, and propagates contextual information (see subsection 6.3.2).

After visiting the children of a node, the AST Visitor visits the parent node again in a *post process* step. Visiting the parent again is necessary because in some cases, some properties can only be checked after visiting the children. For example, to know whether a function declaration should be marked `mutating`, we first need to visit the children of the node to determine if any are statements which mutate the state of the contract.

We implement four AST Passes, which we run the in the following order: Semantic Analysis (see section 6.4), Type Checker (see section 6.5), Optimiser (see section 6.6), and IULIA Preprocessor (not shown in figure, see section 6.7).

6.3.2 Propagating Information when Visiting

In many cases, when visiting children of an AST node, an AST Pass requires information from the parent node. For instance, when visiting a Function Call node, we statically check that the function call can be performed with regards to caller capabilities (see subsection 4.4.1). For this, the AST Pass (in this case, the Semantic Analyser) needs to know which caller capabilities are required to call the function which contains the function call. To propagate information down the visit tree, the AST Visitor provides a Context value[1] (`AST/ASTPassContext.swift`) to each of the functions in the AST Passes. In the Semantic Analyser, we create a `ContractBehaviorDeclarationContext` value when processing a Contract Behaviour Declaration, and insert it into the Context. The AST Visitor propagates the Context to each of the children nodes, and the value is retrieved when visiting the Function Call node.

other properties

[1] In Swift, a value is like a Java object, but with value semantics (copied when assigned or passed as a function argument).

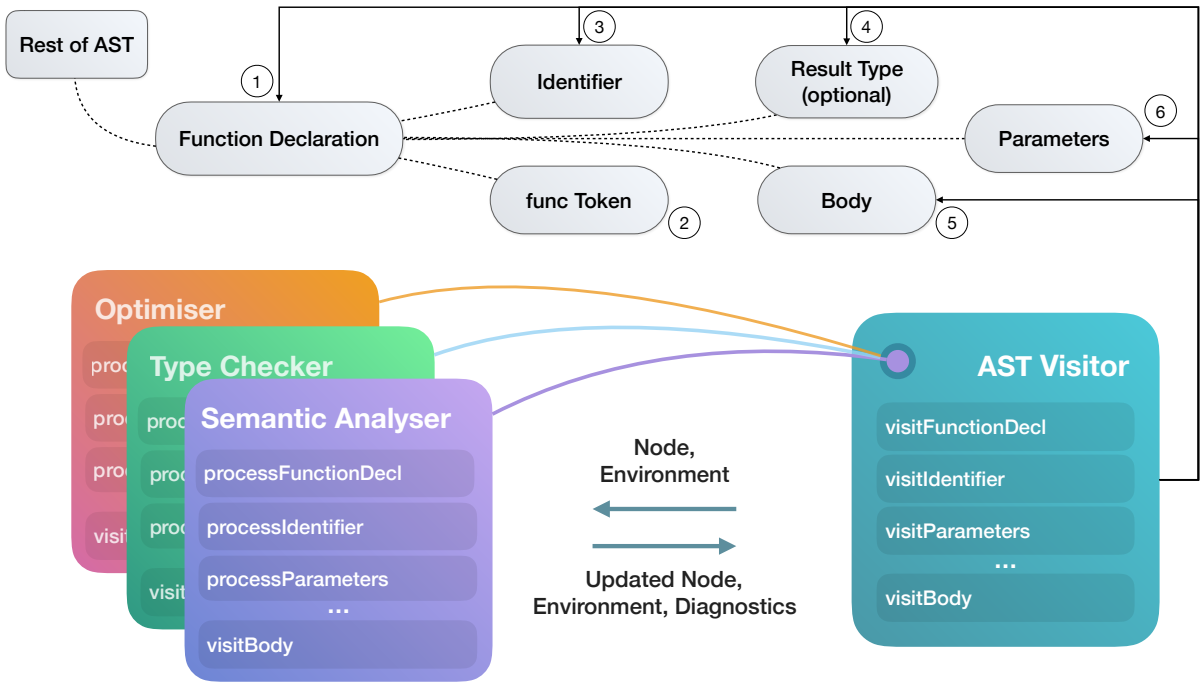


Figure 6.2: Visiting the AST Nodes using AST Passes, in the Specified Order

Context information is also available in the *post process* step, which processes parent nodes *after* visiting their children. For determining whether a function declaration should be marked `mutating`, the Semantic Analyser writes the mutating statements of the function's body to the Context when visiting the statements of the function declaration. When the function declaration is processed again, the Semantic Analyser retrieves the list of mutating statements from the context.

The information stored in a Context value is backed by a dictionary, and value within can be accessed like regular properties (e.g., `context.contractBehaviorDeclarationContext`). Each AST Pass can configure which values to store in the Context, without needing to modify the Context's original source code. In all AST Passes, the program's Environment, which holds information such as which functions and structs are defined in the smart contract, is always present in the Context, and can be updated by AST Passes.

6.3.3 Code Example: Semantic Analyser for Contract Declarations

We provide an example of the AST Visitor and AST Pass mechanism in Listing 6.1. The `ASTVisitor` accepts an `ASTPass` as a generic argument. When visiting a contract declaration, we call the AST Pass' `process` function for each of its child nodes. The `process` function returns an `ASTPassResult` value, which holds the potentially modified ContractDeclaration node, diagnostics, and the potentially modified Context. We use this context when processing the child nodes of the contract declaration. When visiting the child nodes, we combine their `ASTPassResult` values together with the parent's `ASTPassResult`. This allows us to propagate information to child nodes during the visit, without having to modify the original AST. The combining operation is performed by `combine`, which merges diagnostics and contexts, and returns the mutated child node, so that it can be attached to the parent node. After visiting the child nodes, we process the Contract Declaration again. In the Semantic Analyser, the process function makes use of the `environment` property of the context given by the AST

Visitor to check whether a contract or struct of the same name has already been defined and whether the contract has a public initialiser defined, generating an error in the negative case.

```
// ASTVisitor.swift
public struct ASTVisitor<Pass: ASTPass> {
 func visit(_ contractDeclaration: ContractDeclaration, passContext: ASTPassContext) ->
     ASTPassResult<ContractDeclaration> {

  // Process the Contract Declaration.
  var processResult: ASTProcessResult<ContractDeclaration> = pass.process(
      contractDeclaration: contractDeclaration, passContext: passContext)

  // Process the children nodes.

  processResult.element.identifier = processResult.combining(visit(processResult.element.
      identifier, passContext: processResult.passContext))

  processResult.passContext.contractStateDeclarationContext =
      ContractStateDeclarationContext(contractIdentifier: contractDeclaration.identifier)

  processResult.element.variableDeclarations = processResult.element.variableDeclarations.
      map { variableDeclaration in
    return processResult.combining(visit(variableDeclaration, passContext: processResult.
        passContext))
  }

  // Reset the context.
  processResult.passContext.contractStateDeclarationContext = nil

  // Process the Contract Declaration again, after visiting the children.
  let postProcessResult = pass.postProcess(contractDeclaration: processResult.element,
      passContext: processResult.passContext)
  return ASTPassResult(element: postProcessResult.element, diagnostics: processResult.
      diagnostics + postProcessResult.diagnostics, passContext: postProcessResult.
      passContext)
 }

 // Other visit functions.
}

// SemanticAnalyzer.swift
public struct SemanticAnalyzer: ASTPass {
 public func process(contractDeclaration: ContractDeclaration, passContext: ASTPassContext)
      -> ASTPassResult<ContractDeclaration> {
  var diagnostics = [Diagnostic]()
  let environment = passContext.environment!

  // Check whether a contract or struct with the same identifier.
  if let conflict = environment.conflictingTypeDeclaration(for: contractDeclaration.
      identifier) {
    diagnostics.append(.invalidRedeclaration(contractDeclaration.identifier, originalSource
        : conflict))
  }

  // Check whether the contract has a public initializer defined.
```

```
41    if environment.publicInitializer(forContract: contractDeclaration.identifier.name) == nil
          {
42      diagnostics.append(.contractDoesNotHaveAPublicInitializer(contractIdentifier:
            contractDeclaration.identifier))
43    }
44    return ASTPassResult(element: contractDeclaration, diagnostics: diagnostics, passContext:
          passContext)
45    }
46
47    public func postProcess(contractDeclaration: ContractDeclaration, passContext:
          ASTPassContext) -> ASTPassResult<ContractDeclaration> {
48      return ASTPassResult(element: contractDeclaration, diagnostics: [], passContext:
            passContext)
49    }
50
51    // Other process functions.
52  }
```

Listing 6.1: ASTVisitor and ASTPass Code Example

6.4 Semantic Analysis

The Semantic Analysis phase is an AST Pass which verifies the correctness of the input program. This includes performs the static checks for caller capabilities, checking whether functions are annotated `mutating` when required, verifying whether there are uses of undefined variables, etc.

The errors and warnings (*diagnostics)* `flintc` produces in the semantic analysis phase are highlighted below. When displaying diagnostics, we display the message and print the code from the input program which caused the issue, highlighting the relevant portion of the code. Some diagnostics also include additional notes which can help find bugs. Examples of compiler diagnostics are shown in Listing 3.3, Listing 3.5, and Listing 4.2.

Caller Capabilities

Use of undeclared caller capability.

```
Caller capability 'admin' is undefined in 'Bank', or has incompatible type.
```

No matching function for function call.

```
Function 'setManager' is not in scope or cannot be called using caller capabil-
ity '(any)'.  Note:  Perhaps you meant this function, which requires caller ca-
pability '(manager)'.
```

Mutation

Mutating statement in nonmutating function.

```
Use of mutating statement in a nonmutating function.
```

No mutating statements in mutating function (Warning).

```
Function does not have to be declared mutating:  none of its statements are mu-
tating.
```

Reassignment to constant.

```
Cannot reassign to value: 'manager' is a let-constant. Note: 'manager' is declared on line 18, column 12.
```

Initialisation

State property is not assigned a value.

```
State property 'manager' needs to be assigned a value, as no initialiser was declared.
```

Return from initialiser without initialising all properties.

```
Return from initialiser without initialising all properties. Note: 'manager' is uninitialised.
```

Contract does not have a public initialiser.

```
Contract 'Bank' needs a public initialiser accessible using caller capability 'any'.
```

Contract has multiple public initialisers.

```
A public initialiser has already been defined. Note: A public initialiser is defined on line 5, column 6.
```

Public contract initialiser is not accessible using caller capability `any`.

```
Public contract initialiser should be callable using caller capability 'any'.
```

Invalid Declarations

Invalid redeclaration of an identifier.

```
Invalid redeclaration of 'setManager'. Note: Previous declaration on line 12, column 4.
```

Use of invalid character. The `$` character is reserved for use in the standard library.

```
Use of invalid character '$' in 'my$Func'.
```

Contract Behaviour Declaration has no matching Contract Declaration.

```
Contract behaviour declaration for 'Bank' has no associated contract declaration.
```

Invalid contract behaviour declaration.

```
Contract behaviour declaration for Bank has no associated contract declaration.
```

Invalid `@payable` function.

```
receive is declared @payable but doesn't have an implicit parameter of a currency type.
```

Ambiguous @payable value parameter.

```
Ambiguous implicit payable value parameter. Only one parameter can be declared 'implicit' with a currency type.
```

Public function has a parameter of dynamic type, such as struct, array, or dictionary.

```
Function 'isSeatFree' cannot have dynamic parameters. Note: 'seat' cannot be used as a parameter.
```

Use of undeclared identifier.

```
Use of undeclared identifier 'manager'.
```

Missing return in non-void function.

```
Missing return in function expected to return 'Int'.
```

Code after return (Warning).

```
Code after return will never be executed.
```

6.5 Type Checker

The type checker ensures the input program is type correct. As we do not support type inference yet, and do not plan to support subtyping, the type checker has a straightforward implementation. The diagnostics which this AST Pass produces are listed below.

Incompatible return type.

```
Cannot convert expression of type 'Int' to expected return type 'Address'.
```

Incompatible assignment.

```
Incompatible assignment between values of type Int and Wei.
```

Incompatible argument type.

```
Cannot convert expression of type Int to expected argument type Wei
```

6.6 Optimiser

This AST Pass currently does nothing. We plan to implement optimisations, such as constant folding (pre-computing operations involving number literals), peephole optimisations (replacing multiple instructions by a single one), and avoiding writing the value 0 to uninitialised memory or storage (all entries memory and storage are 0 by default).

6.7 Code Generation and Runtime

Flint targets the IULIA [43] intermediate representation (IR), developed by the engineers behind Solidity, who created IULIA as a future IR for Solidity. IULIA code can also be embedded within a Solidity function, and is used by developers who want more fine-grained control over the bytecode execution.

6.7.1 IULIA Preprocessor

Before the code generation phase, which we describe in subsection 6.7.2, we apply a preprocessing step which prepares the AST for code generation. This AST Pass mangles function names and passes the receiver of function calls as their first argument (see subsection 6.8.2), introduces an `isMem` parameter for each parameter which can be passed by value or by reference (see subsection 6.8.1), and default property assignments to the beginning of the contract's initialiser (see subsection 6.11.3).

6.7.2 Generating Code

The code generation logic is available in `IRGen/`. We implement a code generating struct per AST Node, which takes an AST node as a parameter, and returns its IULIA representation. The `IULIAStruct` struct, which generates code for a Flint struct, is shown in Listing 6.2. Similarly, we implement `IULIAFunction`, `IULIAAssignment`, `IULIAExpression`, etc.

```
/// Generates code for a struct. Structs functions and initialisers are embedded in the
    contract.
public struct IULIAStruct {
  var structDeclaration: StructDeclaration
  var environment: Environment

  func rendered() -> String {
    // At this point, the initializers have been converted to functions.

    return structDeclaration.functionDeclarations.compactMap { functionDeclaration in
      return IULIAFunction(
        functionDeclaration: functionDeclaration,
        typeIdentifier: structDeclaration.identifier,
        environment: environment
      ).rendered()
    }.joined(separator: "\n\n")
  }
}
```

Listing 6.2: IULIAStruct definition, which Generates Code for Flint Structs

6.7.3 Public Functions and Application Binary Interface

Flint's Application Binary Interface (ABI) specifies at the bytecode level how Ethereum users and other smart contracts can call the public functions of a Flint smart contract. Flint's ABI follows Solidity's ABI, which allows interoperability between the two languages.

Users can call a smart contract's function on Ethereum is performed by creating a transaction, and specify which function to call with which arguments in the transaction payload. Transaction payloads are raw bytes, thus the data needs to be encoded.

Function Resolution

Specifying which function to call is done via encoding the function's signature. The encoding is performed as follows.

1. Canonicalising the function signature. The canonical form of a function f with three parameters of types $T1, T2, T3$ is `f(T1,T2,T3)`. A function f with no arguments has the canonical form `f()`. Public functions, which can be called from outside the smart contract, can only carry parameters of basic types (except for implicit parameters such as in a `@payable` function). The canonical IR type for each basic type is described in Figure 6.3.
2. Computing the Keccak-256 hash of the canonical form.
3. The first four bytes of the hash constitute the final encoding.

Flint type	Canonical IR type
`Address`	`address`
`Int`	`uint256`
`Bool`	`uint256`
`String`	`bytes32`

Figure 6.3: Canonical Types

For example, a function f with parameters `Int` and `Bool`, has canonical form `f(uint256,uint256)`. The encoding of f is the first four bytes of the Keccak-256 hash, i.e., `0x13d1aa2e`.

Currently, Flint uses 256-bit values to represent integers and booleans, but we are planning to optimise memory usage to use fewer bytes (especially for booleans). see section 7.1.7 for more details.

Specifying Function Arguments

Function arguments are appended to the function signature hash, as a hexadecimal value. Calling f with arguments `100` and `true` would be encoded as `0x64` and `1`, padded with zeros to fill a 256-bit value.

An Ethereum user or another smart contract can thus call f with arguments 100 and *true* by entering the following value in the transaction payload (without newline characters):

```
0x13d1aa2e
0000000000000000000000000000000000000000000000000000000000000064
0000000000000000000000000000000000000000000000000000000000000001
```

6.8 Internal functions

6.8.1 Pass by Reference Implementation

A struct value passed as an `inout` argument to a function is an implicit reference to either an EVM memory location or an EVM storage location. When accessing the memory location, the runtime needs to know whether it should read the value from memory or from storage. To support this, when a struct is passed by reference to a function, an extra boolean argument, specifying the location of the reference, is inserted in the argument list of the function call. If the compiler cannot determine the location of the reference statically, a special `isMem` argument is passed. If it is statically known that a reference is a memory location or a storage location, the value `0` or `1`, respectively, is passed.

We provide an example contract Listing 6.3, and its generated IR code in Listing 6.4. The function foo takes an `inout` parameter, indicating it is passed by reference. The compiler inserts an `isMem` parameter to indicate whether the reference is a memory location or a storage location. When `bar` is called, the compiler replaces the reference to the storage property `element` by its offset in storage. The second argument is 0, to indicate the value is a storage property. When passing `arg` as a reference, we do not know its provenance statically. Therefore, we forward its `isMem` argument. When calling `baz`, we know that `s` is a

memory reference (local variables are stored in memory), so we pass the value `1` for its `isMem` parameter.

```
contract C {
  var element: S
}

C :: (any) {
  // Initialiser omitted.

  mutating func foo(arg: inout S) {

    bar(&self.element, &arg)
  }

  func bar(a: inout S, b: inout S) {
    var s: S = S()

    baz(&b, &s)
  }

  func baz(b: inout S, c: inout S) {
    // Body

  }

}
```

Listing 6.3: C Contract

```


// Generated IR code
function foo(_arg, _arg$isMem)  {
  // '0' as 'element' is a storage location.
  bar(add(0, 0), 0, _arg, _arg$isMem)
}

function bar(_a, _a$isMem, _b, _b$isMem)  {
  let _s := flint$allocateMemory(0)
  S_init(_s, 1) // '1' as '_s' is a memory
      location.
  baz(_b, _b$isMem, _s, 1)
}

function baz(_b, _b$isMem, _c, _c$isMem)  {
  // Body


}
```

Listing 6.4: Generated Bytecode

6.8.2 Mangling

Local variable and parameter names are prepended with an underscore to avoid clashes with IULIA keywords, e.g., a parameter `p` will be referred to as `_p` after code generation.

Struct function names are mangled to support function overloading and defining functions of the same name in different structs. Contract function names are not mangled, as external smart contracts and users rely on knowing their exact name to call them. Thus, overloading is not supported for contract functions.

A function f declared in a struct S with parameters of types $T1, T2, T3$, the mangled name is `S_f_T1_T2_T3`. If a parameter is passed `inout`, its type is prepended with `$inout` when mangling, e.g., `S_f_$inoutT1`.

When calling struct functions, a reference to the receiver is passed `inout` as the first argument, with its corresponding `$isMem` parameter as described in 6.8.1.

6.9 Storage and Memory Organisation

The storage and memory of a Flint smart contract are organised similarly. The runtime functions `load` and `store` (see Table 6.1) work for both variants.

A contract's state properties are stored in EVM storage sequentially, except for values in dynamic arrays and dictionaries.

Local variables are stored in memory, and are allocated dynamically. The `allocateMemory` runtime function reserves a number of bytes in memory, and returns the start pointer of the block. The first 64 bytes of memory (8 words) are reserved as scratch space and can be used to perform temporary computation, or load values into memory to compute `sha3` hashes or emit Ethereum events. Memory location `0x40` (64th byte) holds a pointer to the next available memory location (initially `0x60`). Because Ethereum transactions are quite short, we have not implemented a memory freeing mechanism yet.

6.9.1 Contracts and Structs

State properties of smart contracts are stored contiguously in storage, starting at location 0. Each state property occupies one word (32 bytes) in the case of basic types (see Figure 6.3), or multiple words when storing structs or fixed-sized arrays.

Structs can also be stored in memory.

6.9.2 Arrays and Dictionaries

A fixed-size array of size n of element type of size e is allocated $n * e$ bytes in storage or memory.

Dynamically-sized types, such as arrays and dictionaries, are not allocated contiguously. Metadata for dynamically-sized types is stored in a single byte. The metadata byte for arrays specifies the number of elements in the array, and it is unused for dictionaries. The location of the metadata byte is used for retrieving the offset of a value at a given key.

The deterministic operation which determines the storage offset for a value of key k is in a dictionary for which the metadata is stored at offset d is:

$$\text{DynStorageOffset}(d, k) \triangleq \textsc{Keccak-256}(d \cdot k)$$

where $\cdot$ is the concatenation operator.

The hashing operation returns an offset into storage which, for two distinct pairs of key and dictionary offsets, yields the same value (i.e., collides) with a very low probability ($\approx \frac{1}{2^{128}}$). The offsets for consecutive keys do not necessarily yield consecutive offsets for their associated values. Practically, value offsets can be any number between 0 and 2^{256}. Although this would be problematic for traditional computer architectures, as 2^{256} bytes would need to need to be allocated for each smart contract, it is not a problem on Ethereum as storage is itself a key-value mapping which is efficiently allocated. Storage accesses yield the same gas cost regardless of which location is accessed.

Accessing values in a dynamic array is performed similarly, with the key value k being the integer index into the array.

Memory is not a key-value store, but a contiguous sequence of bytes. Allowing value offsets to be any number between 0 and 2^{256} would yield enormous gas costs, as 2^{256} bytes of memory would have to be allocated in the worst case. We need another scheme to implement dynamic dictionaries and arrays in memory, which we have not implemented yet.

6.10 Runtime Functions

The Flint runtime contains 20 runtime functions to perform low-level operations. A runtime function f can be called from the Flint standard library using `flint$f`. Runtime functions cannot be called from user-defined code. We provide a short overview of Flint's runtime functions in Figure 6.1.

Function	Description
`selector()`	Returns the first four bytes of the Ethereum transaction payload (per the ABI, see subsection 6.7.3), to determine which function to call.
`decodeAsAddress(offset)`	Decodes the argument at the given byte offset of the transaction payload as an address.
`decodeAsUInt(offset)`	Same as above, but for integer arguments.
`store(ptr, val, isMem)`	Stores the given value at the given pointer, in storage or memory according to `isMem`.
`load(ptr, isMem)`	Loads a value from storage or memory.
`computeOffset(base, offset, isMem)`	Computes the offset of a value in a data type which starts at the given base. Accesses both storage and memory in a word addressed fashion.
`allocateMemory(size)`	Allocates a block of the given size in memory.
`isValidCallerCapability(address)`	Indicates whether the caller has the given caller capability.
`isCallerCapabilityInArray(offset)`	Indicates whether the caller has one of the caller capabilities in the array starting at offset.
`return32Bytes(v)`	Terminates the Ethereum transaction, returning a 32 byte value.
`isInvalidSubscriptExpression(index, arraySize)`	Returns whether the given array index is out of bounds with respect to the array's size.
`storageFixedSizeArrayOffset(offset, index, arraySize)`	Computes the offset of a fixed sized array.
`storageArrayOffset(arrayOffset, index)`	Computes the offset of a dynamic array.
`storageDictionaryOffsetForKey(offset, key)`	Computes the offset of value corresponding to the given key, given the dictionary's offset.
`send(value, address)`	Sends Wei to the given Ethereum address.
`fatalError()`	Terminates the transaction, reverting all state changes.

`add(a, b)`	Computes `a + b`, causing a `fatalError()` if an overflow occurs. We detect overflows by checking whether the result of the operation is bigger than `a`
`sub(a, b)`	Computes `a - b`, causing a `fatalError()` if an overflow occurs. We check whether `b` is bigger than `a` to detect overflows.
`mul(a, b)`	Computes `a * b`, causing a `fatalError()` if an overflow occurs. We check whether the result divided by `a` is equal to `b`.
`div(a, b)`	Computes `a / b`, causing a `fatalError()` if `b` is zero.

Table 6.1: Flint Runtime Functions

6.11 Intermediate Representation Organisation

6.11.1 IR Overview

We take a look at the structure of the intermediate representation code the Flint compiler generates for the following Flint contract.

```
contract Counter {
  var value: Int = 5
}

Counter :: (any) {
  public init() {}

  public func getValue() -> Int {
    return value
  }

  mutating public func set(value: Int) {
    self.value = value
  }
}
```

Listing 6.5: Flint Counter Contract

1. Selecting the function to execute. When the smart contract receives a function call, the first step is to determine which function to execute. The runtime function `selector()` decodes the first four bytes of the transaction data to determine which function should be called. The four bytes are compared to the encoding of the public functions defined in the contract, using a switch statement generated at compile time. If no match is found, an exception is thrown. The function arguments are then decoded from the transaction payload, and are passed to the relevant function. An IR function is generated per Flint contract function.

```
switch flint$selector()
```

```
2
3  case 0x20965255 /* getValue() */ {
4    flint$return32Bytes(getValue())
5  }
6
7  case 0x60fe47b1 /* set(uint256) */ {
8    set(flint$decodeAsUInt(0))
9  }
10
11 default {
12   revert(0, 0)
13 }
```

Listing 6.6: Intermediate Representation Example

2. Contract function definitions. The next part of the code consists of the generated code for user-defined contract functions. The IULIA code does not include explicit declarations of state properties, as accesses to storage properties in functions are represented as static offsets into memory.

```
14 function getValue() -> ret {
15   ret := sload(add(0, 0))
16 }
17
18 function set(_value)  {
19   sstore(add(0, 0), _value)
20 }
```

Listing 6.7: Contract Functions

3. Struct function definitions. The function code for each user-defined and standard library struct is included next. The first parameter is the receiver, `_flintSelf`.

```
21 // Standard library struct functions
22
23 function Wei_init_Int(_flintSelf, _flintSelf$isMem, _unsafeRawValue)  {
24   flint$store(flint$computeOffset(_flintSelf, 0, _flintSelf$isMem), _unsafeRawValue,
         _flintSelf$isMem)
25 }
26
27 ...
```

Listing 6.8: Struct Functions

4. Runtime functions. Finally, we include the definition of each runtime function.

```
28 // Runtime functions
29
30 function flint$decodeAsUInt(offset) -> ret {
31   ret := calldataload(add(4, mul(offset, 0x20)))
32 }
```

Listing 6.9: Runtime Functions

6.11.2 Embedding in a Solidity File

The IR code is placed within the fallback function of a Solidity file, as shown in Listing 6.10. The fallback function is executed when any function call targeting the Solidity contract is performed.

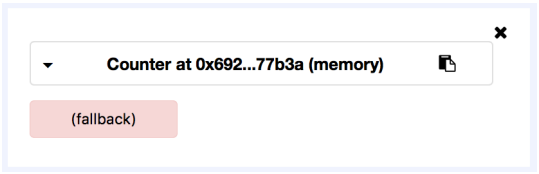


Figure 6.4: The Functions Which Can Be Called on the Flint Contract `Counter`, Without Using the Generated Interface.

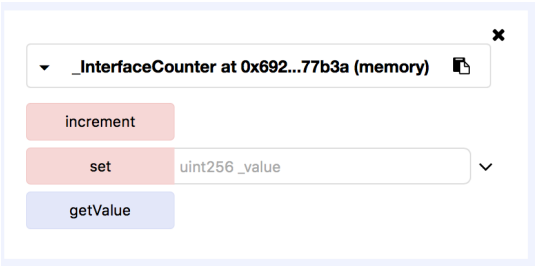


Figure 6.5: The Functions Which Can Be Called on the Flint Contract `Counter`, Using the Generated Interface.

```
33 // Solidity file
34 contract Counter {
35   function () public payable {
36     assembly {
37       switch flint$selector()
38       // rest of IR
39     }
40   }
41 }
```

Listing 6.10: Embedding the IR Code in a Solidity File

The generated Solidity file can serve as an input to the tools built for Solidity, such as analysers and IDEs like Remix [38]. It also allows developers to use testing frameworks such as Truffle [50], which we use to test Flint's generated bytecode (see section 6.13). However, as all the functions are accessed through the fallback function, their signatures are not visible at the Solidity level. For this reason, the compiler also produces a separate Solidity interface which includes the signatures of the contract's public functions. Solidity requires functions in interfaces to be marked `external`[1]. Non-mutating functions are marked as `view` in the Solidity signature. This allows tools to interpret Flint contracts through the generated interface. An example of a generated interface is shown in Listing 6.11.

```
42 interface _InterfaceCounter {
43   function getValue() view external returns (uint256 ret);
44   function increment() external;
45   function set(uint256 _value) external;
46 }
```

Listing 6.11: Generated Solidity Interface for a Flint Contract

When deploying the generated Solidity file without its generated interface, the Remix IDE [38] does not know which functions can be called on the contract apart from the fallback function, as shown in Figure 6.4. When interpreting a deployed contract using the generated interface, Remix displays the functions which can be called, as shown in Figure 6.5.

The full embedded IR code for Listing 6.5 is in Appendix F.

[1]We thank GitHub user `frogg` for implementing this.

6.11.3 Contract Initialisation

The contract initialiser IR code is generated separately from the rest of the function's contracts. This is because it cannot be placed in the fallback function of the Solidity file, as the fallback function is not called during contract Initialisation. Instead, we add a `constructor` function in the Solidity function and embed the code within it:

```
47 // Solidity file
48 contract Counter {
49   constructor() public {
50     assembly {
51       // Initialises value to 5, as specified on line 2 of the Flint contract.
52       sstore(add(0, 0), 5)
53       // etc.
54     }
55   }
56
57   function () public payable {
58     assembly {
59       switch flint$selector()
60       // rest of IR
61     }
62   }
63 }
```

Listing 6.12: Embedding the IR in a Solidity file

In Solidity, functions which are defined in an `assembly` block are not accessible in other `assembly` blocks. To support calling functions within the contract initialiser, the generated function declarations for each struct and contract function are included in the Solidity constructor's `assembly` block as well. This duplication is necessary to make Flint contracts work with tools built for Solidity, and will not occur when the Solidity compiler supports compiling standalone IULIA files.

6.11.4 Alternative Intermediate Representations

Using an intermediate representation helped us avoid writing a bytecode generator. Even though emitting bytecode presents advantages such as supporting fine-tuned performance optimisations, emitting to an IR allows us to leverage tools built for the IR. Embedding the IR code into a Solidity file allows us to use the Solidity compiler, analysers, and IDEs, as shown in the previous section. If a better IR is developed in the future, we can reimplement the code generation portion of the compiler, without modifying anything else.

Another choice for an intermediate representation could have been LLL, or *Lisp-Like Language*, which was developed by the Ethereum Foundation before Solidity. LLL does not seem to prevent benefits over IULIA, and development for the LLL compiler seems to have stopped.

6.12 Command-line Tool

The command-line usage of `flintc` is shown in Listing 6.13. `flintc` is compatible on macOS and Linux operating systems.

```
Usage:
    $ flintc <input file>

Arguments:

    input file - The input file to compile.

Options:
    --emit-ir [default: false] - Emit the internal representation of the code.
    --emit-bytecode [default: false] - Emit the EVM bytecode representation of the code.
    --dump-ast [default: false] - Print the abstract syntax tree of the code.
    --verify [default: false] - Verify expected diagnostics were produced.
```

Listing 6.13: Command-line Usage of `flintc`

We cover the installation process for the Flint compiler in Appendix B.

6.13 Testing

To ensure the correctness of our implementation, we create an automated test suite for the different stages of our compiler and the produced bytecode. We also implement an automated deployment infrastructure for `flintc` binaries using GitHub Releases.

Our development workflow is described in Figure 6.6. We use Travis CI [51] to run our tests on remote Linux and macOS servers. To test the compiler, we implement tests verifying whether the AST produced by the parser is correct (see subsection 6.13.1), and tests to verify the compiler produces the valid warnings and errors (see subsection 6.13.2). In addition, we test the behaviour of the bytecode produced by the Flint compiler by running a local simulated Ethereum blockchain (see subsection 6.13.3).

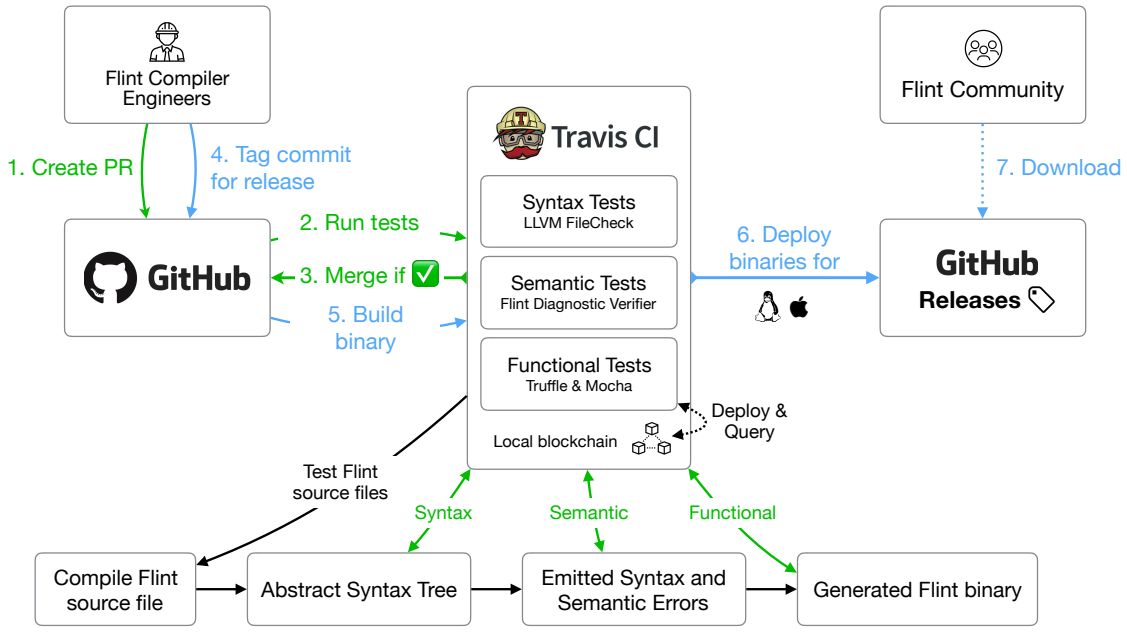


Figure 6.6: Overview of Flint's Continuous Integration infrastructure

Our syntax and semantic tests are available in the `Tests/ParserTests/` and `Tests/SemanticTests/` directories of the Flint project. Instead of maintaining a consolidated list of tests, we use the Lite [52] library to automatically find and run tests in the `Tests/` directory. Our functional tests are located in `Tests/BehaviorTests`.

6.13.1 Syntax Tests

To verify the correctness of the AST produced by the parser, we create an infrastructure similar to LLVM's [53] FileCheck [54], which compares the contents of two files using a flexible pattern matcher. We use the `--dump-ast` flag when calling `flintc` to output the produced AST of the input program as text, which we compare with the AST we expect. For each test file, we specify the expected AST nodes inline in the source, by using the `CHECK-AST` prefix. We provide an example in Listing 6.14. FileCheck checks whether the nodes of AST `flintc` produces for this program (ignoring comments) match the ones specified in the comments, in the same order.

```
1 // CHECK-AST: TopLevelModule
2 // CHECK-AST: TopLevelDeclaration
3 // CHECK-AST: ContractDeclaration
4 // CHECK-AST:   identifier "Test"
5 contract Test {
6
7 // CHECK-AST: VariableDeclaration
```

```
// CHECK-AST:   identifier "owner"
// CHECK-AST:   built-in type Address
  var owner: Address

// CHECK-AST: VariableDeclaration
// CHECK-AST:   identifier "arr"
// CHECK-AST:   FixedSizeArrayType
// CHECK-AST:     built-in type Int
// CHECK-AST:     size 4
// CHECK-AST:   ArrayLiteral
  var arr: Int[4] = []

// CHECK-AST: VariableDeclaration
// CHECK-AST:   identifier "arr2"
// CHECK-AST:   ArrayType
// CHECK-AST:     built-in type Int
// CHECK-AST:   ArrayLiteral
  var arr2: [Int] = []

// CHECK-AST: VariableDeclaration
// CHECK-AST:   identifier "numWrites"
// CHECK-AST:   built-in type Int
// CHECK-AST:   0
  var numWrites: Int = 0
}
```

Listing 6.14: Flint Syntax Test

We implement 8 syntax tests.

6.13.2 Semantic Tests

We also verify that the compiler produces the expected errors and warnings, and does not produce unexpected ones. We write test programs, and specify inline in the source file the expected diagnostics, using the `expected-error`, `expected-warning`, and `expected-note` prefixes. We provide an example in Listing 6.15. When `flintc` runs with the `--verify` flag, we check whether the expected diagnostics are produced at the lines where they are defined.

```
contract Constants {
  var a: Int // expected-note {{'a' is uninitialized}}
  var b: Int = "a" // expected-error {{Incompatible assignment between values of type 'Int' and 'String'}}
  let c: Int = 2 + 3
  let d: Int = 3
  let e: Int // expected-note {{'e' is uninitialized}}
}

Constants :: (any) {
  public init() {} // expected-error {{Return from initializer without initializing all properties}}

  mutating func foo() {
    let a: Int = 2 // expected-note {{'a' is declared here}}
    a = 3 // expected-error {{Cannot reassign to value: 'a' is a 'let' constant}}

```

```
    let b: Int = a
    self.a = 3

    if true {
      a = 5 // expected-error {{Cannot reassign to value: 'a' is a 'let' constant}}
    } else {
      a = 7 // expected-error {{Cannot reassign to value: 'a' is a 'let' constant}}
    }

    d = 4 // expected-error {{Cannot reassign to value: 'd' is a 'let' constant}}
  }
}
```

Listing 6.15: Flint Semantic Tests

We implement 16 semantic tests.

6.13.3 Functional Tests

We use the Truffle [50] library for Solidity smart contracts to test the behaviour of the bytecode the Flint compiler produces. We embed our generated IR code in a Solidity file, and use the generated Solidity interface (see subsection 6.11.2) for Truffle to interpret Flint contracts as Solidity contracts. We write our tests in JavaScript using the Web3 [33] library to call smart contract functions from Truffle. An example is given in Listing 6.16, in which we verify the functionality of a bank contract.

```
contract(config.contractName, function(accounts) {
  it("should correctly mint account1", async function() {
    const instance = await Contract.deployed();
    let t;

    await instance.mint(0, 20);

    t = await instance.get(0);
    assert.equal(t.valueOf(), 20);

    t = await instance.get(1);
    assert.equal(t.valueOf(), 0);
  });

  it("should transfer funds from account1 to account2", async function() {
    const instance = await Contract.deployed();
    let t;

    await instance.transfer1(0, 1, 5);

    t = await instance.get(0);
    assert.equal(t.valueOf(), 15);

    t = await instance.get(1);
    assert.equal(t.valueOf(), 5);
  });

  it("should transfer funds from account2 to account1", async function() {
    const instance = await Contract.deployed();
    let t;
```

```
31
32      await instance.transfer2(1, 0, 2);
33
34      t = await instance.get(0);
35      assert.equal(t.valueOf(), 17);
36
37      t = await instance.get(1);
38      assert.equal(t.valueOf(), 3);
39    });
40  });
```

Listing 6.16: Functional Test

We implement 11 functional tests.

6.13.4 Automated Deployments

We implement an infrastructure allowing us to easily deploy the latest version of `flintc` as a binary executable. Flint binary executables can be downloaded from `https://github.com/franklinsch/flint/releases`. To trigger a build, we push a git tag of the form `flint-$VERSION-snapshot-$DATE-$BUILD-$PLATFORM` on the master branch of Flint's GitHub repository, where `$VERSION` refers to the Flint compiler version (`0.1` at the moment), `$DATE` refers to the date of the tag creation, `$BUILD` refers to build number for the day (`a`, `b`, etc.), and `$PLATFORM` is `macos` or `linux`. We use a script to generate such tags automatically in `utils/tag_snapshot.sh`.

When tags are pushed, GitHub triggers a build on Travis CI, which runs `make release` to build a release version of the compiler. The binary executable for each platform, along with the standard library files, are made available as an archive on the GitHub release page.

6.14 Remarks

We have implemented a well designed and extensible compiler for Flint. We can compile a variety of versatile Flint programs, which we use in our extensive testing infrastructure which allows us to quickly iterate over compiler and language features confidently. Our AST Pass mechanism allows the easy addition of finer grain AST passes in the future, including different optimisation passes for code generation in the future. We provide an extensive suite of compiler diagnostics when performing semantic analysis and type checking.

Our generated intermediate representation code can be embedded in a Solidity file and Flint's ABI is compatible with Solidity's, allowing us to leverage the development and testing tools built for Solidity. In the future, we might consider developing our own intermediate representation for formal verification purposes, similar to Scilla [40].

Chapter 7

Future Implementation Work

Designing new features for programming languages require significant investigation work. It requires considering various scenarios and understanding the impact on existing features. In this chapter, we present features we have designed but not implemented due to time constraints.

7.1 Language Features

7.1.1 Type States

The first Parity Multi-sig wallet hack, described in 2.3.2, was the result of a function being called at the wrong time. Specifically, the `initWallet` function was supposed to be called exclusively during the "initialisation" phase of the smart contract. The developers resolved the issue by adding an `only_uninitialized` Solidity modifier to the function, which checks whether an owner has been set to the wallet. The "initialisation" phase is not explicitly encoded in the smart contract, as Solidity does not support any specific construct to represent such phases.

The programming language Bamboo [42] provides explicit syntax for encoding these phases, by regarding smart contracts as state machines. Flint has a mechanism to protect function calls from unauthorised users through caller capabilities, which we believe can be complemented using a *type state* [55] mechanism. The type state mechanism would protect functions from being called at the wrong time. Including such restrictions would be an opt-in feature for developers, and would use the same constructs as caller capabilities blocks.

We provide an example usage of this feature by defining an `Auction` smart contract, which allows users to bid for an item by sending Ether to a smart contract.

Specifying the contract's type states. An auction can be in three states: `Preparing`, `InProgress`, and `Terminated`. These are specified on line 1 and describe the state in which the contract is in. Line 6 defines a `state` property which records which state the contract is currently in. After initialisation, it is in state `Preparing`.

```
1 contract Auction (Preparing, InProgress, Terminated) {
2   var beneficiary: Address
3   var highestBidder: Address
4   var highestBid: Wei
5
```

```
6   var state: State = Preparing
7 }
```

Initialiser. The initialiser is called when the contract is deployed. Implicitly, it can only be called during the Preparing state, as it is the default state.

```
8  Auction :: caller <- (any) {
9    public init() {
10     self.beneficiary = caller
11     self.highestBidder = caller
12     self.highestBid = highestBid
13   }
14 }
```

Preparing the auction. The beneficiary can set a new beneficiary or open the auction when the contract is in the Preparing state.

```
15 Auction :: (beneficiary, Preparing) {
16   mutating func setBeneficiary(beneficiary: Address) {
17     self.beneficiary = beneficiary
18   }
19
20   mutating func openAuction() {
21     self.state = InProgress
22   }
23 }
```

Bidding and ending the auction. While the auction is in progress, anyone can bid, and the beneficiary can end the auction.

```
24 Auction :: (any, InProgress) {
25   @payable
26   mutating func bid(implicit value: Wei) {
27     // body
28   }
29 }
30
31 Auction :: (beneficiary, InProgress) {
32   mutating func endAuction() {
33     self.state = Terminated
34   }
35 }
```

Retrieving information. While the auction is in progress or has been terminated, anyone can get the highest bidder's address, and the highest bid.

```
36 Auction :: (any, InProgress, Terminated) {
37   func getHighestBidder() -> Address {
38     return highestBidder
39   }
40
41   func getHighestBid() -> Int {
42     return highestBid.getRawValue()
43   }
44 }
```

Collecting funds. After the auction has been terminated, the beneficiary can collect funds.

```
45 Auction :: (beneficiary, Terminated) {
46   mutating func collectFunds() {
47     send(beneficiary, &highestBid)
48   }
49 }
```

7.1.2 Bounded Loops

To aid with formal verification, we do not plan to support unbounded loops. A function call resulting in an infinite loop is aborted when the attached gas is exhausted. Furthermore, static analysers cannot always determine how many iterations the loop will cycle through.

Instead, we plan to add support for iterating over data structures, such as arrays or close ranges of numbers. As data structures cannot occupy infinite memory, loops are bounded by their size. An example is shown in Listing 7.1. The for-in loop allows to determine a bound to the computational cost of executing the function, in terms of the `elements` array's size. This important for estimating the gas cost of functions, as explained in 7.2.

```
1 func getSum() -> Int {
2   var sum = 0
3
4   for i in elements {
5     sum += i
6   }
7
8   return sum
9 }
```

Listing 7.1: Bounded Loops in Flint

7.1.3 External Function Calls

Flint smart contracts cannot currently call functions on other smart contracts. Flint will support calling into other Flint smart contracts, or Solidity contracts. For Flint contracts, the developer can import relevant source files to enable static checking of function calls. For Solidity contracts, the canonical function signature (see section 6.7.3) will have to be specified as a string. In Listing 7.2, we show how a bank contract could interact with the logging contract `Logger`. In both cases, checking whether the contract is allowed to perform the external call is the responsibility of the target contract.

```
1 Bank :: caller <- (any) {
2   func transferFunds(amount: Int, destination: Address, logger: Logger) {
3     // Transfer funds operation omitted.
4
5     // Log transaction.
6     logger.log(caller, destination, amount)
7   }
8 }
```

Listing 7.2: Flint External Call

7.1.4 Capability Functions

Currently, function declarations are protected by *caller capability groups*. The caller must have `any` of the caller capabilities in the group in order to call functions within the block. To allow more complex caller capability checks, we would like to support calling functions in caller capability groups. Such *capability functions* must not be `mutating`, take an `Address` as their only parameter, have a boolean return type, and be declared in an `(any)` caller capability block. We provide an example in Listing 7.3. The `isRichCustomer` function is used as a capability function, and checks whether the caller has a Bank account with more than 1000 Ether. The `richDeposit` function allocates a 1% interest every time a deposit is made.

```
1  Bank :: (any) {
2    func isRichCustomer(address: Address) -> Bool {
3      return balances.keys.contains(address) && balances[address] > 1000
4    }
5  }
6
7  Bank :: caller <- (isRichCustomer) {
8    @payable
9    mutating func richDeposit(implicit value: inout Wei) {
10     balances[caller].transfer(&value)
11
12     // Give the customer a 1% interest.
13     interests[caller].transfer(&interestPool, value.getRawValue() / 100)
14   }
15 }
```

Listing 7.3: Capability Functions

Static checks for internal calls would be also be performed if a function calls a function declared in the same caller capability group.

7.1.5 Attempt Function Calls

When performing internal function calls, it may not be statically guaranteed that the caller has the appropriate capability, such as in Listing 7.4. `bar` cannot be called from `foo`, as statically, we do not know whether the caller holds the `admin` capability.

```
1  contract C {
2    var admin: Address
3  }
4
5  C :: (any) {
6    // Initialiser omitted.
7
8    func foo() {
9      bar() // Invalid.
10   }
11 }
12
13 C :: (admin) {
14   func bar() {}
15 }
```

Listing 7.4: Insufficient Static Caller Capabilities

Attempting to call a protected function could be a useful feature. We propose the concept of *attempt calls*, which perform function calls by checking caller capabilities at runtime:

- `try? bar()`: The function bar's body is executed if at runtime, the caller's capability matches bar's. The expression `try? bar()` returns a boolean.
- `try! bar()`: If at runtime, the caller does not have the `admin` capability, an exception is thrown and the body doesn't get executed.

7.1.6 Late Assignment of Local Constants

Currently, a local let-constant needs to be assigned a value at the declaration site. However, we would like to improve upon this in the future by allowing let-constants to be initialised after they have been declared, but before they are accessed. This is particularly useful when a variable is assigned a value in different branches, such as in Listing 7.5.

```
1  func foo() -> Int {
2    let a: Int
3
4    if cond {
5      a = 1
6    } else {
7      a = 2
8    }
9
10   return a
11 }
```

Listing 7.5: Late Let-constant Initialisation

7.1.7 Other Improvements

In the future, we would like to bring other language and compiler improvements to Flint. An extensive list of issues can be found on GitHub [24].

Language Features

We discussed new language features previously (supporting traits and safe external calls), and would also like to implement new ones, such as supporting fallback functions, user-defined exception, and more.

Compiler Features

In terms of compiler features, we are planning to bring improvements to the semantic analyser, such as producing warnings when variables are unused or can be made let-constants. We would also like to optimise runtime memory usage, for example, by making the default integer size smaller (it is currently 256 bits) to store multiple integers in a single EVM word. This would result in lower gas costs.

7.2 Gas Estimation

Flint was built with formal verifiability in mind. We would also like to build tools to precisely determine the gas costs required to call functions in a smart contract. By knowing the gas cost for each EVM instruction, and disallowing unpredictable patterns such as infinite loops, we believe we can compute the gas cost for each function in terms of each function's arguments and the state properties it uses. For example, for the example below, we could estimate `distribute` to require `420 + beneficiaries.count * 1500` gas.

```
1  contract Distribution {
2    var admin: Address
3    var total: Wei
4  }
5
6  Distribution :: (admin) {
7    mutating func distribute(beneficiaries: [Account]) {
8      let amount = total.getRawValue() / beneficiaries.count
9
10     for beneficiary in beneficiaries {
11       let payout = Wei(&total, amount)
12       send(beneficiary, &payout)
13     }
14   }
15 }
```

7.3 Flint Package Manager

The Flint Package Manager allows developers to share and use Flint contracts. We aim to implement a Package Manager smart contract to record package information, including hashes of the source code and security warnings produced by the Flint compiler. The original source code of packages would be stored in a traditional database, as storing large files on Ethereum is very costly. The Flint compiler would verify the integrity of downloaded packages by computing a hash of the package, and comparing it against the value stored in the Package Manager contract. Developers can use packages to use library code from other contracts, and safely interact with deployed Flint contracts. Storing security warnings about packages allows developers to know whether a package is safe to use. These warnings cannot be overridden by Ethereum users, and can only be set when uploading a Flint package.

In addition, we aim for the package manager to be the central location for the source code of all deployed Flint contracts. As only bytecode is maintained by miners, users would use the Flint Package Manager to Flint source code of contracts.

We provide a prototype of the Flint Package Manager smart contract in Appendix D.

7.4 Remarks

Programming languages take years to develop. Determining which features to implement is not straightforward, and requires careful analysis. We designed extensions to Flint we think would be a natural fit. Type states allow regarding a smart contract as a state machine, hence facilitating reasoning by better grouping state. Capability functions implement a more

versatile mechanism for disallowing unauthorised accesses to functions. In terms of tooling, gas estimation and the Flint Package Manager will be essential elements in the Flint toolchain, making the use of Flint even more compelling.

Chapter 8

Evaluation

In this chapter, we show how Flint allows the writing of safe and expressive programs. We evaluate the language from a qualitative and quantitative perspective. We compare Flint to Solidity, as it is the most used programming language to write Ethereum smart contracts, and in some cases to Vyper, an emerging new language developed by the creators of Ethereum.

We translate Solidity and Vyper contracts to Flint, allowing us to see code differences, observe issues raised by various analysers, and compare runtime performance. We were able to use the same analysers by embedding Flint's IR code in a Solidity file[1].

The examples we choose highlight safety and performance differences between Solidity, Vyper, and Flint programs. We also see how Flint helps prevent TheDAO (see subsection 2.3.1) and the Proof of Weak Hands Coin (see subsection 2.3.5) vulnerabilities.

The full programs in each language are in Appendix G.

8.1 Performance and Programming Style

We evaluate the performance of Flint's caller capabilities, Asset types, and safe arithmetic operations. We also observe differences in code conciseness when using Asset types.

8.1.1 Caller Capabilities

We write equivalent Flint, Solidity, and Vyper smart contracts to compare the differences in runtime performance when using caller capabilities. We define simple functions which can be called by any user, some by `owner`, and some by any customer in the `customers` array. In Solidity, we implement two modifiers, `onlyOwner` and `anyCustomer`, which check whether the caller is `owner` or is in the `customers` array, respectively. Vyper does not support a modifier mechanism. Instead, we include assertions at the beginning of relevant function bodies.

[1] As these were built with Solidity in mind, they did not present high code coverage results when running with Flint code. This is because they rely on Solidity's mechanism to determine which function body to execute to discover the contract's functions. As Flint's mechanism is slightly different, analysers do not always discover all the executable branches of the program. We asked the developers of the analysers to enable integration with Flint, but the work has not been completed yet.

We expect similar gas costs for Solidity, Vyper, and Flint for simple functions with no additional internal calls, as the runtime behaviour of the languages is similar. We also note that Flint's and Vyper's safe arithmetic operations, in this case *addition*, uses more gas than the regular addition operation Solidity provides. The full example is available in section G.1. Flint does not require writing modifiers, like in Solidity:

Solidity

```
modifier onlyOwner {
  require(msg.sender == owner);
  _;
}

modifier anyCustomer {
  uint numCustomers = customers.length;
  bool found = false;

  for (uint i = 0; i < numCustomers; i++) {
    if (customers[i] == msg.sender) { found = true; }
  }

  require(found);
  _;
}
```

Code organisation. The organisation of the smart contracts present notable differences. In Solidity and Vyper, the state and the functions are all defined at the top level of the contract. Solidity does not enforce including the user-defined modifiers we have specified in some function signatures, while Vyper does not support a modifier mechanism (we insert assertions at the beginning of the function bodies).

Solidity

```
contract Caps {
  address owner;
  address[] customers;
  uint256 counter;
  modifier anyOwner {...}
  modifier anyCustomer {...}
  function anyUser() public constant returns(uint256) {...}
  function ownerCall(uint counter) public constant onlyOwner returns(uint256) {...}
  function customerCall(uint counter) public constant anyCustomer returns(uint256) {...}
  function ownerInternalCalls() public onlyOwner {...}
  function customerInternalCalls() public anyCustomer {...}
}
```

Vyper

```
owner: public(address)
customers: public(address[50])
counter: public(uint256)
@public @constant def anyUser() -> uint256
@private @constant def isCustomer(a: address) -> bool
@public @constant def customerCall(_counter: uint256) -> uint256
@public @constant def ownerCall(_counter: uint256) -> uint256
@public def ownerInternalCalls()
@public def customerInternalCalls()
```

Flint

```
contract Caps {
  var owner: Address
  var customers: [Address]
  var numCustomers: Int
  var counter: Int
}

Caps :: owner <- (any) {
  mutating public func addCustomer(customer: Address) {...}
  public func anyUser() -> Int {...}
}

Caps :: (owner) {
  public func ownerCall(counter: Int) -> Int {...}
  public mutating func ownerInternalCalls() {...}
}

Caps :: (customers) {
  public func customerCall(counter: Int) -> Int {...}
  public mutating func customerInternalCalls() {...}
}
```

Analysis

We compare the result of the analyses performed by Oyente and Mythril for Solidity. Unfortunately, we were not able to find an analyser for Vyper. For the Solidity file, Oyente reported a code coverage of 47.5%, and 33.7% for the Flint contract.

Issue	Solidity	Flint	Details
Integer Overflow	Yes	No	Both analysis tools detect that if the `customers` array becomes too large, its length will overflow in Solidity. This is an unlikely situation, but nonetheless not possible in Flint thanks to the overflow protecting operators.

Table 8.1: Analysis Results for `CallerCaps`

Performance

We compare the gas costs of executing each function in Table 8.2 in the Solidity, Vyper, and Flint contracts. The Solidity and Flint gas costs have been retrieved from executing the calls on our simulated Ethereum network, while the Vyper gas costs are estimates given by the Vyper compiler. The tool to run Vyper code does not support obtaining the number of gas consumed after calling a function. Overall, we observe that Flint is significantly faster when functions perform internal calls to functions which require the same caller capability. This is expected, as the caller capability check is only executed once.

Operation	Solidity	Vyper (estimate)	Flint	Difference (Solidity/Vyper vs Flint)	Possible Explanation
Deploying	438121	-	514268	S: -14.8%	The Solidity optimiser produces a smaller binary.
Any call	268	193	283	S: -5% V: -32%	Determining which function was called is slightly faster in Flint.
Owner call	633	655	836	S: -24% V: -22%	The dynamic check for a single caller capability is slightly slower in Flint.
Owner many internal calls	32662	171127	28673	S: +14% V: +496%	Only one dynamic caller capability check is performed in Flint. The gas cost would be marginally smaller if we used the `&+` operator, which provides unsafe overflowing semantics for addition.
Customer call (with 5 entries in `customers`)	3791	4401	1707	S: +122% V: +158%	Flint's array iteration algorithm is better suited for small arrays.
Customer call (with 20 entries in `customers`)	13136	11286	18047	S: -27% V: -37%	Solidity's array iteration algorithm is better suited for larger arrays.
Customer call (with 50 entries in `customers`)	31434	24996	43847	S: -28% V: -43%	Same as above.
Customer many internal calls (with 5 entries in `customers`)	168804	635958	54426	S: +210% V: +1068%	Only one dynamic caller capability check is performed in Flint.
Customer many internal calls (with 20 entries in `customers`)	510129	1634283	67326	S: +658% V: +2327%	Same as above.

Customer many internal calls (with 50 entries in `customers`)	1192779	3624393	93126	S: +1181% V: +3792%	Same as above.

Table 8.2: Gas Costs for `CallerCaps`. S and V refer to the gas cost differences between Solidity and Vyper.

Conclusion

We observe the effect of using caller capabilities in Flint. The gas costs are similar for simple examples using a caller capability backed by a single address. When calling a function which calls multiple functions, requiring the same caller capability, Flint is up to 12 times faster than Solidity, and 38 times faster than Vyper.

For caller capabilities backed by an array, the results vary. Flint code is cheaper to run when the array is relatively small, but becomes more expensive when it is large. We believe Solidity has a more efficient scheme for iterating over arrays. We are planning to optimise Flint's array iteration scheme. In the example which performs a large number of internal calls, the Flint code is up to two times faster. This is possible as Flint performs a single runtime check, whereas Solidity and Vyper perform one per function call.

8.1.2 Asset Types and Safe Arithmetic Operations

We define a smart contract to assess the safety and performance of Flint's Asset operations. The `Bank` smart contract uses Asset types, namely the Wei type. Customers can send Ether to the contract, and `Bank` keeps track of how much each customer has sent. Customers can then withdraw their Ether, or transfer it to another Bank account. This smart contract should allow us to assess whether the Flint Asset operations are safe, and how much additional gas they incur. We have not translated this contract to Vyper, as it does not present major code design and runtime behaviour differences with its Solidity counterpart.

Depositing funds. The `deposit` function of both smart contracts are similar, and record received Ether to state. Flint however encodes Wei in the `Wei` type, rather than as an integer.

Solidity

```
function deposit() anyCustomer public payable {
  balances[msg.sender] += msg.value;
}
```

Flint

```
@payable
public mutating func deposit(implicit value: Wei) {
  balances[account].transfer(&value)
}
```

Transferring funds internally. Using Flint's Wei functions allows safer and more concise code. In the Solidity contract, the `transfer` operation requires three lines of code, whereas the Flint version requires one.

Solidity

```
function transfer(uint amount, address destination) anyCustomer public {
  require(balances[msg.sender] >= amount);
  balances[destination] += amount;
  balances[msg.sender] -= amount;
}
```

Flint

```
public mutating func transfer(amount: Int, destination: Address) {
  balances[destination].transfer(&balances[account], amount)
}
```

Withdrawing funds. The `withdraw` function is one line shorter in Flint as well. Essentially, we do not need to check whether the account holder has enough funds to perform an operation, as an exception is thrown if the result of an arithmetic operation overflows (wraps around from 0 to the maximum 256-bit value, or from the maximum 256-bit value to 0). When writing the `transfer` function, which transfers Wei from the caller's account to another account, we initially forget to check whether the caller had enough funds to transfer the amount they requested. Thus, when a caller requested to transfer an amount larger than they had, their account balance would overflow and become a number close to the maximum 256-bit number. This is an issue as they would be able to call `withdraw` to transfer all the Wei of the smart contract to their Ethereum account.

Solidity

```
function withdraw(uint amount) anyCustomer public {
  require(balances[msg.sender] >= amount);
  balances[msg.sender] -= amount;
  msg.sender.transfer(amount);
}
```

Flint

```
public mutating func withdraw(amount: Int) {
  // Transfer some Wei from balances[account] into a local variable.
  let w: Wei = Wei(&balances[account], amount)

  // Send the amount back to the Ethereum user.
  send(account, &w)
}
```

We also note that in Solidity, there is no built-in Wei type for representing currency. It is easy to accidentally add currency to an account, or forget to do so. For example, when writing `withdraw`, we could have forgotten to decrease Ether from the caller's account before sending the amount to their Ethereum account. This issue is not possible by using `Wei`'s initialiser which atomically transfers a subset of another Wei variable to the receiver, which we use to send the money.

We note that in Flint, we do not need to define our own `onlyManager` and `anyCustomer` to control access to functions.

Analysis

We compare the result of the analyses performed by Oyente and Mythril. For the Solidity file, Oyente reported a code coverage of 91.3%, and 64.7% for the Flint contract.

Issue	Solidity	Flint	Details
Integer Overflow	Yes (5 cases)	No	Both analysis tools find 4 potential integer overflows on lines 30 (the array can become too large), and lines 34, 43, and 48 (the integer value in the `balances` mapping can become too large).

Table 8.3: Analysis Results for `CallerCaps`

Performance

Operation	Solidity	Flint	Difference	Possible Explanation
Deploying	422589	415901	+1.6%	Solidity and Flint produce similarly sized binaries.
Register	40741	61528	-34%	Solidity is more efficient at adding elements to an array.
Deposit 10 Ether	21460	24002	-11%	This is because of the overhead of Flint's safe Asset operations, which prevent overflows and state inconsistencies.
Transfer 80 Wei	27200	30685	-11%	Same as above.
Withdraw 5 Ether	14301	17245	-17%	Same as above.
Mint 100 Wei	23098	20867	+10%	Similar results.

Table 8.4: Gas Costs for `Bank`

Conclusion

The Flint version of the smart contract is marginally more expensive to run in terms of gas costs. However, using the built-in Asset type Wei allow the use of safe currency transfer operations. This is confirmed by the analysers, which highlight integer overflow vulnerabilities for the Solidity contract, but none for the Flint version.

8.1.3 Auction

We translate a popular Solidity smart contract, `Auction`, in Flint and Vyper. This contract is used as a main example in the Solidity documentation [5]. We look for differences in the conciseness of code between the three languages, and try to find vulnerabilities in the Solidity contract Flint is immune to. Unfortunately, we could not analyse Vyper code as we did not

have access to any compatible analysis tool. This smart contract showcases the use of Asset types and caller capabilities in Flint.

Bidding. Ethereum users send Ether to the auction smart contract in order to bid. If their bid is the highest, it is recorded. Otherwise, the transaction is aborted.

Solidity

```
1  function bid() public payable {
2    require(now <= auctionEnd);
3    require(msg.value > highestBid);
4    if (highestBid != 0) {
5        pendingReturns[highestBidder] += highestBid;
6    }
7    highestBidder = msg.sender;
8    highestBid = msg.value;
9    emit HighestBidIncreased(msg.sender, msg.value);
10 }
```

Vyper

```
1  @public
2  @payable
3  def bid():
4      assert block.timestamp < self.auction_end
5      assert msg.value > self.highest_bid
6      if not self.highest_bid == 0:
7          self.pending_returns[highest_bidder] += highest_bid
8
9      self.highest_bidder = msg.sender
10     self.highest_bid = msg.value
11     log.Transfer(HighestBidIncrease(msg.sender, msg.value))
```

Flint

```
1  @payable
2  public mutating func bid(implicit value: Wei) {
3    assert(hasAuctionEnded == false)  // The unary not (!) operator is not implemented yet.
4    assert(value.getRawValue() > highestBid.getRawValue())
5    if highestBid.getRawValue() != 0 {
6      pendingReturns[highestBidder].transfer(&highestBid)
7    }
8    highestBidder = caller
9    highestBid.transfer(&value)
10   highestBidDidIncrease(caller, value.getRawValue())
11 }
```

The `bid` function is implemented similarly in all three languages. The Solidity contract is vulnerable to an integer overflow on line 5, as shown by the analysers' findings in the table below. This is however unlikely to happen as it would require the caller to send a very large amount of Ether to the transaction.

The use of a `pendingReturns` mapping may seem unnecessary. When a user is outbid, their bid in Ether is not immediately sent back to their account. Instead, the amount of Ether the contract owes each bidder is stored in the `pendingReturns` mapping. The author of the original Solidity contract argues this is a safer approach as it avoid executing code in another smart contract, hence avoiding reentrancy issues which would allow a bidder to retrieve more money than they are owed. To the extent of our knowledge, this particular program is protected

against such issues, and the indirection was included for teaching purposes. We also include it in our Flint contract and in the Vyper contract, even though we could simply not use the `pendingReturns` dictionary and replace line 6 in the Flint contract by `send(highestBidder, &highestBid)`.

Ending the auction. The original Solidity contract ends the auction at a given time. The global variable `now` represents a UNIX time (seconds since 1970). We argue relying on time when writing smart contracts is not a good approach. Both analysis tools Mythril [12] and Oyente [11] recommend against using this approach. In general, Flint does not surface such unsafe information to the developer by design. The time given by the `now` variable is the UNIX time of the miner systems. We cannot assume the time set on these computers are synchronised, or even similar. A large number of miners recording a wrong time can alter the expected behaviour of the contract significantly. In this case, time is used to check whether the auction is still running. The assertion on line 66 might fail for some miners, and not for others, resulting in each miner having an inconsistent state of the smart contract. We argue making the beneficiary call an `endAuction` function is a better approach, as it does rely on the assumption that the miners' time is synchronised. We implement the Vyper contract with the time dependency, as this is how the authors of Vyper implemented it[1].

Solidity

```
1 function auctionEnd() public {
2   require(now >= auctionEnd, "Auction not yet ended.");
3   require(!ended, "auctionEnd has already been called.");
4   ended = true;
5   emit AuctionEnded(highestBidder, highestBid);
6   beneficiary.transfer(highestBid);
7 }
```

Vyper

```
1 @public
2 def end_auction():
3     assert block.timestamp >= self.auction_end
4     assert not self.ended
5     self.ended = True
6     log.AuctionEnded(HighestBidIncrease(highestBidder, highestBid))
7     send(self.beneficiary, self.highest_bid)
```

Flint

```
1 SimpleAuction :: (beneficiary) {
2   public mutating func endAuction() {
3     assert(hasAuctionEnded == false)
4     hasAuctionEnded = true
5     auctionDidEnd(highestBidder, highestBid.getRawValue())
6     send(beneficiary, &highestBid)
7   }
8 }
```

Withdrawing Ether. The beneficiary can transfer the highest bid once the auction has ended. In Solidity the `send` function on line 6 returns a boolean indicating whether the transfer was successful. In Vyper and Flint, we throw an exception instead.

Solidity

[1] https://github.com/ethereum/vyper/blob/master/examples/auctions/simple_open_auction.v.py

```
function withdraw() public returns (bool) {
  uint amount = pendingReturns[msg.sender];
  if (amount > 0) {
    pendingReturns[msg.sender] = 0;

    if (!msg.sender.send(amount)) {
        pendingReturns[msg.sender] = amount;
        return false;
     }
   }
   return true;
}
```

Vyper

```
@public
def withdraw():
  uint256 amount = pending_returns[msg.sender]
  if amount > 0:
    pending_returns[msg.sender] = 0
    send(msg.sender, amount)
```

Flint

```
mutating func withdraw() {
  var amount: Wei = Wei(&pendingReturns[caller])
  assert(amount.getRawValue() != 0)
  send(caller, &amount)
}
```

The `withdraw` function is significantly shorter in Flint. In Solidity, the amount owed is set to 0 on line 4, sent on line 6, and potential failures are handled until line 8. We have similar code in Vyper. In Flint, these operations are performed atomically in a single statement, on line 3. The `endAuction` in Flint is protected by the `beneficiary` caller capability, allowing only the beneficiary to end the auction.

Analysis

We compare the result of the analyses performed by Oyente and Mythril. For the Solidity file, Oyente reported a code coverage of 95%, and 51.1% for the Flint contract.

Issue	Solidity	Flint	Details
Timestamp Dependency	Yes	No	Both Oyente and Mythril warn that line 66 of the Solidity contract (`require(now >= auctionEnd)`) rely on the environment information.
Integer Overflow	Yes	No	Both analysis tools detect that line 37 of the Solidity contract (`pendingReturns[highestBidder] += highestBid`) can result in an integer overflow.
Transaction-Ordering Dependence (TOD)	Yes	Yes	Both analysis tools detect a TOD on lines 74 in the Solidity contract and 63 in Flint. Specifically, if a bid is placed around the same time `endAuction` is called, the bid amount sent to the beneficiary may be incorrect.

Table 8.5: Analysis Results for `Auction`

Performance

We compare the gas costs of executing each function in Table 8.6. Overall, we do not observe significant differences between the execution costs of the contracts.

Operation	Solidity	Flint	Difference	Possible Explanation
Deploying	319922	377976	-15%	The Solidity optimiser produces a smaller binary.
Bid 100 Wei	41076	31174	+32%	
Withdraw	13210	15984	-17%	This due to the overhead of Flint's safe Ether transfer operations.
End auction	23949	37383	-36%	This might be due to the overhead of checking caller capabilities dynamically.
Get Highest Bid	504	660	-24%	Similar results.
Get Highest Bidder	530	443	+20%	Similar results.

Table 8.6: Gas Costs for `Auction`

Conclusion

The Vyper and Solidity code is similar. The Flint and Vyper versions are not vulnerable to integer overflows as all arithmetic operations in those languages throw an exception when overflows occur by default. In Flint, the withdraw function is more concise and performs the Ether transfer atomically.

In Flint, we avoided the timestamp dependency, as the language does not surface unsafe information to the developer. Gas costs are on average higher on Flint, which is due to the overhead incurred by using safe operators.

8.2 Preventing Vulnerabilities

In this section, we show how TheDAO and the Proof of Weak Hands Coin vulnerabilities could have been prevented using Flint. We also look at how a programmer could bypass Flint's security features.

8.2.1 Preventing TheDAO Vulnerability

We show how using Flint for writing TheDAO contract could have helped prevent the loss of 3.6M+ Ether (880 million dollars). We focus on the vulnerable portion of the code, as

presented in 2.3.1. The attack relies on a reentrant call, which drains the contract's Ether balance.

Solidity

```
contract TheDAO {
  mapping(address => uint256) public balances;

  function TheDAO() public payable {}

  function deposit(address recipient) public payable {
    balances[recipient] += msg.value;
  }

  function withdraw(address recipient) public {
    uint256 balance = balances[recipient];
    recipient.call.value(balance)();
    balances[recipient] = 0;
  }
}
```

Listing 8.1: Code Reentrancy Vulnerability in Solidity

Flint

```
contract TheDAO {
  var balances: [Address: Wei] = [:]
}

TheDAO :: (any) {
  public init() {}

  @payable
  public mutating func deposit(implicit value: Wei, account: Address) {
    balances[account].transfer(&value)
  }

 mutating public func withdraw(account: Address) {
    send(account, &balances[account])
 }
}
```

Performing the Attack

We implement an Attacker contract in Solidity, as it requires calling a low-level function performing an external call without propagating exceptions thrown by the external call. The Attacker contract can also target the Flint smart contract, as Flint is interoperable with Solidity.

```
// This contract will be used to attempt an attack on the Flint contract as well.
contract Attacker {
    uint256 public total;
    function () public payable {
        total += msg.value;
        msg.sender.call(bytes4(keccak256("withdraw(address)")), this);
    }
}
```

We perform the following sequence of steps in both smart contracts.

1. Alice calls `deposit`, with arguments 10 Ether and her Ethereum address.
2. Eve, the attacker, calls `deposit`, with 1 Ether and the Ethereum address of the Attacker contract as arguments.
3. Eve calls `withdraw`, with the Attacker contract's address as an argument.

Results

In Solidity, `balances[account]` is never set to 0, as the attacker's code calls back into `TheDAO` contract until all gas has been consumed. Funds are sent every time `withdraw` is called. In Flint, the `transfer` operation atomically updates `balances[account]` and transfers the funds out of the smart contract. 1 Ether is sent (what the attacker deposited) the first time `withdraw` is called, and 0 Ether is sent for the subsequent calls.

Assuming that the `balances` mapping was empty before the steps were followed, and that Alice had 10 Ether and exactly enough to pay for gas costs to call `deposit`, we observe the differences in the resulting state.

Value	Solidity	Flint
Attacker's Ether balance	11 Ether	1 Ether
TheDAO's Ether balance	0 Ether	10 Ether
Alice's Ether balance	0 Ether	0 Ether
`balances[Alice]`	10 Ether	10 Ether
`balances[Attacker]`	0 Ether	0 Ether

Table 8.7: State After Performing TheDAO Attack

8.2.2 Preventing the Proof of Weak Hands Coin Vulnerability

We show how we can write the Proof of Weak Hands Coin smart contract using Flint's Asset types to prevent vulnerabilities.

We reimplement three functions in Flint. The `allowance` mapping (type `address => (address => uint256)`) in the Solidity contract refers to how much a user has authorised another user to retrieve from their account. When a user decides to retrieve an allowed amount, the smart contract checks whether the account the money is being retrieved from has sufficient funds. If it does not, an exception is thrown. Under this scheme, it is therefore possible for a user to allow others to retrieve more money than they actually have. We believe this is not a good scheme, we think allowing an individual to retrieve Ether is an operation which should always succeed. In our implementation, we implement a `Coin` struct which implements the `Asset` trait, and use its transfer operations. When creating an allowance, we *transfer* Coin from the `balances` mapping to the `allowance` mapping. This ensures the total supply of Coin remains does not change when allocating Coins to others. However, users might want to share an allocation among a group of people: even though a user might only have 3 Ether, they could allow four users to each collect 1 Ether. Then, an exception would be thrown when the fourth user attempts to perform the retrieval. This behaviour can still be achieved our

implementation, by modifying the type of `allowances` to `[Address: [[Address]: Coin]]`. This would allow users to allocate an amount of Ether to a group of users.

Retrieving from an allowance. We implement the `transferFrom` function, which retrieves an allowance and calls `transferTokens`. If the caller attempts to retrieve a larger amount than what allocation permits, an exception is thrown. In Flint, we use the Coin initialiser which takes an integer value, in order to retrieve a subset of the allowance. The `allowance` dictionary is updated in the same operation. We pass the retrieved Coin by reference to `transferTokens` (Assets cannot be passed by value).

Solidity

```
function transferFrom(address _from, address _to, uint256 _value)  public {
    var _allowance = allowance[_from][msg.sender];
    if (_allowance < _value)
        revert();
    allowance[_from][msg.sender] = _allowance - _value;
    transferTokens(_from, _to, _value);
}
```

Flint

```
Coin :: caller <- (any) {
  public mutating func transferFrom(from: Address, to: Address, value: Int) {
    var allowance = Coin(&allowance[from][to], value)
    transferTokens(from, to, &allowance)
  }
}
```

Transferring tokens. When calling `transferTokens` with the contract's address as the `from` address, the contract calls `sell`. We focus on this function as it presented a vulnerability in the Solidity contract. As discussed above, the initial check which verifies whether the `from` account has enough funds to perform the transfer is not needed. When creating an allowance, the `Wei` is transferred from `balanceOfOld` to `allowances`.

Solidity

```
function transferTokens(address _from, address _to, uint256 _value) internal {
    if (balanceOfOld[_from] < _value)
        revert();
    if (_to == address(this)) {
        sell(_value);
    } else {
        // Omitted, as the code was not vulnerable.
    }
}
```

Flint

```
mutating func transferTokens(from: Address, to: Address, value: inout Coin) {
  if (to == contractAddress()) {
    sell(&value)
  } else {
    // Omitted, as the Solidity code was not vulnerable.
  }
}
```

Listing 8.2: transferFrom, in Flint

Selling coins for Ether. We use a user-defined initialiser of Wei which converts a Coin to Wei, and send it using `send`. Line 5 of the Solidity contract is vulnerable, as it allows for integer overflow. Updating the `balanceOfOld` entry is not needed in Flint, as the Coin which we are sending had already been removed when creating the allocation.

```
1 function sell(uint256 amount) internal {
2     var numEthers = getEtherForTokens(amount);
3     // remove tokens
4     totalSupply -= amount;
5     balanceOfOld[msg.sender] -= amount;
6
7     // Send Ether.
8 }
```

Listing 8.3: sell, in Solidity

```
1  Coin :: caller <- (any) {
2    mutating func sell(amount: inout Coin) {
3      var value: Ether = getEtherForTokens(coin.getRawValue())
4      // remove tokens
5      totalSupply -= amount.getRawValue()
6
7      // No need to remove from 'balanceOfOld', as the Coin was transferred to
8      // the 'allowances' mapping.
9
10     send(caller, &value)
11   }
12 }
```

Listing 8.4: sell, in Flint

8.2.3 Bypassing Flint's Safety Features

Caller Capabilities

Flint's caller capability feature can be bypassed entirely by declaring all functions in a caller capability block protected by the `any` caller capability. However, as Flint requires users to specify which caller capabilities are needed to call a function, we argue this safety mechanism cannot be bypassed accidentally.

Asset Types

It is still possible to represent currency as integers in Flint, hence ignoring the safe transfer operations we provide, as shown in Listing 8.5. As the only way for a contract to receive Ether is through `@payable` functions, an explicit conversion from `Wei` to `Int` must still occur. However, the compiler would still produce a warning as the asset is not transferred in the function body's scope.

```
1 contract Wallet {
2   var balance: Int = 0
3 }
4
5 Wallet :: (any) {
6   @payable
```

```
7     mutating public func deposit(implicit value: inout Wei) {
8       balance += value.getRawValue()
9     }
10  }
```

Listing 8.5: Bypassing Asset Types

Creating, destroying, and duplicated Assets is still possible by using privileged operations such as the unsafe `Wei` initialiser which constructs a `Wei` from an integer.

Although warnings can be ignored, we think bypassing Flint's safe transfer operations for currency cannot be accidental.

Safe Arithmetic Operations

Flint provides the `&+`, `&-`, and `&*` operators which do not cause exceptions when overflows occur. These operators should be rarely used, and can be easily spotted by readers of a smart contract.

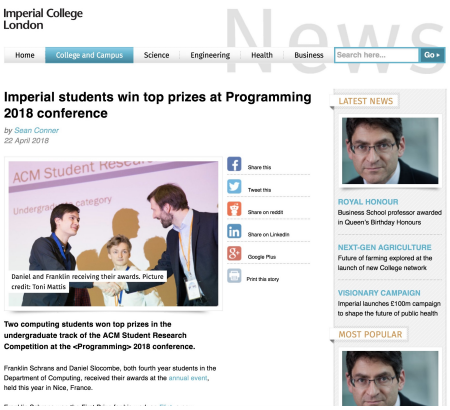
8.3 Community Feedback

Flint was made open source in April 2018. Since then, we have presented Flint at a programming languages conference, and were honoured to be awarded a prize. We also received great feedback from the Ethereum Community, and presented Flint at the Imperial Blockchain Forum.

8.3.1 Publications and Awards

We presented Flint at the *International Conference on the Art, Science, and Engineering of Programming* in Nice, France, and published *Writing Safe Smart Contracts in Flint* [25]. The feedback was very positive, and Flint won the First Prize in the Undergraduate track of the ACM Student Research Competition, and was selected to participate in the ACM Grand Finals (Figure 8.1).

(a) Presenting Flint



(b) Imperial College News

Figure 8.1: Flint at <Programming> 2018

8.3.2 Ethereum Community

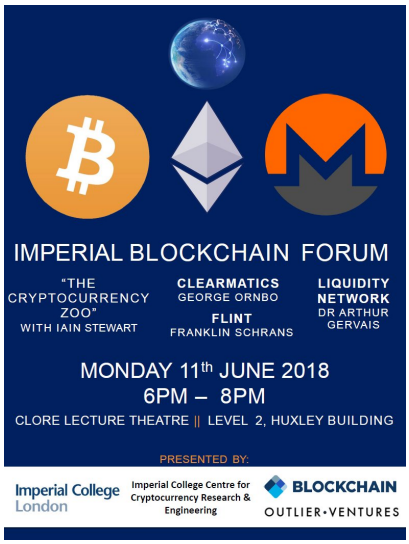
The feedback from the online community was very positive. We published an article, *Flint: A New Language for Safe Smart Contracts on Ethereum* [26], which accumulated over a thousand "claps" on the web platform Medium. Flint has also been discussed about on Twitter, Reddit, and in email newsletters. The Flint GitHub repository [24] has over 140 "stars" and is among the top 30 GitHub projects about smart contract programming.

We made the Flint project open source in April 2018, under the MIT copyright license. Since then, its safety-focused features and ease of use were praised by the Ethereum community. Websites such as DApps Weekly [56] described Flint as being "on its way to filling a sorely needed gap in the developer tooling space". Reddit user Chugwig stated: "I haven't been a big fan of the Solidity alternatives [...], this is the first one to really catch my interest."

The Ethereum Foundation (the original developers of Ethereum) have awarded a ten thousand dollar grant for us to pursue the development of Flint[1].

Ethereum developers have also started experimenting with programming in Flint. Nick Doiron published an article, *Modeling a mudaraba smart contract* [27], in which he explains how he implemented a mudaraba, a type of contract in islamic finance, in Flint. He followed the exact sequence of steps Flint invites him to follow: declaring state, then creating caller capability blocks, and finally defining functions.

We were also honoured to present Flint at the Imperial Blockchain Forum alongside blockchain experts (Figure 8.2).



(a) Imperial Blockchain Forum Poster

(b) Presenting Flint in Huxley Building

Figure 8.2: Imperial Blockchain Forum, June 2018

8.4 Conclusion

Flint makes it easy to write smart contracts which are are safe by design. We did not simply port the features of other programming languages, such as unbounded loops, as they were not always a good fit for the Ethereum programming model. Instead, we have focused on

[1] https://blog.ethereum.org/2018/05/02/announcing-may-2018-cohort-ef-grants/

novel features which allowed writing safe, concise programs. Our additional safety mechanisms prevent common vulnerabilities of Solidity programs, while delivering similar or better performance[1]. In addition, our extensible language design allows us to introduce new safety features in the next version of Flint.

[1] Apart from iterating over large arrays, which we can address at a later stage.

Chapter 9

Conclusion

The emergence of the Internet services has shaped new ways for us to connect and share. We trust that most of the services we use will behave in the way we expect them to. The Ethereum platform seeks to eliminate the need for trust, by executing services without the need for a central authority. It introduces a new programming model which presents challenges linked to its very nature: users trust the code, not the developer. The code of a smart contract cannot be changed once deployed. Hence the developer must be certain of the correctness of the program's behaviour. However, in many cases, unintentional bugs in smart contracts have led to significant currency losses.

Modern programming languages for traditional computer architectures leverage years of research to prevent the introduction of accidental bugs and optimise runtime performance. Java does not allow direct manipulation of memory and uses a bytecode verifier to prevent harmful operations. Rust uses ownership types to efficiently free memory without the need for a garbage collector. Pony uses reference capabilities to prevent data races in concurrent programs.

With Flint, we followed this trend by identifying the challenges specific to developing programs for Ethereum, and designed a programming language tailored for writing smart contracts. We focused on providing a safe development experience while maintaining ease of use. In addition to providing intuitive semantics, such as the prevention of implicit integer overflows, we have designed novel security features. To protect attackers from performing unauthorised operations on smart contracts, we use Flint's *caller capabilities* to systematically require the programmer to specify which parties can call each of the contract's functions. We verify the validity of internal function calls at compile-time, allowing us to omit expensive checks at runtime. To help developers safely transfer currency in smart contracts, we implement Flint's *Asset types*. These prevent the accidental creation and destruction of currency, and allow the use of safe atomic transfer operations. This results in smart contracts maintaining a consistent state at all times. To prevent spurious mutations of the state, we make the distinction between *mutating* and *nonmutating* functions, and require the programmer to specifically indicate mutability in function signatures. This also helps readers to identify them quickly.

In our evaluation, we have shown that Flint programs are significantly safer than programs written in Solidity, Ethereum's most popular language, while maintaining similar or better performance. Flint has also received very positive feedback from the programming languages and Ethereum communities. We hope to see more work on Flint and its toolchain, in particular to support formal verification and easy deployment of Flint programs.

Bibliography

[1] V. Buterin, "Ethereum: A next-generation smart contract and decentralised application platform." https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[2] G. Wood, "Ethereum: a Secure Decentralised General Transaction Ledger." http://gavwood.com/paper.pdf, 2014.

[3] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London mathematical society*, vol. 2, no. 1, pp. 230–265, 1937.

[4] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[5] Ethereum, "Solidity Documentation." http://solidity.readthedocs.io/en/latest/, 2014.

[6] OpenZeppelin, "StandardToken." https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/StandardToken.sol, 2018.

[7] CCN, "Ethereum's Solidity Flaw Exploited in DAO Attack Says Cornell Researcher." https://www.ccn.com/ethereum-solidity-flaw-dao/.

[8] Parity, "The Multi-sig Hack: A Postmortem." http://paritytech.io/the-multi-sig-hack-a-postmortem/.

[9] Parity, "A Postmortem on the Parity Multi-Sig Library Self-Destruct." http://paritytech.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/.

[10] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts (SoK)," in *International Conference on Principles of Security and Trust*, pp. 164–186, 2017.

[11] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 254–269, 2016.

[12] ConsenSys, "Mythril — Security analysis tool for Ethereum smart contracts." https://github.com/ConsenSys/mythril/.

[13] Oracle, "The Java Language Specification." https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf, 2018.

[14] S. P. Jones, *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[15] Apple, "The Swift programming language." https://swift.org.

[16] Rust, "Rust Documentation." https://doc.rust-lang.org, 2018.

[17] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil, "Deny capabilities for safe, fast actors," in *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pp. 1–12, ACM, 2015.

[18] B. Edgington, "LLL Programming Language." `http://lll-docs.readthedocs.io/en/latest/`, 2017.

[19] J. Wilcke, "Mutan Programming Language." `https://github.com/obscuren/mutan`, 2017.

[20] Ethereum, "Serpent." `https://github.com/ethereum/serpent`, 2017.

[21] Ethereum, "The Vyper programming language." `https://github.com/ethereum/vyper`.

[22] D. Flanagan, *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2006.

[23] Python, "Python." `https://www.python.org`, 2018.

[24] F. Schrans, "The Flint Programming Language GitHub Repository." `https://github.com/franklinsch/flint`.

[25] F. Schrans, S. Eisenbach, and S. Drossopoulou, "Writing Safe Smart Contracts in Flint," *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming>)*, 2018.

[26] F. Schrans, "Flint: A New Language for Safe Smart Contracts on Ethereum." `https://medium.com/@fschrans/flint-a-new-language-for-safe-smart-contracts-on-ethereum-a5672137a5c7`.

[27] N. Doiron, "Modeling a mudaraba smart contract." `https://medium.com/@mapmeld/modeling-a-mudaraba-smart-contract-eaa040c5dac6`.

[28] K. Team, "The Keccak-256 algorithm." `https://keccak.team`.

[29] M. J. Dworkin, "SHA-3 standard: Permutation-based hash and extendable-output functions," tech. rep., 2015.

[30] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ECDSA)," *International journal of information security*, vol. 1, no. 1, pp. 36–63, 2001.

[31] Ethereum, "Patricia Tree." `https://github.com/ethereum/wiki/wiki/Patricia-Tree`, 2018.

[32] Ethereum, "Recursive Length Prefix." `https://github.com/ethereum/wiki/wiki/RLP`.

[33] Web3, "web3.js." `https://github.com/ethereum/web3.js/`.

[34] "Solidity Application Binary Interface." `https://solidity.readthedocs.io/en/develop/abi-spec.html`.

[35] Wired, "A $50 million hack just showed that the dao was all too human." `https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/`.

[36] P. Daian, "Chasing the DAO Attacker's Wake." `https://pdaian.com/blog/chasing-the-dao-attackers-wake/`.

[37] GoodAudience, "How $800k Evaporated from the PoWH Coin Ponzi Scheme Overnight." `https://blog.goodaudience.com/how-800k-evaporated-from-the-powh-coin-ponzi-scheme-overnight-1b025c33b530`.

[38] Ethereum, "Remix IDE for Solidity." `https://remix.ethereum.org`, 2017.

[39] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Beguelin, "Formal verification of smart contracts," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security-PLAS16*, pp. 91–96, 2016.

[40] I. Sergey, A. Kumar, and A. Hobor, "Scilla: a Smart Contract Intermediate-Level LAnguage," *arXiv preprint arXiv:1801.00687*, 2018.

[41] Zeppelin, "Serpent Compiler Audit." `https://blog.zeppelin.solutions/serpent-compiler-audit-3095d1257929`, 2017.

[42] Y. Hirai, "The Bamboo programming language." `https://github.com/pirapira/bamboo`.

[43] Ethereum, "Joyfully Universal Language for (Inline) Assembly." `http://solidity.readthedocs.io/en/develop/julia.html`.

[44] Inria, "The Coq Proof Assistant." `https://coq.inria.fr`, 2018.

[45] M. S. Miller, K.-P. Yee, J. Shapiro, *et al.*, "Capability myths demolished," tech. rep., Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www.erights.org/elib/capability/duals, 2003.

[46] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[47] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[48] D. Walker, "Substructural type systems," *Advanced Topics in Types and Programming Languages*, pp. 3–44, 2005.

[49] J. Palsberg and C. B. Jay, "The essence of the visitor pattern," in *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International*, pp. 9–15, IEEE, 1998.

[50] Ethereum, "Truffle." `https://github.com/trufflesuite/truffle`.

[51] T. CI, "Travis CI." `https://travis-ci.org`, 2018.

[52] H. Haskins, "Lite." `https://github.com/llvm-swift/Lite`, 2018.

[53] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, p. 75, IEEE Computer Society, 2004.

[54] Robert Widmann, "FileCheck." `https://github.com/llvm-swift/FileCheck`, 2018.

[55] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, no. 1, pp. 157–171, 1986.

[56] D. Weekly, "DApps Weekly, about Flint." `http://dappsweekly.com/issues/15`.

[57] Marcin Krzyzanowski, "CryptoSwift." `https://github.com/krzyzanowskim/CryptoSwift`, 2018.

[58] Kyle Fuller, "Commander." `https://github.com/kylef/Commander`, 2018.

[59] F. Schrans, "The Flint Language Guide." `https://docs.flintlang.org`.

Third-Party Libraries

We used the following third-party libraries when using the Flint compiler.

CryptoSwift [57], by Marcin Krzyzanowski, to compute Keccak-256 hashes in Swift.

Commander [58], by Kyle Fuller under the BSD license, to parse command line arguments.

Lite [52], by Harlan Haskins under the MIT License, as a test runner tool.

FileCheck [54], by Robert Widmann under the MIT License, as a Swift version of LLVM's [53] FileCheck tool.

Appendix A

The Flint Grammar

The grammar is specified in Backus-Naur form. Elements in square brackets are tokens, and elements in parentheses are optional.

```
// Top-level declarations

top-level-declarations -> top-level-declaration (top-level-declarations)
top-level-declaration -> contract-declaration | contract-behavior-declaration | struct-
    declaration

// Contract declaration

contract-declaration -> [contract] identifier [{] (variable-declarations) [}]

// Variable declarations

variable-declarations -> variable-declaration (variable-declarations)
variable-declaration -> [var] identifier type-annotation ([=] expression)

// Type annotations

type-annotation -> [:] type

// Types

type -> identifier (generic-argument-clause) | [[] type []] | type [[] numeric-literal []]
    | [[] type [:] type []]

generic-argument-clause -> [<] generic-argument-list [>]
generic-argument-list -> type | type [,] generic-argument-list

// Struct declaration

struct-declaration -> [struct] identifier [{] (struct-members) [}]
struct-members -> struct-member (struct-members)
struct-member -> variable-declaration | function-declaration | initializer-declaration

// Contract behavior declaration

contract-behavior-declaration -> identifier [::] (caller-capability-binding) caller-
    capability-group (contract-behavior-members)
contract-behavior-members -> contract-behavior-member (contract-behavior-members)
```

```
contract-behavior-member -> function-declaration | initializer-declaration

// Caller capability group

caller-capability-group -> [(] caller-capability-list [)]
caller-capability-list -> caller-capability-identifier | caller-capability-identifier [,]
    caller-capability-list
caller-capability-identifier -> identifier
caller-capability-binding -> identifier [<-]

// Identifier

identifier -> [a-zA-Z] . [a-zA-Z0-9$]*

// Function and initializer declarations

initializer-declaration -> initializer-head parameter-clause code-block
function-declaration -> function-head identifier parameter-clause (function-result) code-
    block

initializer-head -> (declaration-attributes) (declaration-modifiers) [init]
function-head -> (declaration-attributes) (declaration-modifiers) [func]

declaration-modifier -> [public] | [mutating]
declaration-modifiers -> declaration-modifier (declaration-modifiers)

function-result -> [->] type

parameter-clause -> [(] [)] | [(] parameter-list [)]
parameter-list -> parameter | parameter [,] parameter-list
parameter -> identifier type-annotation

declaration-attribute -> [@] . [a-zA-Z]*
declaration-attributes -> declaration-attribute (declaration-attributes)

// Code block

code-block -> [{] statements [}]

// Statements

statements -> statement (statements)
statement -> expression
statement -> [return] (expression)
statement -> if-statement

// Expression

expression -> identifier | in-out-expression | binary-expression | function-call | literal
    | bracketed-expression | subscript-expression
in-out-expression -> [&] expression
binary-expression -> expression binary-operator expression
bracketed-expression -> [(] expression [)]
subscript-expression -> identifier [[] expression []]

// Function Call
```

```
function-call -> identifier function-call-argument-clause
function-call-argument-clause -> [() ()] | [() function-call-argument-list ()]
function-call-argument-list -> expression | expression [,] function-call-argument-list

// Binary Operators

binary-operator -> [+] | [-] | [=] | [.]

// Branching

if-statement -> [if] expression code-block (else-clause)
else-clause -> [else] code-block

// Literal

literal -> numeric-literal | string-literal | boolean-literal

numeric-literal -> decimal-literal
decimal-literal -> [0-9]+ | [0-9]+ [.] [0-9]+

array-literal -> [[] array-literal-elements []]
array-literal-elements -> expression ([,] array-literal-elements)
dictionary-literal -> [[] dictionary-literal-elements []]
dictionary-literal-elements -> expression [:] expression ([,] dictionary-literal-elements)

string-literal -> ["] [a-zA-Z0-9]* ["]
boolean-literal -> [true] | [false]
```

Listing A.1: The Flint Grammar

We thank GitHub user `vietlq` for finding typos in the grammar.

Appendix B

Installing the Flint Compiler and Running Flint Smart Contracts

B.1 Installing Flint

This installation guide is available on `https://docs.flintlang.org/installation`.

B.1.1 Docker

The Flint compiler and its dependencies can be installed using Docker:

```
docker pull franklinsch/flint
docker run -i -t franklinsch/flint
```

Example smart contracts are available in `/flint/examples/valid/`.

B.1.2 Binary Packages and Building from Source

Dependencies

Swift

The Flint compiler is written in Swift, and requires the Swift compiler to be installed, either by:

- Mac only: Installing Xcode (recommended)
- Mac/Linux: Using `swiftenv`
    1. Install swiftenv: `brew install kylef/formulae/swiftenv`
    2. Run `swiftenv install 4.1`

solc

Flint also requires the Solidity compiler to be installed:

Mac

```
brew update
brew upgrade
brew tap ethereum/ethereum
brew install solidity
```

Linux

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install solc
```

Binary Packages

Flint is compatible on macOS and Linux platforms, and can be installed by downloading a built binary directly. The latest releases are available at `https://github.com/franklinsch/flint/releases`.

Building From Source

The best way to start contributing to the Flint compiler, `flintc`, is to clone the GitHub repository and build the project from source. Once you have the swift command line tool installed, you can build `flintc`.

```
git clone https://github.com/franklinsch/flint.git
cd flint
make
```

The built binary is available at `.build/debug/flintc`. Add `flintc` to your `PATH` using:

```
export PATH=$PATH:.build/debug/flintc
```

Using Xcode

If you have Xcode on your Mac, you can use Xcode to contribute to the compiler. You can generate an Xcode project using:

```
swift package generate-xcodeproj
open flintc.xcodeproj
```

B.1.3 Vim Syntax Highlighting

Syntax highlighting for Flint source files can be obtained by running:

```
ditto utils/vim /.vim
```

B.2 Compiling and Running Flint Smart Contracts

Flint compiles to EVM bytecode, which can be deployed to the Ethereum blockchain using a standard client, or Truffle. For testing purposes, the recommended way of running a contract is by using the Remix IDE.

B.2.1 Using Remix

Remix is an online IDE for testing Solidity smart contracts. Flint contracts can also be tested in Remix, by compiling Flint to Solidity.

In this example, we are going to compile and run the Counter contract, available to download on `https://github.com/franklinsch/flint/blob/master/examples/valid/counter.flint`.

Compiling

A Flint source file named `counter.flint` containing a contract `Counter` can be compiled to a Solidity file using:

```
flintc main.flint --emit-ir
```

You can view the generate code, embedded as a Solidity program:

```
cat bin/main/Counter.sol
```

Example smart contracts are available in the repository, under `examples/valid`.

B.2.2 Interacting with the Contract in Remix

To run the generated Solidity file on Remix:

1. Copy the contents of `bin/main/Counter.sol` and paste the code in Remix.
2. Press the red Create button under the Run tab in the right sidebar.
3. You should now see your deployed contract below. Click on the copy button on the right of `Counter` to copy the contract's address.
4. Select from the dropdown right above the Create button and select `_InterfaceMyContract`.
5. Paste in the contract's address in the "Load contract from Address" field, and press the At Address button.
6. You should now see the public functions declared by your contract (`getValue`, `set`, and `increment`). Red buttons indicate the functions are mutating, whereas blue indicated non-mutating.
7. You should now be able to call the contract's functions.

Appendix C

Flint GitHub Repository and Flint Language Guide

C.1 GitHub

Flint's GitHub repository is available on `https://github.com/franklinsch/flint`. The project is released under the MIT license, and contains the source code of the compiler, automated tests, and documentation.

With over 560 commits, the development of the compiler started in late December 2017. A contribution graph is presented in Figure C.1.

C.1.1 Questions and Feature Suggestions

We use GitHub's Issues system to track work on the Flint compiler. Any user can ask questions and suggest features. We have closed about 70 issues, and about 40 are still open and track future work. The addition of new features is performed through Pull Requests, and have to be approved by the owner of the GitHub repository (me). We have closed 114 since the inception of the project.

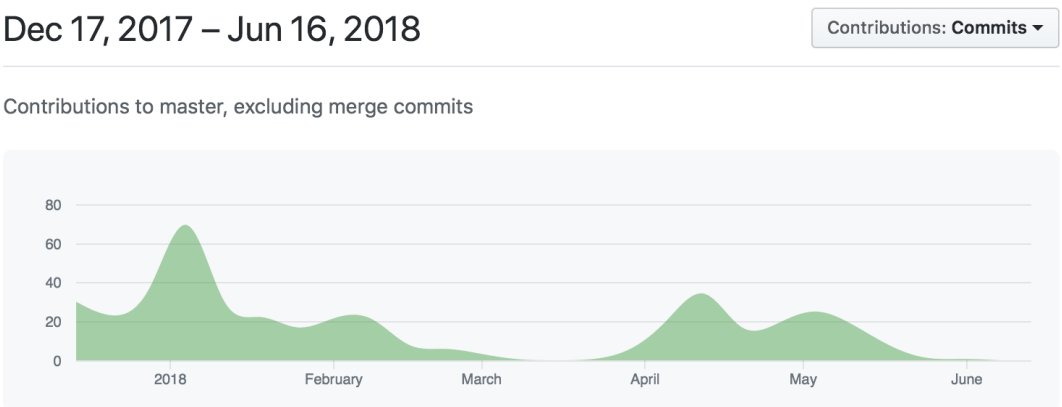


Figure C.1: Flint contributions, since December 2018

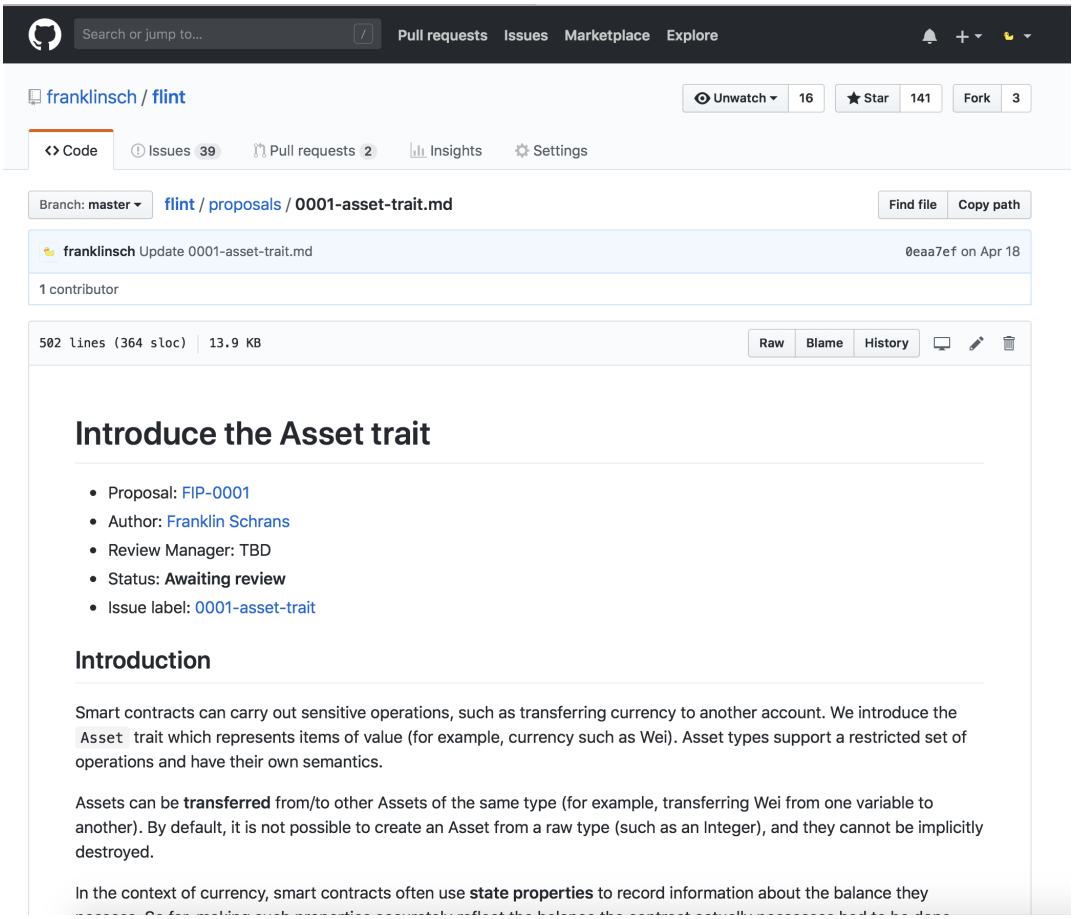


Figure C.2: Original Flint Asset proposal

C.1.2 Proposals

Flint Improvement Proposals (FIPs) track the design and implementation of larger new features for Flint or the Flint compiler. We follow the same model as the Swift Evolution process[1]. We show the original proposal for the Asset type in Figure C.2.

C.2 Flint Language Guide

The Flint Language Guide [59] provides a instructions to get started with Flint programming, and an overview of its features, as shown in Figure C.3.

[1] `https://github.com/apple/swift-evolution`

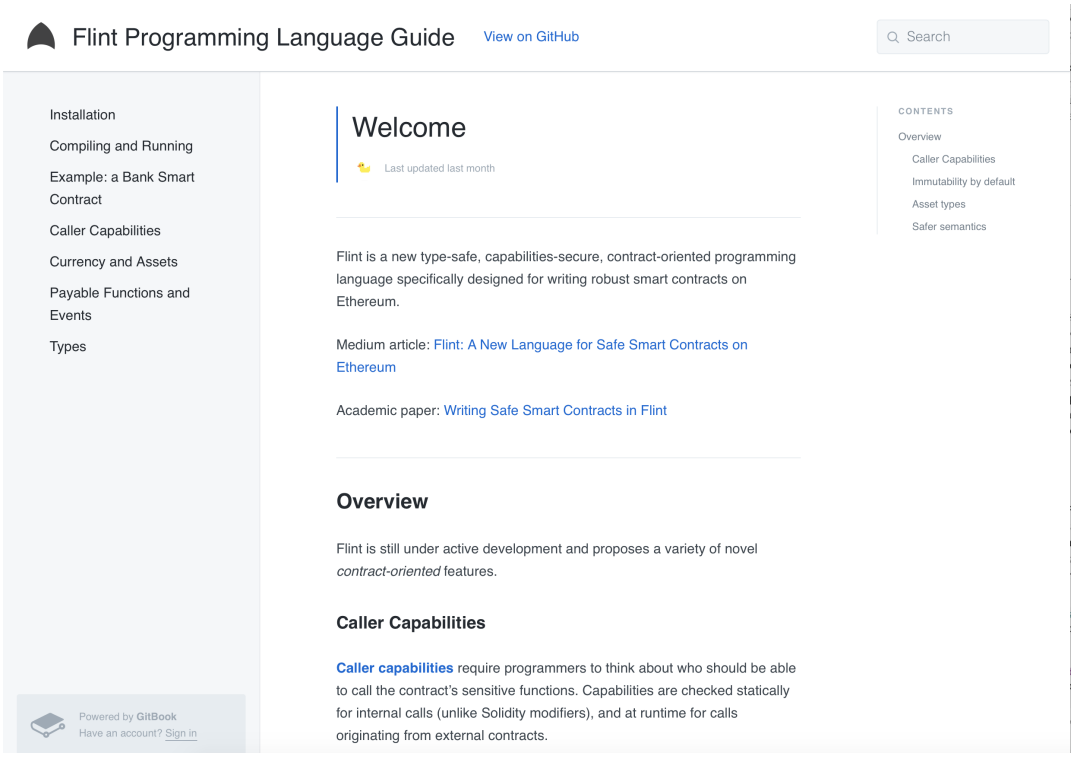


Figure C.3: Original Flint Asset proposal

Appendix D

Flint Package Manager Smart Contract

```
// The Flint Package Manager allows developers to share and use Flint contracts.
// This smart contract records package information, including hashes of the source
// code and interface, and security warnings produced by the Flint compiler.
contract PackageManager {
  var admin: Address
  var packages: [String: Package]
}

// Public API
PackageManager :: caller <- (any) {
  public mutating func createPackage(name: String, version: Int, hash: String) {
    assert(!packageExists(name))
    packages[name] = Package(name, owner, version, hash)
  }

  public mutating func updatePackage(packageName: String, newVersion: Int, newHash: String)
      {
    assert(!packageExists(packageName))
    assert(packages[packageName].isOwnedBy(caller))
    assert(newVersion > packages[packageName].version)

    packages[packageName].version = newVersion
    packages[packageName].hash = newHash
  }

  public mutating func addSecurityWarning(packageName: String, warning: String) {
    packages[packageName].addSecurityWarning(warning)
  }

  public func getPackageHash(name: String) -> String {
    return packages[name].hash
  }

  public func getPackageVersion(name: String) -> Int {
    return packages[name].version
  }

  public func getPackageNumSecurityWarnings(name: String) -> Int {
```

```
    return packages[name].securityInformation.numWarnings
  }

  public func getPackageSecurityWarnings(name: String, index: Int) -> String {
    return packages[name].getSecurityWarning(index)
  }

  func packageExists(name: String) -> Bool {
    return packages[name].name != ""
  }
}

// API for the PackageManager's administrator.
PackageManager :: (admin) {
  public mutating func deletePackage(name: String) {
    // Reset the fields
    packages.removeEntry(name)
  }
}

// Represents a Flint package.
struct Package {
  // Name of the package.
  var name: String

  // Creator and owner of the package.
  var owner: Address

  // Latest version of the package.
  var version: Int

  // Hash of the package's source code.
  var hash: String

  // Hash of the package's interface.
  var interfaceHash: String

  // Security information relating to this package.
  var securityInformation: SecurityInformation

  // == Functions ==

  func isOwnedBy(address: Address) -> Bool {
    return owner == address
  }

  mutating func addSecurityWarning(warning: String) {
    securityInformation.addWarning(warning)
  }

  func getSecurityWarning(index: Int) -> String {
    return securityInformation.getWarning(index)
  }
}

// Encapsulates security information about a package.
struct SecurityInformation {
```

```
95    var warnings: [String]
96
97    mutating func addWarning(warning: String) {
98      warnings.append(warning)
99    }
100
101   func getWarning(index: Int) -> String {
102     return warnings[index]
103   }
104 }
```

Listing D.1: Package Manager Prototype

Appendix E

Proposal to Rename Caller Capabilities

Rename the term "caller capabilities"

- Proposal: FIP-0002
- Author: Franklin Schrans
- Review Manager: TBD
- Status: **Awaiting review**
- Issue label: not yet

Introduction

The term "caller capabilities" has been used to refer to the mechanism which ensures functions of a Flint contract can only be called by a specific set of users. In particular, we say the caller of a function must have the correct "caller capability" in order to be able to call a function. This term might be however confusing, as the term "capability" is used differently in other languages which feature capabilities. Capabilities are usually *transferable*, and Flint caller capabilities are not. We propose renaming caller capabilities to **caller identities**.

Motivation

Programming languages such as Pony use the term "reference capabilities" to express access rights on *objects*. In Flint, caller capabilities express access rights to *functions*. However, the term "capability" usually refers to *transferable* access rights. This means that if an entity is allowed to access a resource, it should be able to transfer that right to another entity. Mark Miller et al. describe four security models which make the distinction between Access Control Lists (ACLs) and different types of capabilities. Flint caller capability would actually fit more under "Model 1. ACLs as columns". Some definitions regard a capability as an *unforgeable token*, i.e., a bit string which when possessed by a user, allows access to a resource.

Flint caller capabilities in fact implement something more similar to Role-Based Access Control (RBAC). RBAC-based systems restrict certain operations to sets of users, through *roles*: if a user has the appropriate role, it is allowed to perform the operation. In Flint, functions can only be called if the user has the appropriate role.

In the following example, `clear(address:)` can only be called by the user which has the Ethereum address stored in the `manager` state property.

```
contract Bank {
  var manager: Address
}

Bank :: (manager) {
  public func clear(address: Address) {
    // ...
  }
}
```

The manager's right to call `clear(address:)` is non-transferable, i.e., it cannot be delegated to another Ethereum user, which might be the expectation when thinking about a capability.

Proposed solution

We suggest the term **caller identity**. It clearly portrays that the determination of whether a caller is allowed to call a function is based on an *identity* check. Naturally, identities cannot be transferred, and this term better describes Flint's mechanism.

We'll say a caller is allowed to call a function if it has the appropriate **caller identity**, or simply the appropriate *identity*, rather than *capability*.

The error message related to invalid function calls due to incompatible caller identities would be updated:

```
Bank :: (any) {
  func foo() {
    // Error: Function bar cannot be called by any user
    bar()
  }
}

Bank :: (manager) {
  func bar() {
  }
}

Bank :: (manager, admin) {
  func baz() {
    // Error: Function bar cannot be called by all users in (manager, admin)
    bar()
  }
}

Bank :: (admin) {
  func qux() {
    // Error: Function bar cannot be called by admin
    bar()
  }
}
```

Alternatives considered

The term *role* was also considered instead of *identity*. I personally prefer "caller identity" than "caller role", and "requiring to have a specific identity to call a function" than "requiring to have a specific role". I am open to discussion for using "role" instead.

Thank you

Thank you Mark Miller for bringing up the incorrect use of the term, and for suggesting Flint's mechanism is closer to RBAC-based systems.

Appendix F

Example Intermediate Representation File

```
pragma solidity ^0.4.21;

contract Counter {

  constructor() public {
    assembly {
      // Memory address 0x40 holds the next available memory location.
      mstore(0x40, 0x60)

      init()
      function init() {

        sstore(add(0, 0), 0)
      }
      function Flint$Global_send_Address_$inoutWei(_address, _value, _value$isMem)  {
        let _w := flint$allocateMemory(32)
        Wei_init_$inoutWei(_w, 1, _value, _value$isMem)
        flint$send(Wei_getRawValue(_w, 1), _address)
      }

      function Flint$Global_fatalError()  {
        flint$fatalError()
      }

      function Flint$Global_assert_Bool(_condition)  {
        switch eq(_condition, 0)
        case 1 {
          Flint$Global_fatalError()
        }

      }

      function Wei_init_Int(_flintSelf, _flintSelf$isMem, _unsafeRawValue)  {
        flint$store(flint$computeOffset(_flintSelf, 0, _flintSelf$isMem), _unsafeRawValue,
            _flintSelf$isMem)
      }

      function Wei_init_$inoutWei_Int(_flintSelf, _flintSelf$isMem, _source, _source$isMem,
```

```
          _amount)  {
38        switch lt(Wei_getRawValue(_source, _source$isMem), _amount)
39        case 1 {
40          Flint$Global_fatalError()
41        }
42
43        flint$store(flint$computeOffset(_source, 0, _source$isMem), flint$sub(flint$load(
              flint$computeOffset(_source, 0, _source$isMem), _source$isMem), _amount),
              _source$isMem)
44        flint$store(flint$computeOffset(_flintSelf, 0, _flintSelf$isMem), _amount,
              _flintSelf$isMem)
45      }
46
47      function Wei_init_$inoutWei(_flintSelf, _flintSelf$isMem, _source, _source$isMem)  {
48        let _value := Wei_getRawValue(_source, _source$isMem)
49        flint$store(flint$computeOffset(_source, 0, _source$isMem), flint$sub(flint$load(
              flint$computeOffset(_source, 0, _source$isMem), _source$isMem), _value),
              _source$isMem)
50        flint$store(flint$computeOffset(_flintSelf, 0, _flintSelf$isMem), _value,
              _flintSelf$isMem)
51      }
52
53      function Wei_transfer_$inoutWei_Int(_flintSelf, _flintSelf$isMem, _source,
            _source$isMem, _amount)  {
54        switch lt(Wei_getRawValue(_source, _source$isMem), _amount)
55        case 1 {
56          Flint$Global_fatalError()
57        }
58
59        flint$store(flint$computeOffset(_source, 0, _source$isMem), flint$sub(flint$load(
              flint$computeOffset(_source, 0, _source$isMem), _source$isMem), _amount),
              _source$isMem)
60        flint$store(flint$computeOffset(_flintSelf, 0, _flintSelf$isMem), flint$add(
              flint$load(flint$computeOffset(_flintSelf, 0, _flintSelf$isMem),
              _flintSelf$isMem), _amount), _flintSelf$isMem)
61      }
62
63      function Wei_transfer_$inoutWei(_flintSelf, _flintSelf$isMem, _source, _source$isMem)
            {
64        Wei_transfer_$inoutWei_Int(_flintSelf, _flintSelf$isMem, _source, _source$isMem,
              Wei_getRawValue(_source, _source$isMem))
65      }
66
67      function Wei_getRawValue(_flintSelf, _flintSelf$isMem) -> ret {
68        ret := flint$load(flint$computeOffset(_flintSelf, 0, _flintSelf$isMem),
              _flintSelf$isMem)
69      }
70      function flint$selector() -> ret {
71        ret := div(calldataload(0), 0
              x100000000000000000000000000000000000000000000000000000000)
72      }
73
74      function flint$decodeAsAddress(offset) -> ret {
75        ret := flint$decodeAsUInt(offset)
76      }
77
78      function flint$decodeAsUInt(offset) -> ret {
```

```
        ret := calldataload(add(4, mul(offset, 0x20)))
      }

      function flint$store(ptr, val, mem) {
        switch iszero(mem)
        case 0 {
          mstore(ptr, val)
        }
        default {
          sstore(ptr, val)
        }
      }

      function flint$load(ptr, mem) -> ret {
        switch iszero(mem)
        case 0 {
          ret := mload(ptr)
        }
        default {
          ret := sload(ptr)
        }
      }

      function flint$computeOffset(base, offset, mem) -> ret {
        switch iszero(mem)
        case 0 {
          ret := add(base, mul(offset, 32))
        }
        default {
          ret := add(base, offset)
        }
      }

      function flint$allocateMemory(size) -> ret {
        ret := mload(0x40)
        mstore(0x40, add(ret, size))
      }

      function flint$isValidCallerCapability(_address) -> ret {
        ret := eq(_address, caller())
      }

      function flint$isCallerCapabilityInArray(arrayOffset) -> ret {
        let size := sload(arrayOffset)
        let found := 0
        let _caller := caller()
        let arrayStart := flint$add(arrayOffset, 1)
        for { let i := 0 } and(lt(i, size), iszero(found)) { i := add(i, 1) } {
          if eq(sload(flint$storageArrayOffset(arrayOffset, i)), _caller) {
            found := 1
          }
        }
        ret := found
      }

      function flint$return32Bytes(v) {
        mstore(0, v)
```

```
        return(0, 0x20)
      }

      function flint$isInvalidSubscriptExpression(index, arraySize) -> ret {
        ret := or(iszero(arraySize), or(lt(index, 0), gt(index, flint$sub(arraySize, 1))))
      }

      function flint$storageArrayOffset(arrayOffset, index) -> ret {
        let arraySize := sload(arrayOffset)

        switch eq(arraySize, index)
        case 0 {
          if flint$isInvalidSubscriptExpression(index, arraySize) { revert(0, 0) }
        }
        default {
          sstore(arrayOffset, flint$add(arraySize, 1))
        }

        ret := flint$storageDictionaryOffsetForKey(arrayOffset, index)
      }

      function flint$storageFixedSizeArrayOffset(arrayOffset, index, arraySize) -> ret {
        if flint$isInvalidSubscriptExpression(index, arraySize) { revert(0, 0) }
        ret := flint$add(arrayOffset, index)
      }

      function flint$storageDictionaryOffsetForKey(dictionaryOffset, key) -> ret {
        mstore(0, key)
        mstore(32, dictionaryOffset)
        ret := sha3(0, 64)
      }

      function flint$send(_value, _address) {
        let ret := call(gas(), _address, _value, 0, 0, 0, 0)

        if iszero(ret) {
          revert(0, 0)
        }
      }

      function flint$fatalError() {
        revert(0, 0)
      }

      function flint$add(a, b) -> ret {
        let c := add(a, b)

        if lt(c, a) { revert(0, 0) }
        ret := c
      }

      function flint$sub(a, b) -> ret {
        if gt(b, a) { revert(0, 0) }

        ret := sub(a, b)
      }

```

```
193       function flint$mul(a, b) -> ret {
194         switch iszero(a)
195         case 1 {
196           ret := 0
197         }
198         default {
199           let c := mul(a, b)
200           if iszero(eq(div(c, a), b)) { revert(0, 0) }
201           ret := c
202         }
203       }
204
205       function flint$div(a, b) -> ret {
206         if eq(b, 0) { revert(0, 0) }
207         ret := div(a, b)
208       }
209     }
210   }
211
212   function () public payable {
213     assembly {
214       // Memory address 0x40 holds the next available memory location.
215       mstore(0x40, 0x60)
216
217       switch flint$selector()
218
219       case 0x20965255 /* getValue() */ {
220
221         flint$return32Bytes(getValue())
222       }
223
224       case 0xd09de08a /* increment() */ {
225
226         increment()
227       }
228
229       case 0x60fe47b1 /* set(uint256) */ {
230
231         set(flint$decodeAsUInt(0))
232       }
233
234       default {
235         revert(0, 0)
236       }
237
238       // User-defined functions
239
240       function getValue() -> ret {
241         ret := sload(add(0, 0))
242       }
243
244       function increment()  {
245         sstore(add(0, 0), flint$add(sload(add(0, 0)), 1))
246       }
247
248       function set(_value)  {
249         sstore(add(0, 0), _value)
```

```
250       }
251
252       // Struct functions
253
254       function Flint$Global_send_Address_$inoutWei(_address, _value, _value$isMem)  {
255         let _w := flint$allocateMemory(32)
256         Wei_init_$inoutWei(_w, 1, _value, _value$isMem)
257         flint$send(Wei_getRawValue(_w, 1), _address)
258       }
259
260       function Flint$Global_fatalError()  {
261         flint$fatalError()
262       }
263
264       function Flint$Global_assert_Bool(_condition)  {
265         switch eq(_condition, 0)
266         case 1 {
267           Flint$Global_fatalError()
268         }
269
270       }
271
272       function Wei_init_Int(_flintSelf, _flintSelf$isMem, _unsafeRawValue)  {
273         flint$store(flint$computeOffset(_flintSelf, 0, _flintSelf$isMem), _unsafeRawValue,
              _flintSelf$isMem)
274       }
275
276       function Wei_init_$inoutWei_Int(_flintSelf, _flintSelf$isMem, _source, _source$isMem,
          _amount)  {
277         switch lt(Wei_getRawValue(_source, _source$isMem), _amount)
278         case 1 {
279           Flint$Global_fatalError()
280         }
281
282         flint$store(flint$computeOffset(_source, 0, _source$isMem), flint$sub(flint$load(
              flint$computeOffset(_source, 0, _source$isMem), _source$isMem), _amount),
              _source$isMem)
283         flint$store(flint$computeOffset(_flintSelf, 0, _flintSelf$isMem), _amount,
              _flintSelf$isMem)
284       }
285
286       function Wei_init_$inoutWei(_flintSelf, _flintSelf$isMem, _source, _source$isMem)  {
287         let _value := Wei_getRawValue(_source, _source$isMem)
288         flint$store(flint$computeOffset(_source, 0, _source$isMem), flint$sub(flint$load(
              flint$computeOffset(_source, 0, _source$isMem), _source$isMem), _value),
              _source$isMem)
289         flint$store(flint$computeOffset(_flintSelf, 0, _flintSelf$isMem), _value,
              _flintSelf$isMem)
290       }
291
292       function Wei_transfer_$inoutWei_Int(_flintSelf, _flintSelf$isMem, _source,
          _source$isMem, _amount)  {
293         switch lt(Wei_getRawValue(_source, _source$isMem), _amount)
294         case 1 {
295           Flint$Global_fatalError()
296         }
297
```

```
298         flint$store(flint$computeOffset(_source, 0, _source$isMem), flint$sub(flint$load(
                flint$computeOffset(_source, 0, _source$isMem), _source$isMem), _amount),
                _source$isMem)
299         flint$store(flint$computeOffset(_flintSelf, 0, _flintSelf$isMem), flint$add(
                flint$load(flint$computeOffset(_flintSelf, 0, _flintSelf$isMem),
                _flintSelf$isMem), _amount), _flintSelf$isMem)
300       }
301
302       function Wei_transfer_$inoutWei(_flintSelf, _flintSelf$isMem, _source, _source$isMem)
              {
303         Wei_transfer_$inoutWei_Int(_flintSelf, _flintSelf$isMem, _source, _source$isMem,
                Wei_getRawValue(_source, _source$isMem))
304       }
305
306       function Wei_getRawValue(_flintSelf, _flintSelf$isMem) -> ret {
307         ret := flint$load(flint$computeOffset(_flintSelf, 0, _flintSelf$isMem),
                _flintSelf$isMem)
308       }
309
310       // Flint runtime
311
312       function flint$selector() -> ret {
313         ret := div(calldataload(0), 0
                x100000000000000000000000000000000000000000000000000000000)
314       }
315
316       function flint$decodeAsAddress(offset) -> ret {
317         ret := flint$decodeAsUInt(offset)
318       }
319
320       function flint$decodeAsUInt(offset) -> ret {
321         ret := calldataload(add(4, mul(offset, 0x20)))
322       }
323
324       function flint$store(ptr, val, mem) {
325         switch iszero(mem)
326         case 0 {
327           mstore(ptr, val)
328         }
329         default {
330           sstore(ptr, val)
331         }
332       }
333
334       function flint$load(ptr, mem) -> ret {
335         switch iszero(mem)
336         case 0 {
337           ret := mload(ptr)
338         }
339         default {
340           ret := sload(ptr)
341         }
342       }
343
344       function flint$computeOffset(base, offset, mem) -> ret {
345         switch iszero(mem)
346         case 0 {
```

```
          ret := add(base, mul(offset, 32))
        }
        default {
          ret := add(base, offset)
        }
      }

      function flint$allocateMemory(size) -> ret {
        ret := mload(0x40)
        mstore(0x40, add(ret, size))
      }

      function flint$isValidCallerCapability(_address) -> ret {
        ret := eq(_address, caller())
      }

      function flint$isCallerCapabilityInArray(arrayOffset) -> ret {
        let size := sload(arrayOffset)
        let found := 0
        let _caller := caller()
        let arrayStart := flint$add(arrayOffset, 1)
        for { let i := 0 } and(lt(i, size), iszero(found)) { i := add(i, 1) } {
          if eq(sload(flint$storageArrayOffset(arrayOffset, i)), _caller) {
            found := 1
          }
        }
        ret := found
      }

      function flint$return32Bytes(v) {
        mstore(0, v)
        return(0, 0x20)
      }

      function flint$isInvalidSubscriptExpression(index, arraySize) -> ret {
        ret := or(iszero(arraySize), or(lt(index, 0), gt(index, flint$sub(arraySize, 1))))
      }

      function flint$storageArrayOffset(arrayOffset, index) -> ret {
        let arraySize := sload(arrayOffset)

        switch eq(arraySize, index)
        case 0 {
          if flint$isInvalidSubscriptExpression(index, arraySize) { revert(0, 0) }
        }
        default {
          sstore(arrayOffset, flint$add(arraySize, 1))
        }

        ret := flint$storageDictionaryOffsetForKey(arrayOffset, index)
      }

      function flint$storageFixedSizeArrayOffset(arrayOffset, index, arraySize) -> ret {
        if flint$isInvalidSubscriptExpression(index, arraySize) { revert(0, 0) }
        ret := flint$add(arrayOffset, index)
      }

```

```
404       function flint$storageDictionaryOffsetForKey(dictionaryOffset, key) -> ret {
405         mstore(0, key)
406         mstore(32, dictionaryOffset)
407         ret := sha3(0, 64)
408       }
409
410       function flint$send(_value, _address) {
411         let ret := call(gas(), _address, _value, 0, 0, 0, 0)
412
413         if iszero(ret) {
414           revert(0, 0)
415         }
416       }
417
418       function flint$fatalError() {
419         revert(0, 0)
420       }
421
422       function flint$add(a, b) -> ret {
423         let c := add(a, b)
424
425         if lt(c, a) { revert(0, 0) }
426         ret := c
427       }
428
429       function flint$sub(a, b) -> ret {
430         if gt(b, a) { revert(0, 0) }
431
432         ret := sub(a, b)
433       }
434
435       function flint$mul(a, b) -> ret {
436         switch iszero(a)
437         case 1 {
438           ret := 0
439         }
440         default {
441           let c := mul(a, b)
442           if iszero(eq(div(c, a), b)) { revert(0, 0) }
443           ret := c
444         }
445       }
446
447       function flint$div(a, b) -> ret {
448         if eq(b, 0) { revert(0, 0) }
449         ret := div(a, b)
450       }
451     }
452   }
453 }
454 interface _InterfaceCounter {
455   function getValue() view external returns (uint256 ret);
456   function increment() external;
457   function set(uint256 _value) external;
458 }
```

Appendix G

Full Contracts from Evaluation

G.1 Caller Capabilities

Solidity

```
contract CapsSolidity {
  address owner;
  address[] customers;
  uint256 counter;

  // Modifiers.

  modifier onlyOwner {
    require(msg.sender == owner);
    _;
  }

  modifier anyCustomer {
    uint numCustomers = customers.length;
    bool found = false;

    for (uint i = 0; i < numCustomers; i++) {
      if (customers[i] == msg.sender) { found = true; }
    }

    require(found);
    _;
  }

  function CapsSolidity() { owner = msg.sender; }
  function addCustomer(address customer) { customers.push(customer); }

  function anyUser() public constant returns(uint256) { return 1; }

  function ownerCall(uint counter) public constant onlyOwner returns(uint256) {
    return counter + 1;
  }

  function customerCall(uint counter) public constant anyCustomer returns(uint256) {
    return counter + 1;
```

```
  }

  function ownerInternalCalls() public onlyOwner {
    uint256 x = 0;
    x += ownerCall(x)
    // 34 calls hidden: x += customerCall(x)
    x += ownerCall(x)
    counter = x;
  }

  function customerInternalCalls() public anyCustomer {
    uint256 x = 0;
    x += ownerCall(x)
    // 34 calls hidden: x += customerCall(x)
    x += ownerCall(x)
    counter = x;
  }
}
```

Vyper

```
owner: public(address)
customers: public(address[50])
counter: public(uint256)

@public
@constant
def anyUser() -> uint256:
  return 1

@private
@constant
def isCustomer(a: address) -> bool:
  found: bool = False
  for c in range(50):
    if self.customers[c] == a:
      found = True

  return found

@public
@constant
def customerCall(_counter: uint256) -> uint256:
  assert self.isCustomer(msg.sender)
  return _counter + 1

@public
@constant
def ownerCall(_counter: uint256) -> uint256:
  assert msg.sender == self.owner
  return _counter + 1

@public
def ownerInternalCalls():
  assert msg.sender == self.owner
  x: uint256 = 0
  x += self.ownerCall(x)
```

```
  // 34 calls hidden: x += self.ownerCall(x)
  x += self.ownerCall(x)
  self.counter = x

@public
def customerInternalCalls():
  assert self.isCustomer(msg.sender)
  x: uint256 = 0
  x += self.customerCall(x)
  // 34 calls hidden: x += customerCall(x)
  x += self.customerCall(x)
  self.counter = x
```

Flint

```
contract Caps {
  var owner: Address
  var customers: [Address] = []
  var numCustomers: Int = 0

  var counter: Int = 0
}

Caps :: owner <- (any) {
  public init() {
    self.owner = owner
  }

  mutating public func addCustomer(customer: Address) {
    customers[numCustomers] = customer
    numCustomers += 1
  }

  public func anyUser() -> Int {
    return 1
  }
}

Caps :: (owner) {
  public func ownerCall(counter: Int) -> Int {
    return counter + 1
  }

  public mutating func ownerInternalCalls() {
    var x: Int = 0
    x += ownerCall(x)
    // 34 calls hidden: x += customerCall(x)
    x += ownerCall(x)
    self.counter = x
  }
}

Caps :: (customers) {
  public func customerCall(counter: Int) -> Int {
    return counter + 1
  }

```

```
  public mutating func customerInternalCalls() {
    var x: Int = 0
    x += customerCall(x)
    // 34 calls hidden: x += customerCall(x)
    x += customerCall(x)
    self.counter = x
  }
}
```

G.2 Asset Types and Safe Arithmetic Operation

Solidity

```
contract Bank {
  address manager;
  mapping (address => uint) balances;
  address[] accounts;

  modifier onlyManager {
    require(msg.sender == manager);
    _;
  }

  modifier anyCustomer {
    uint numCustomers = accounts.length;
    bool found = false;

    for (uint i = 0; i < numCustomers; i++) {
      if (accounts[i] == msg.sender) {
        found = true;
      }
    }

    require(found);
    _;
  }

  function Bank(address _manager) public {
    manager = _manager;
  }

  function register() public {
    accounts.push(msg.sender);
  }

  function mint(address account, uint amount) onlyManager public payable {
    balances[account] += amount;
  }

  function getBalance() anyCustomer public view returns(uint) {
    return balances[msg.sender];
  }

  function transfer(uint amount, address destination) anyCustomer public {
    require(balances[msg.sender] >= amount);
    balances[destination] += amount;
```

```
    balances[msg.sender] -= amount;
  }

  function deposit() anyCustomer public payable {
    balances[msg.sender] += msg.value;
  }

  function withdraw(uint amount) anyCustomer public {
    require(balances[msg.sender] >= amount);
    balances[msg.sender] -= amount;
    msg.sender.transfer(amount);
  }
}
```

Flint

```
contract Bank {
  var manager: Address
  var balances: [Address: Wei] = [:]
  var accounts: [Address] = []
  var lastIndex: Int = 0
}

Bank :: account <- (any) {
  public init(manager: Address) {
    self.manager = manager
  }

  public mutating func register() {
    accounts[lastIndex] = account
    lastIndex += 1
  }
}

Bank :: (manager) {
  public mutating func mint(account: Address, amount: Int) {
    var w: Wei = Wei(amount)
    balances[account].transfer(&w)
  }
}

Bank :: account <- (accounts) {
  public func getBalance() -> Int {
    return balances[account].getRawValue()
  }

  public mutating func transfer(amount: Int, destination: Address) {
    balances[destination].transfer(&balances[account], amount)
  }

  @payable
  public mutating func deposit(implicit value: Wei) {
    balances[account].transfer(&value)
  }

  public mutating func withdraw(amount: Int) {
    // Transfer some Wei from balances[account] into a local variable.
```

```
    let w: Wei = Wei(&balances[account], amount)

    // Send the amount back to the Ethereum user.
    send(account, &w)
  }
}
```

G.3 Auction

Solidity

```
contract SimpleAuction {
    // Parameters of the auction. Times are either
    // absolute unix timestamps (seconds since 1970-01-01)
    // or time periods in seconds.
    address public beneficiary;
    uint public auctionEnd;

    // Current state of the auction.
    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    // Set to true at the end, disallows any change
    bool ended;

    // Events that will be fired on changes.
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    constructor(uint _biddingTime, address _beneficiary) public {
        beneficiary = _beneficiary;
        auctionEnd = now + _biddingTime;
    }

    function bid() public payable {
        require(now <= auctionEnd);
        require(msg.value > highestBid);

        if (highestBid != 0) {
            // Sending back the money by simply using
            // highestBidder.send(highestBid) is a security risk
            // because it could execute an untrusted contract.
            // It is always safer to let the recipients
            // withdraw their money themselves.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        emit HighestBidIncreased(msg.sender, msg.value);
    }

    /// Withdraw a bid that was overbid.
    function withdraw() public returns (bool) {
```

```
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the recipient
            // can call this function again as part of the receiving call
            // before `send` returns.
            pendingReturns[msg.sender] = 0;

            if (!msg.sender.send(amount)) {
                // No need to call throw here, just reset the amount owing
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// End the auction and send the highest bid
    /// to the beneficiary.
    function auctionEnd() public {
        // 1. Check conditions
        require(now >= auctionEnd, "Auction not yet ended.");
        require(!ended, "auctionEnd has already been called.");

        // 2. Effects
        ended = true;
        emit AuctionEnded(highestBidder, highestBid);

        // 3. Interaction
        beneficiary.transfer(highestBid);
    }
}
```

Vyper

```
beneficiary: public(address)
auction_end: public(timestamp)
highest_bidder: public(address)
highest_bid: public(wei_value)
pending_returns: uint256[address]
ended: public(bool)

HighestBidIncrease: event({bidder: indexed(address), amount: indexed(uint256)})
AuctionEnded: event({winner: indexed(address), amount: indexed(uint256)})

@public
def __init__(_beneficiary: address, _bidding_time: timedelta):
    self.beneficiary = _beneficiary
    self.auction_end = block.timestamp + _bidding_time

@public
@payable
def bid():
    assert block.timestamp < self.auction_end
    assert msg.value > self.highest_bid
    if not self.highest_bid == 0:
        self.pending_returns[highest_bidder] += highest_bid
    self.highest_bidder = msg.sender
```

```
    self.highest_bid = msg.value
    log.Transfer(HighestBidIncrease(msg.sender, msg.value))

@public
def withdraw():
  uint256 amount = pending_returns[msg.sender]
  if amount > 0:
    pending_returns[msg.sender] = 0
    send(msg.sender, amount)

@public
def end_auction():
    assert block.timestamp >= self.auction_end
    assert not self.ended
    self.ended = True

    log.AuctionEnded(HighestBidIncrease(highestBidder, highestBid))

    send(self.beneficiary, self.highest_bid)
```

Flint

```
contract SimpleAuction {
  let beneficiary: Address

  var highestBidder: Address
  var highestBid: Wei = Wei(0)

  var pendingReturns: [Address: Wei] = [:]

  var hasAuctionEnded: Bool = false

  var highestBidDidIncrease: Event<Address, Int>
  var auctionDidEnd: Event<Address, Int>
}

SimpleAuction :: caller <- (any) {
  public init() {
    beneficiary = caller
    highestBidder = caller
  }

  @payable
  public mutating func bid(implicit value: Wei) {
    // We have not implemented the unary not (!) operator yet.
    assert(hasAuctionEnded == false)
    assert(value.getRawValue() > highestBid.getRawValue())

    if highestBid.getRawValue() != 0 {
      // Record the amount owed to the previous highest bidder.
      pendingReturns[highestBidder].transfer(&highestBid)

      // Alternative: send(highestBidder, &highestBid)
    }

    // Set the new highest bidder.
    highestBidder = caller
```

```
36
37      // Record the new highest bid.
38      highestBid.transfer(&value)
39      highestBidDidIncrease(caller, value.getRawValue())
40    }
41
42    mutating func withdraw() {
43      var amount: Wei = Wei(&pendingReturns[caller])
44      assert(amount.getRawValue() != 0)
45      send(caller, &amount)
46    }
47
48    public func getHighestBid() -> Int {
49      return highestBid.getRawValue()
50    }
51
52    public func getHighestBidder() -> Address {
53      return highestBidder
54    }
55  }
56
57  SimpleAuction :: (beneficiary) {
58    public mutating func endAuction() {
59      assert(hasAuctionEnded == false)
60
61      hasAuctionEnded = true
62      auctionDidEnd(highestBidder, highestBid.getRawValue())
63
64      send(beneficiary, &highestBid)
65    }
66  }
```