Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

A Smart Contract Debugger for Flint

Author:
Ka Yau (Noel) Lee

Supervisor:
Prof. Susan Eisenbach

Second Marker:
Mr. Alex Carver

June 15, 2020

Abstract

Flint is a smart contract programming language designed to be a stricter, safer alternative to existing smart contract languages. Just like any programming language, it is unlikely that Flint would be adopted by the wider developer community if there is not enough tooling support for it.

Currently, there is no debugger for Flint. When programming Flint contracts, the programmer has no visibility of what goes on in the contract, as the code runs within an Ethereum blockchain, a closed environment. Solidity, on the other hand, is supported by multiple debuggers developed by different organizations, some of the most popular ones being Truffle and Remix, which allow programmers to debug Solidity contracts on Ethereum.

We have implemented a debugger for Flint that comes in two flavors: a CLI, and a GUI in the form of a Visual Studio Code extension. We believe this would add to the developer experience provided by Flint, and by proxy, help push Flint into the direction of becoming a language that developers would use in practice.

Acknowledgements

I would like to thank

- Prof. Susan Eisenbach for all the guidance and advice she has given me not only as my project supervisor, but also as my personal tutor throughout the past 4 years;
- Mr. Alex Carver for his advice on better managing and planning my project;
- my friends for bringing joy in times of stress;
- and most importantly, my parents and family, for their neverending support and belief in me.

Contents

Chapter 1

Introduction

Smart contracts are programs that execute on a blockchain. Ethereum is a popular blockchain platform for building decentralized applications using smart contracts. Various programming languages designed to write smart contracts have emerged in recent years. Solidity is one of the first smart contract languages, and is the main language used on Ethereum.

Flint [1] is a smart contract language designed to be a safer alternative of Solidity. Currently, it is still in alpha development and not ready to be used in production code. To prepare Flint for wider adoption in the future, one essential step is to provide a richer tool set for Flint development.

A major deciding factor for developers when choosing programming languages is the tooling support for the language. The range and quality of development tools vastly impacts the developer experience and hence their overall productivity. If the set development tools of a language is lacking in any way, it could greatly discourage a developer from using the language.

At the time of writing, the ecosystem of development tools for Flint is rather limited, compared to popular programming languages. The current ecosystem includes language support in editors, an interactive console, code analysis tools, and more [2]. We believe a debugger is a necessary addition to it, for reasons following.

Smart contracts on Ethereum are notoriously difficult to debug. On most other platforms, when a program throws an error, the system usually tells the programmer which line caused the error and what type of error it was, such as an `OutOfMemoryException` or `DivisionByZeroException`. A stack trace is also usually provided so the programmer can understand the context in which the error has occurred. That is not the case with Ethereum[1]. It does not reveal much information about errors that occur – only a generic error is reported, without any context.

Without a proper debugger, a programmer must rely on alternative debugging techniques. Some debugging strategies when there is no debugger available are:

- **Print debugging.** Also known as trace debugging. A programmer can trace the program execution by sprinkling print statements everywhere. Although often quite effective to a certain degree, there is a lot of manual clean up required. It is not uncommon to accidentally commit diagnostic print statements to version control, or even let one slip through to a production environment. However, when debugging code on the blockchain, this technique cannot be used because there is no printing mechanism on the blockchain.
- **Rubber ducking.** This term was famously coined by *The Pragmatic Programmer: From Journeyman to Master* [3]. By explaining out loud how the code works to a rubber duck

[1] Ethereum by default does not print stack traces, but developers can still achieve that by using a 3rd party library sol-trace (https://sol-trace.com/). However, there are a few drawbacks to using this library – it 1. introduces extra dependencies, 2. can only be used with Ganache, a specific implementation of Ethereum, 3. requires modification to the project source code.

(or perhaps some other object or even another person, when a rubber duck is not available), one might notice things about the program that could have caused the bug. The reasoning behind this technique is rather simple: when explaining code to an external entity, one must view it from the perspective of the listener, rather than their own. The change of perspective allows the programmer to notice details they have either forgotten or taken for granted

- **Binary search.**[2] This method is exceptionally helpful when the programmer does not know the location where an error occurs. It works by the process of elimination. First, comment out the second half of the code and run it again to find out which half causes the error. Repeat the process in the problematic half until the error is found. Depending on the amount of code involved, this method could be extremely tedious.

Although these techniques could be more useful than using a debugger in some cases, it is obvious that using an interactive debugger would make the process more efficient.

1.1 Objectives

The goal of the project is to build a fully featured interactive debugger for Flint. It has to be user-friendly and provides a transparent user experience, so that programmers are able to focus on their debugging task when using the debugger. Also, it should make as few assumptions about the user's development environment as possible. For example, it is desirable to not tie in the debugger with a specific Ethereum implementation. This is to make the debugger available wider audience by not requiring them to modify their existing setup.

1.2 Contributions

Flint debugger CLI (flint-debug). A command line tool that functions as an interactive debugger for Flint contracts. It provides standard features one would expect from a debugger (e.g. source code stepping, breakpoints, etc.), with some additional features we believe would be helpful to a Flint programmer.

Flint debugger extension for VS Code. An alternative user interface of the debugger. This extension integrates Flint debugging capabilities into VS Code, a popular code editor. The extension is published on Visual Studio Marketplace[3].

SwiftDAP. A Swift implementation of the Debug Adapter Protocol (DAP). This is an open source library[4] can be used to build debug adapters and tools conforming to DAP in Swift.

Web3.swift extension. A fork[5] of Boilertalk's Web3.swift library[6], which provides additional support for Ethereum's debug API.

[2] This is similar to the "wolf fence" debugging technique [4].
[3] https://marketplace.visualstudio.com/items?itemName=noellee-doc.flint-debug
[4] https://github.com/noellee/SwiftDAP
[5] https://github.com/noellee/Web3.swift
[6] https://github.com/Boilertalk/Web3.swift

Chapter 2

Background

Debugging refers to the process of removing bugs, i.e. incorrect and/or unexpected behaviors from a computer program. With the help of a debugger, programmers can find the piece(s) of code causing the problem more efficiently.

In the context of programming smart contracts, it is even more important for the code to be as bug-free as possible before being deployed to a production blockchain. In other programming models, it is common to implement some variation of a continuous delivery process, where patches and updates are frequently delivered to the end user. However, deploying a smart contract is a non-reversible action. Once it's deployed, there is no way to update it. This could be especially problematic since money is usually involved when it comes to blockchains.

2.1 Debuggers

Logical errors in code often manifest themselves when the actual state of the program does not match with the state of the program in the programmer's head, e.g. some variable's value is not what they think it is. Debuggers assist a programmer in finding these logical errors, by providing a number of features to allow them to understand and visualize the state of the program, and also to control the execution of the program.

Typically, the program to be debugged by a programmer is run under the control of a debugger. The program being debugged is often referred to as the *debuggee*.

2.1.1 Types of debuggers

In general, the term *debugger* refers to the type of debugger that debugs local programs in real time. This generic type of debugger is usually sufficient for most use cases. However, there are many specialized debuggers to serve specific purposes. In this section, we introduce some notable types of debuggers that are relevant to concepts used in existing smart contract debuggers.

Reverse debuggers. Also known as *time travelling debuggers*, these debuggers support "reverse executing" the debuggee. Fundamentally, programs are not reversible by nature. Reverse debuggers employ a number of different techniques to create the impression of going back in time in the program [5]. The following are some reverse debugging techniques.

- **Trace-based.** All changes to the program state during execution are stored as a log. These reverse debuggers effectively navigates through the history of program state, using this log.

- **Re-execution.** This solution simply re-executes the entire program from the beginning to the target point in time in the program.
- **Record-replay.** This solution is a hybrid of the previous solutions. It selectively checkpoints program states by taking a snapshot of the state, so re-execution can begin from from checkpointed states instead of the start of the program.

Remote debuggers. These debuggers are meant to debug programs that do not run on the same system or platform as them [6]. Typically, remote debuggers communicate with the debuggee's platform using remote procedure calls (RPCs).

Assembly/Bytecode debuggers. Instead of debugging at the source code level, these debuggers are used to debug at the machine code level.

2.1.2 Key features

In this section we discuss main operations provided by debuggers, their semantics and uses. Different debugger implementations may have different names for the same operation, or slightly different semantics for the operations of the same name – here we define the terminology we will use in this report.

2.1.2.1 Breakpoints

Debuggers allow programmers to specify breakpoints. Different types of breakpoints would suspend (break) the program execution under different circumstances. The following types of breakpoints are commonly found in most debuggers.

- **Statement breakpoints.**[1] Breakpoints can be set on specific statements in the source code. When the program reaches any of these lines, its execution pauses until it receives the next debug instruction from the user.
- **Function/Method breakpoints.** Similar to statement breakpoints, this type of breakpoints break the program right before it is about to enter the body of the specified function/method.
- **Variable breakpoints.** This type of breakpoints are set on variables (and fields in the case of object-oriented programming languages). Program execution breaks right before read and/or write accesses to the variable.
- **Exception breakpoints.** Pause program execution when a specific type of exception is thrown.
- **Conditional breakpoints.** In addition, many modern debuggers support specifying extra conditions such that they only break on a breakpoint only if both the breaking condition and the user-specified conditions are satisfied. For example, in Listing 2.1, if we set a line breakpoint at line 2 with an extra condition `i == 5`, the program only breaks on line 2 when `i == 5` evaluates to true, i.e. on the 5th iteration of the loop.

```
1 for (int i = 1; i <= 10; i++) {
2     printf("test");  // set line breakpoint here
3 }
```

Listing 2.1: C example

[1] Statement breakpoints are often referred to as line breakpoints. However, most programming languages allow statements to span multiple lines, and one line may contain multiple statements. Execution of programs are based on statements, not physical lines, so "statement breakpoints" would be a more accurate description of this type of breakpoints.

2.1.2.2 Single-stepping

When a program is paused, the user can control execution in a step-by-step manner.

- **Step over.**[2] Fully execute the current statement, then stop at the next statement.
- **Step into.** Enters the function body of the first function invocation on the current statement.
- **Step out.** Exits the current function body. The current function is run to completion, then the program pauses at the statement after the location of the function invocation.

2.1.2.3 Program state inspection

Variables in the program can be examined for their values when the program is paused. Many debuggers also support evaluating expressions on the spot, based on the current state of the program. Many debuggers also provide the functionality to examine and navigate across the call stack, allowing inspection of relevant variables within different scopes.

2.1.3 Other features

These features are either less common in popular debuggers, or are shortcuts based on the key features above.

Run to cursor Some graphical debuggers provide this feature, which continues execution of the program until it reaches the current location of the cursor in the editor. It essentially sets a temporary line breakpoint at the cursor's location.

Watches The user can save an expression they want to monitor as a "watch". Watches are essentially expressions that will be evaluated every time the debugger pauses the program.

Step back Reverse debuggers provide this operation to step through a program in reverse.

Reverse breakpoints Similar to reverse stepping, some debuggers can "run" the program backwards until a breaking condition is met.

2.1.4 Architecture of a debugger

Generally, debuggers are made up of a backend and a frontend[3]. As illustrated in Figure 2.1, the backend communicates with the execution platform of the debuggee via a debug API provided by the platform, and the frontend provides the user interface [6].

Backend. The debugger backend usually has no direct access to the debuggee, due to obvious security reasons – if malicious programs can arbitrarily control execution and inspect the state of other programs running on the same platform, that could cause catastrophic damages. To provide the control needed to debuggers, most systems provide some kind of debug API to allow processes control other processes *only* when they have the permission to. One example of such debug APIs is `ptrace`, a system call in Unix systems [7].

[2] Sometimes referred to as *step next*.

[3] The backend and frontend are sometimes referred to as the *kernel* and the *user interface (UI)* respectively.

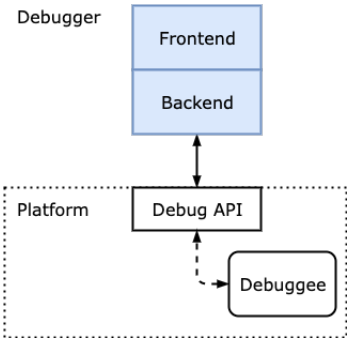
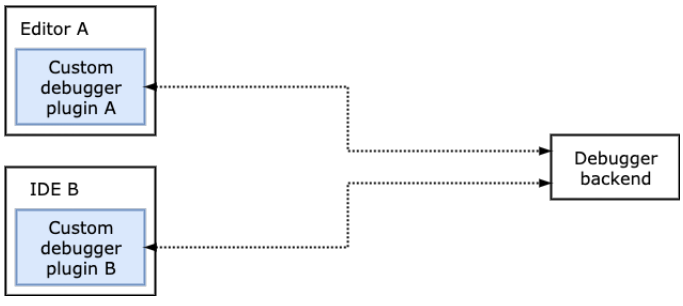


Figure 2.1: Generic debugger architecture

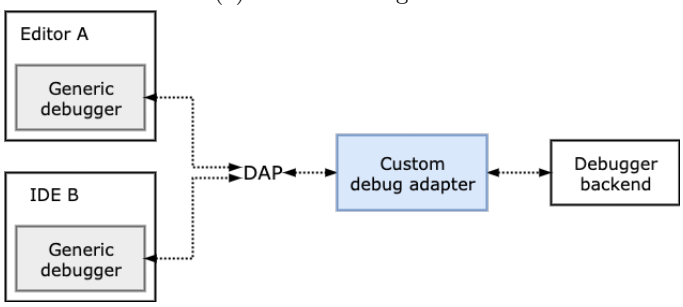
Frontend. In practice, the frontend is usually the built-in debugger of a development tool, i.e. an IDE or a text editor. Debugger developers can leverage the plugin architecture of the target development tool to integrate a debugger backend for a new language to it. Although it is possible to build a standalone frontend specifically for a debugger, it is often unnecessary unless there are specific features that cannot be easily integrated into existing development tools. This is because in most cases, programmers are more inclined to keep using a development tool they are familiar with, rather than installing an extra tool.

2.1.4.1 Debug Adapter Protocol (DAP)

DAP is a protocol developed by Microsoft whose goal is to standardize the communication between debugger frontends and backends [8]. This eliminates the need of building a custom debugger plugin for each development environment. Only a single DAP-compatible debug adapter has to be built in order to integrate a debugger backend into any development environment that supports DAP, as shown in Figure 2.2. Some editors and IDEs that currently support DAP (either natively or through a plugin) include: Visual Studio Code, Eclipse, Vim, Emacs etc [9].



(a) Without using DAP



(b) Using DAP

Figure 2.2: Comparison of development tool debug integration with and without DAP

2.2 Smart contract debuggers

In this section, we will discuss and compare existing debuggers for Solidity, a popular smart contract language targeting Ethereum. We choose to investigate debuggers for an Ethereum smart contract language because Flint also targets Ethereum, thus we can make a more direct comparison between the two. Other Ethereum smart contract languages exist although many are designed for experimental purposes and/or are not being developed any more. Some smart contract languages that are still in development are Vyper, Lira, Huff, etc. However, the programming paradigm and programming style of these languages are not as close to Flint as those of Solidity. Moreover, the size of their user bases cannot compare to that of Solidity, which also explains the lack of tooling support for those languages. Out of the mentioned languages, only Vyper has a debugger, but it has not been updated since May 2019. Because of the above reasons, we will focus on Solidity debuggers only.

2.2.1 Smart contracts on Ethereum

Before going into the discussion of smart contract debuggers, we shall briefly discuss the basics of Ethereum and smart contracts.

Ethereum [10] is a blockchain designed to be a platform for decentralized applications. Ethereum is only a protocol specification – there are many implementations of Ethereum clients. Ethereum clients communicate with each other via the Ethereum protocol, which forms a P2P Ethereum network. An Ethereum network allows for heterogeneity of nodes, as long as they understand the protocol.

There are two types of blockchains: public and private[4]. Anyone is allowed to freely join a public blockchain. On the other hand, to take part in a private blockchain, explicit permission is needed from the owner of the blockchain. Ethereum foundation's official blockchain is referred to as the *mainnet*. Ether on the mainnet is traded for fiat currencies under the symbol *ETH*.

geth[5] is currently the most popular Ethereum client on the mainnet, making up 80% of the mainnet [12].

Ether and gas. Ether (often abbreviated as *eth*) is the currency used on Ethereum. Just like real life currencies, Ether comes in many different denominations: szabo (10^{-6} eth), gwei (10^{-9} eth), wei (10^{-18} eth) etc. Computations on Ethereum incur a fee paid in Ether, depending on the amount of the computation needed. This fee is called *gas*.

Accounts. Accounts are categorized into two types: externally owned accounts (EOA), and contract accounts. The former are owned and controlled by users of Ethereum, and the latter represent deployed smart contracts. Both types of accounts have their own ether balance. Contract accounts also contain contract code. Each account has a unique 20-byte address which is used to identify them.

Transactions. Transaction is comparable to a request message. These following actions all result in a transaction:

- Transferring funds from one EOA to any other account.
- Deploying a smart contract.
- Calling a method on a deployed contract.

[4] They are sometimes referred to as permissionless and permissioned blockchains respectively.
[5] Also known as Go Ethereum, geth is the official Go implementation of Ethereum [11].

All transactions have a `from` field, which contains the address of the EOA that triggered the transaction. Depending on the type of transaction, it could also contain a `to` field that specifies where this transaction is directed to, and/or a `data` field that contains extra data that is sent as a part of the transaction.

Execution model. Ethereum nodes implement an Ethereum virtual machine (EVM), which serves as a platform for code to run on. The EVM supports a set of instructions similar to most standard assembly languages, extended with some blockchain-specific operations. Code written in smart contract languages compile into EVM bytecode, so that it can be executed on an EVM.

Note: It is planned for EVM to be replaced by Ethereum WebAssembly (Ewasm) as the execution engine of the next version of Ethereum [13]. WebAssembly (Wasm) is an assembly language that runs in modern browsers, and designed for performance. Ewasm is a subset of Wasm. Many general purpose programming languages such as C, Rust, Go etc., can compile directly into Wasm. This could greatly expand the smart contract developer community, as there is no need to learn a specialized smart contract language. In addition, the execution speed of Wasm can be taken advantage of.
However, in this project we will only focus on EVM, since:

- adding Ewasm support to Flint is a big migration project on its own because the intermediate representation (IR) code it uses is an older version which cannot be compiled to Ewasm directly.
- the Ewasm toolchain is still in very active development, many features are missing and is generally not stable enough to depend upon at this state. As an example, its testnet has been down since at least Jan 2020 up until now (Jun 2020).

The EVM operates like a standard stack machine. Three types of data storage that are involved in the execution of EVM bytecode.

- **Stack:** The operand stack.
- **Memory:** Local variables and function arguments are stored here.
- **Storage:** A persistent key-value store owned by the contract.

Both the stack and the memory reset after the end of a computation, but data in a contract's storage persists after a computation.

Interacting with smart contracts. Ethereum nodes communicate with other nodes through a set of network protocols. To allow for external applications to also communicate with an Ethereum node, a set of standardized remote procedure call (RPC) APIs can be enabled on a node [14]. These APIs allow external applications to perform actions such as transfering funds, deploying contracts, calling smart contract methods etc. Ethereum's RPC API uses the JSON-RPC protocol, which specifies the encoding of remote procedure calls in JSON. JSON-RPC is transport agnostic – it does not specify any requirements on underlying network layers to facilitate its transportation. Most implementations of Ethereum nodes support some or all of these protocols: WebSocket, HTTP, and IPC.

Listing 2.2 shows an example of calling the RPC API method `eth_getBalance` with parameters `"0xc94770007dda54cF92009BFF0dE90c06F603a09f"` and `"latest"`. The first JSON object is sent from an external application to an Ethereum node, which then responds with the second JSON object with the result of the method call. The fields `jsonrpc`, `id` specifies the JSON-RPC protocol version used, an arbitrary request identifer set by the sender, respectively. `method`, `params`, and `result` specify the name of the remote method, method parameters, and the result from the RPC repectively [15].

```
// Request
{
    "jsonrpc": "2.0",
    "method": "eth_getBalance",
    "params": [
        "0xc94770007dda54cF92009BFF0dE90c06F603a09f",
        "latest"
    ],
    "id": 1
}

// Response
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x0234c8a3397aab58"
}
```

Listing 2.2: JSON-RPC example from the official Ethereum documentation [14].

Development process. Smart contracts cannot be changed once deployed due to the nature of blockchains. This means that when developing smart contracts, it is in the developers' best interest to test them as thoroughly as possible – even more so than most software, which usually can be patched and updated after deployment.

During active development, smart contracts are usually tested on a *local* blockchain running entirely on the developer's machine. This enables more rapid development as there is no external network communication involved and the developer can have full control over the blockchain. Before deploying smart contracts to a production blockchain, developers usually test them on a testnet (e.g. Rinkeby, Ropsten etc.), which acts as a staging environment.

2.2.2 Existing debuggers

2.2.2.1 Remix IDE

Remix [16] is a popular web-based IDE for smart contract development[6]. It provides a graphical user interface for smart contract development.

Using Remix, a smart contract developer can write, compile, and deploy Solidity code to any Ethereum blockchain, including local blockchains. It also supports running an EVM simulator inside the browser to be used as a deploy target.

The Remix debugger does not debug *in real time*. A transaction has to be completed before it can be debugged. The way it works is comparable to a trace-based reverse debugger – it "plays back" the EVM instructions that were involved in a transaction (see subsection 2.1.1). In contrary to generic debuggers, this debugger is attached to a transaction, instead of a process. A transaction can be thought of as a completed program execution.

As shown in Figure 2.3, the panel on the left displays information about the state of the smart contract at the point of execution. This includes the full list of EVM instructions that were run in this transaction, variable values, gas usage, and more. It also provides buttons to control the playback of the execution. The following operations are supported: forward and reverse instruction level stepping, forward and reverse source code level stepping, forward and reverse statement breakpoints, and step out[7].

[6] Remix supports two languages: Solidity and Vyper. Vyper (https://github.com/vyperlang/vyper) is a Python-inspired smart contract language.

[7] *Step into* function calls is equivalent to stepping forward one bytecode instruction.

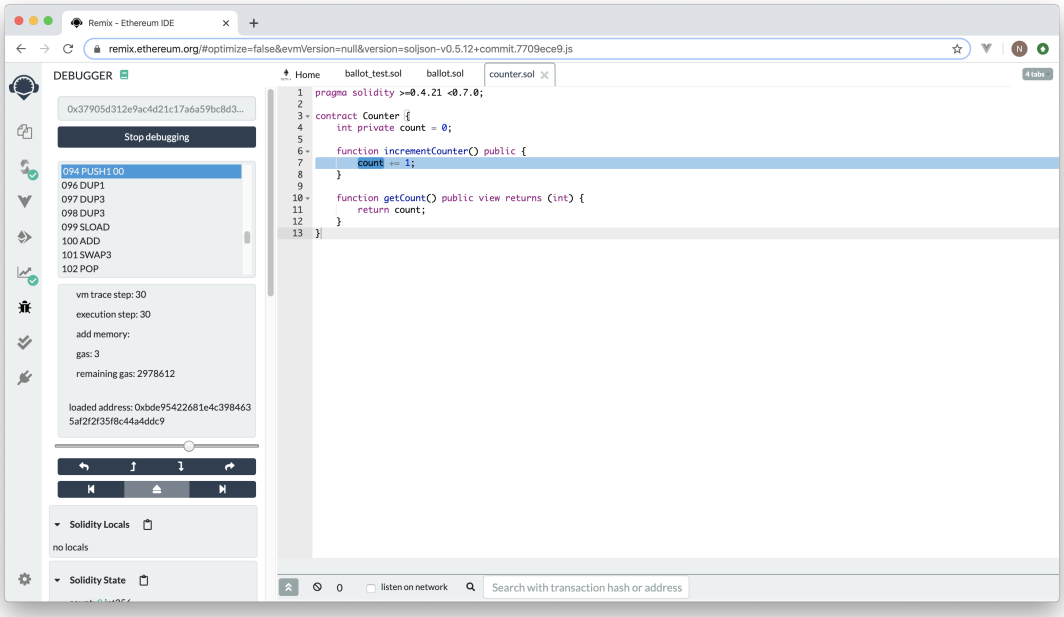


Figure 2.3: Debugging with Remix IDE

remix-debug is the library that powers the Remix IDE debug module. It has been used as the debugger backend in other Solidity debuggers as well. Some examples include:

- **EtherAtom.** A Solidity plugin for Atom[8].
- **Cockpit.** A browser-based IDE that ships with Embark[9], another Solidity development framework.

Implementation. As mentioned in subsection 2.2.1, applications outside of the Ethereum network can communicate and interact with deployed smart contracts via a JSON-RPC API. In addition to the standard set of APIs in the official specification, geth (and some other Ethereum clients) also provides a debug API [17]. remix-debug is agnostic to the implementation of the Ethereum client. Any client that implements the same debug API will be compatible with this debugger. Internally, remix-debug uses web3.js[10], a JavaScript wrapper for the Ethereum RPC API.

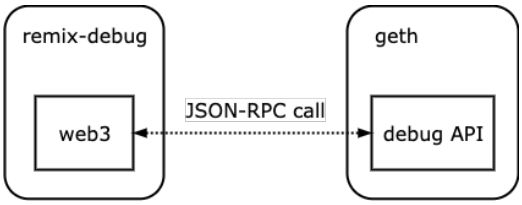


Figure 2.4: How remix-debug would communicate with geth.

The debug API method `debug_traceTransaction` is used to retrieve the full trace of a transaction. The method returns a JSON object that contains a list of log entries. Each of these log entries correspond to an EVM instruction that was run as a part of the transaction. A log entry (see Listing 2.3) contains the state of the EVM when its associated instruction was run.

[8] Atom (https://atom.io/) is a popular text editor for programmers.
[9] https://framework.embarklabs.io/
[10] https://github.com/ethereum/web3.js

```
{
    "depth": <function call depth>,
    "gas": <remaining gas>,
    "gasCost": <gas price of this operation>,
    "memory": [
        <memory item 1>,
        <memory item 2>,
    ],
    "op": <EVM instruction>,
    "pc": <program counter>,
    "stack": [
        <stack item 1>,
        <stack item 2>,
        ...
    ],
    "storage": {
        <storage address 1>: <value at storage address 1>,
        <storage address 2>: <value at storage address 2>,
        ...
    }
}
```

Listing 2.3: Structure of a log entry.

Each Ethereum implementation has their own way of retrieving a the trace of a transaction. As an example, geth reproduces the transaction and captures a full trace of that.

Once the debugger retrieves the bytecode trace, the remaining task would be to match the bytecode to the original source code. A source map can be generated by the Solidity compiler as a compilation artifact when compiling the contract. The source map contains a mapping from the original source code to the generated bytecode. This is used by the debugger to highlight the source code relevant the current debugged instruction.

2.2.2.2 truffle debug

Truffle [18] is one of the most popular Solidity development frameworks, with over 100k downloads per month [19] from npm[11]. It provides a command line interface for developers to compile, test, deploy, interact with, and debug Solidity contracts.

Similar to Remix, the Truffle debugger debugs transactions after they are completed.

To begin a debug session, a user would run `truffle debug <tx_hash>` in the terminal of their choice, where `<tx_hash>` is the transaction hash of the transaction to be debugged. Once the debugger is ready, the user can interact with it by entering commands into the command line interface (CLI) (see Figure 2.5). The debugger supports the following operations: forward stepping, statement breakpoints, and watch expressions. In addition, there are commands to view contract variables and active breakpoints, since these are not always visible in the UI, unlike how they would be in a graphical debugger.

One limitation of this is that it is designed specifically to work on Truffle projects. It cannot be used on generic Solidity projects that do not follow the structure of a Truffle project.

Implementation. The implementation of the Truffle debugger is very similar to remix-debug. It also uses web3.js internally to communicate with an Ethereum client that supports the debug API.

[11] npm (https://www.npmjs.com/) is the official package manager for NodeJS [20]

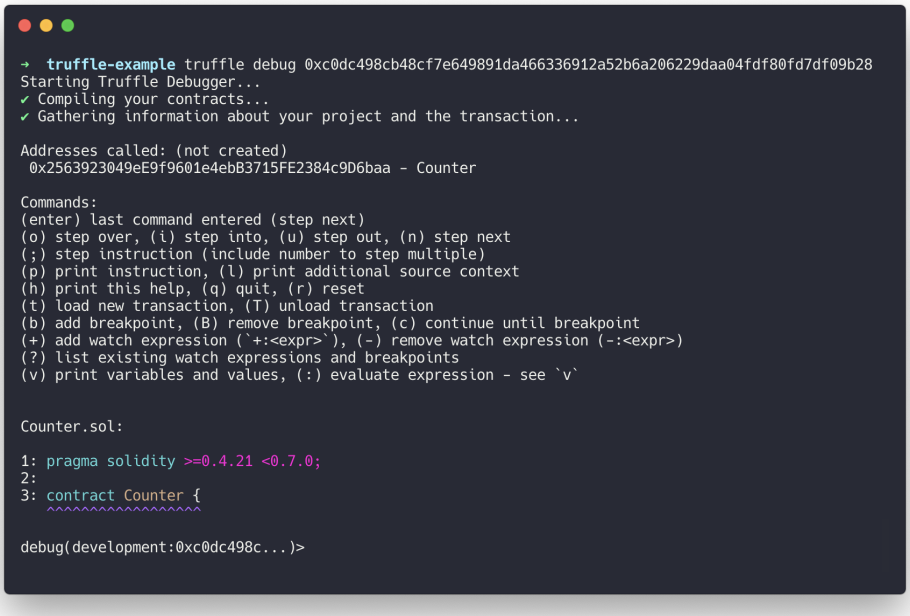


Figure 2.5: Debugging with `truffle debug` in the terminal

2.2.2.3 dapp debug

dapp [21] is a suite of tools for Solidity development. It includes hevm, an EVM implementation in Haskell, designed specifically for unit testing and debugging [22].

This debugger provides a text-based user interface (TUI), which is somewhat of a middle ground between a GUI and a CLI (see Figure 2.6). It has a limited set of graphical features compared to a GUI, but still provides a richer experience than a CLI does.

It supports the following operations: forward and reverse instruction level stepping, and forward and reverse source code level stepping. There is no support for breakpoints. In addition to the bytecode and source code view, it also provides a view of the EVM stack, available gas, and execution trace.

Implementation. This debugger's debugging functionality is completely reliant on the implementation of hevm.

hevm has a special interactive debug mode in which instructions are only executed when it receives input from the user. In a way, there is no distinction between the debugger and the EVM as there is in the previous examples – hevm is both the debugger and the EVM.

Reverse stepping is implemented by re-executing all instructions from the start state until it reaches the instruction right before the current instruction, with some caching optimizations.

2.2.3 Summary

Truffle and Remix's debuggers are remote debuggers that debug from outside of the Ethereum blockchain, while dapp's implementation debugs from within. This gives Truffle and Remix an advantage of being blockchain agnostic, such that the user is not restricted to only one specific Ethereum implementation.

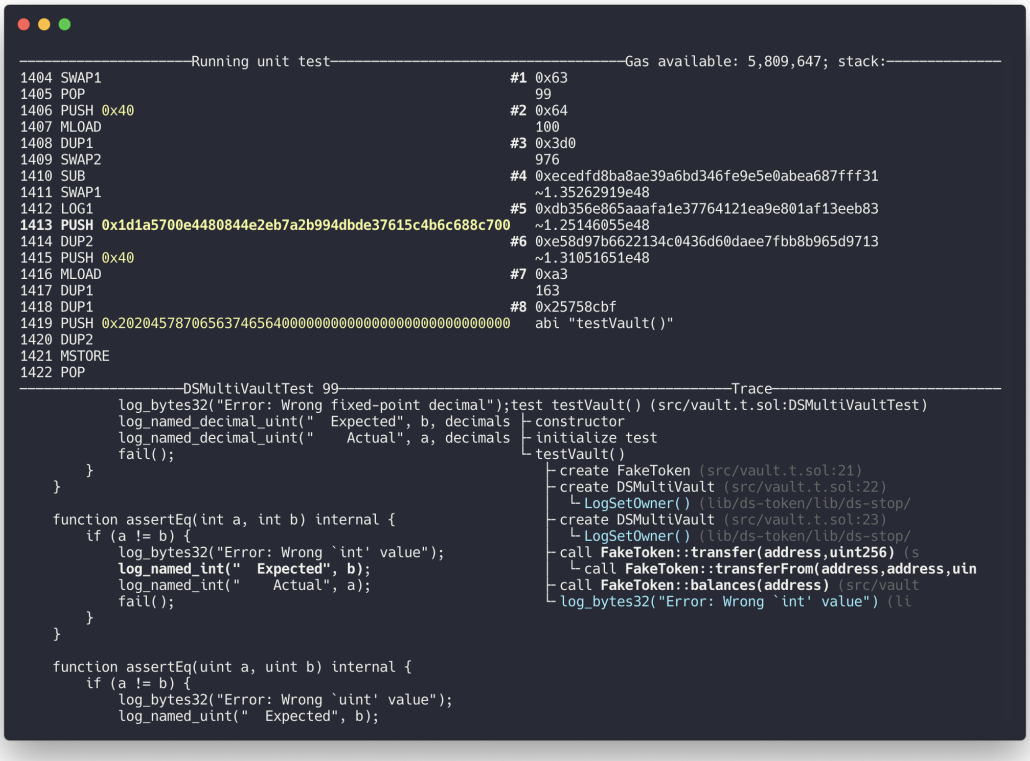


Figure 2.6: Debugging with dapp debug in the terminal

	Blockchain agnostic?	Remote debugging	Reverse debugging	Bytecode debugging	User interface
Truffle	Yes	Yes	No	No	CLI
Remix	Yes	Yes	Yes	Yes	GUI
dapp	No	No	Yes	Yes	TUI

Table 2.1: A summary of Solidity debuggers.

Reverse debugging is relatively easier to implement in smart contract debuggers than in typical debuggers due to the deterministic nature of blockchains as an execution platform and the fact that a blockchain is essentially a historic log of all events that had happened on it. In terms of the implementation, Remix uses a trace-based technique, while dapp uses a re-execution based technique (see subsection 2.1.1 for reverse debugging techniques).

2.3 Flint

Arguably, many high profile attacks targeted towards decentralized applications on Ethereum had been made possible by the design flaws of Solidity. Flint aims to improve upon Solidity, and is designed to make writing unsafe contracts harder [1].

2.3.1 Language features

Flint's syntax is mainly influenced by Swift. Its programming paradigm is rather similar to object-oriented programming (OOP). *Contracts* are analogous to *classes* in OOP, as is *traits* to *interfaces*, *state variables* to *fields*, and *functions* to *methods*. When a contract is deployed to a blockchain,

it is similar to the creation of an object from a class. Deployed smart contracts can call functions of other deployed contracts via *external calls*[12].

The following are two Flint-specific features that helps programmers avoid vulnertabilities caused by unintended behaviors through enforcing static and runtime checks.

Type states. Flint contracts may optionally be modeled as state machines with type states. Static and runtime checks ensure the validity of state transitions within the contract, to avoid vulnerabilities caused by by unexpected state transitions.

For example, in Listing 2.4, the possible states of the contract are defined on line 1, such that `FlintSupermarket` is either `Opened` or `Closed` at any time. The function `open()` lives within a "protection block" declared in line 68-74, such that the function can only be called when the contract is in the state `Closed`. Line 71 transitions the contract's state to `Opened`.

```
1  contract FlintSupermarket (Opened, Closed) {
       ...
68 FlintSupermarket @(Closed) :: (manager) {
69   public func open() {
70     if stock >= 1005011 {
71       become Opened
72     }
73   }
74 }
```

Listing 2.4: Example code snippet from a Flint contract.

Caller protection. Flint requires explicit specification of who, i.e. what Ethereum addresses, are allowed to call each contract function. In the example above (Listing 2.4), only the "manager" can call the contract's `open()` function. Making the programmer specify this explicitly lowers the chance of accidentally allow unauthorized accounts access privileged functions.

2.3.2 Code generation

`flintc` is the command line tool that compiles Flint code to bytecode. It uses Yul[13] as its intermediate language. After generating Yul code from Flint code, the Yul code is embedded into a Solidity file. Then, the Solidity compiler `solc` is used internally to compile the generated Yul code to EVM bytecode.

In addition to EVM bytecode, `flintc` supports compiling to Move IR[14] to target Facebook's Libra blockchain.

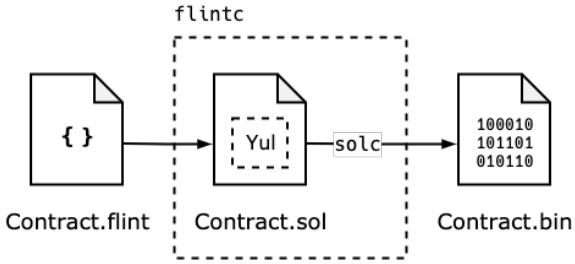


Figure 2.7: Compiling Flint contracts with `flintc`

[12] External calls are currently broken [23].
[13] Previously known as IULIA or JULIA
[14] Move is a programming language designed for Libra. Move IR is the intermediate representation it uses, which compiles directly to Move bytecode [24].

2.3.3 Current tooling support.

Currently, a number of tools are available to make Flint development easier [2].

Syntax highlighting. Available for editors/IDEs that support TmLanguage[15], e.g. VS Code, Atom, etc.

Language server. Integrates with editors to provide live feedback as the programmer codes. Syntax and semantic errors generated by the Flint compiler is reported through the editor.

Contract analysis. Generates reports and visualizations for contracts. It analyzes type state and caller protections of a contract, and can also estimate gas cost for contract functions.

Interactive console (REPL). Enables programmers to interact with contracts through a console.

Testing framework. Enables programmers to write tests for Flint contracts. It also supports generating test coverage reports.

flint-block. A tool that launches a blockchain for testing, using geth[16].

2.4 Concluding remarks

We explored features commonly provided by generic debuggers and how these debuggers are generally implemented. Debuggers usually provide various features such as breakpoints, source code stepping, state inspection etc., to help the programmer understand what is happening in their program. We also looked specifically at existing Solidity debuggers (Truffle, Remix, and dapp), to see in what ways they are similar and different from generic debuggers. dapp's model is quite different from Truffle and Remix's model. dapp debugs in real time on its own implementation of EVM, while Truffle and Remix are trace-based debuggers which integrates with existing Ethereum blockchains. Within the smart contract developer community, Truffle and Remix are more popular, which could be due to the fact that they integrate well with other tools. They can be seen as a standard of what a smart contract developer would expect from a smart contract debugger.

The Flint debugger we implement should be able to perform common debugging commands as mentioned above. It would also be helpful to include features tailored specifically to Flint, based on the special language features it provides, such as type states.

[15] TmLanguage is a programming language grammar specification format.
[16] geth is an Ethereum client implementation, as mentioned in subsection 2.2.1.

Chapter 3

Core Debugger

3.1 Overview

The Flint debugger provides two user interfaces:

- A **command line interface (CLI)**: when used as a standalone program.
- A **graphical interface (GUI)**: when used as a Visual Studio Code (VS Code) extension.

The CLI was created as the first iteration of the project. Once the basic debugging functionalities were completed, the debugger was extended to integrate with VS Code as an extension. Both interfaces use the same underlying core debugger, which is where the debug logic is implemented, therefore they provide the same functionalities.

The debugger is a trace-based debugger – it "plays back" the sequence of EVM instruction executed in a transaction. We also refer to this sequence as the transaction trace.

A typical debugging session using any of the two interfaces would be similar to the following:

1. Compile a Flint contract using the Flint compiler flintc, with the flag `--emit-srcmap`. This generates EVM bytecode for the contract as well as a source map (see chapter 4).
2. Deploy the bytecode to a test Ethereum network.
3. Call a contract function. This results in an Ethereum *transaction*.
4. Start a debug session on the transaction from the previous step.
5. Start debugging, e.g. setting breakpoints, stepping through source code etc.

Currently the debugger supports stepping through source code in a number of ways (step instruction, step in, step next, and step out), setting breakpoints, and inspecting the state of the contract. More about stepping and breakpoints is discussed in section 3.5.

3.2 System Architecture

The overall architecture of the Flint debugger is shown in Figure 3.1.

flint-debug and flint-da are the two external interfaces of the debugger. flint-debug is the debugger's CLI, and flint-da is a debug adapter conforming to the Debug Adapter Protocol (DAP) (see

subsubsection 2.1.4.1) which is used to integrate with VS Code (and possibly other editors in the future, more in chapter 5).

The core debugger depends on 2 other internal components: a Web3 client, and a source code manager. The core debugger uses the Web3 client to communicate with an Ethereum client, so that it can retrieve transaction traces and information about deployed contracts. The source code manager "translates" the raw EVM information obtained from the Ethereum client into the context of Flint source code. The core debugger itself handles the debugging logic, such as managing breakpoints, stepping through the transaction trace etc. This abstracts away the debugging logic from the presentation layer.

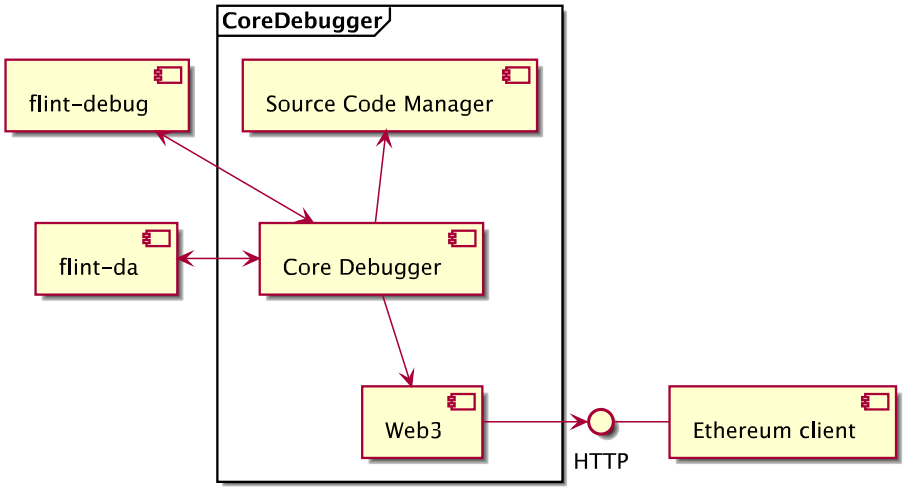


Figure 3.1: Overview of the system architecture.

Language choice. All debugger components shown here are implemented in Swift. In fact, there is only one small part of the project that is not implemented in Swift, which we discuss in chapter 5. It was a deliberate choice to implement as many components of the project in Swift, as the original Flint project is written in Swift. However, a major disadvantage of using Swift is that there is very limited library support for working with Ethereum and the Debug Adapter Protocol (DAP). An alternative would be JavaScript/TypeScript, which has official libraries for both Ethereum and DAP. Nonetheless, we believe the advantages of using Swift outweigh the disadvantages, for the following reasons: (1) Flint's build process is already rather complex, adding another language will only make it even more complex; (2) this allows us to directly share components between the debugger and the Flint compiler.

3.3 Transaction trace

Similar to remix-debug and truffle-debug (subsection 2.2.2), the Flint debugger debugs the trace of an Ethereum transaction. It obtains the trace by communicating with the debug API of an Ethereum client of the user's choice via HTTP, as explained in subsubsection 2.2.2.1. Specifically, the API method in use is `debug_traceTransaction`, which gives us a result similar to what is shown in Listing 3.1 for a transaction.

A transaction trace contains 3 fields: gas, returnValue, and structLogs. Gas and returnValue specify the total amount of gas used in the transaction and the return value of the function executed by the transaction respectively. structLogs contains the most valuable information from the perspective of the debugger. It is an array of log entries which describes the state of the EVM from the beginning of the transaction till the end. Essentially, each log entry is a snapshot of the EVM at each execution of an instruction. Table 3.1 describes the information contained in each log entry.

```
{
   "gas":49074,
   "returnValue":"",
   "structLogs":[
      // ...
      {
         "depth":0,
         "gasCost":12,
         "pc":5,
         "gas":6700613,
         "op":"PUSH1",
         "stack":[
            "0000000000000000000000000000002710",
            "0000000000000000000000000000000112"
         ],
         "memory":[
            "0000000000000000000000000000000000",
            "0000000000000000000000000000000000",
            "0000000000000000000000000000000080"
         ],
         "storage":[
            "00000000":"0000000000000000000000000012345678",
            "00000020":"000000000000000000000000deadbeef"
         ]
      },
      // ...
   ]
}
```

Listing 3.1: Example transaction trace (some leading zeros are omitted for readability).

Field	Description
depth	execution depth
gasCost	cost of each unit of gas
pc	current program counter
gas	amount of remaining gas
op	EVM instruction executed in this step
stack	an array of hex strings describing the contents of the EVM stack
memory	an array of hex strings describing the contents of the contract's memory
storage	a mapping from storage position to the value stored at the position

Table 3.1: Fields in each log entry.

3.3.1 Extending Web3.swift

Libraries that interface with Ethereum clients are referred to as Web3 libraries. Currently, the only official Web3 implementations are in Python and JavaScript – there is no official Web3 implementation for Swift, and none of the existing 3rd party Swift Web3 libraries support the debug API[1]. Therefore, we must select one and extend it to support the `debug_traceTransaction` API method which we need. We chose Web.swift [2] as its codebase is relatively easy to extend, and it is the only library out of all the ones in consideration that supports Swift Package Manager, the dependency manager Flint uses.

[1]The debug API is not a standard Ethereum API, but many Ethereum clients e.g. geth, Ganache, provide the same debug API.

[2]https://github.com/Boilertalk/Web3.swift

3.4 Source Code Manager

From the trace, we know the following:

- The sequence of EVM instructions executed and their corresponding program counter (PC)
- A snapshot of the EVM stack and memory at each instruction execution

As described in section 3.3, this information exist in the context of the EVM. For example, at each execution step, we know what the program counter (PC) is, but we do not know which part of the Flint source code it corresponds to. The source code manager the component responsible for putting the above information into the context of the source code.

This requires a *source map* and some additional metadata generated at compile time. A **compiler artifact** containing this information can optionally be outputted by the compiler as a JSON file, which then can be read by the source code manager. This compiler artifact follows the format shown in Listing 3.2. This format is essentially an augmented version of the Solidity compiler solc's compiler artifact format. This allows us to reuse internal structures when dealing with both compiler's artifacts. The generation of source map and metadata is explained in more detail in chapter 4.

```
{
    <ContractName>: {
        "bin": <contract bytecode>,
        "bin-runtime": <contract runtime bytecode>,
        "srcmap": <contract source map>,
        "srcmap-runtime": <contract runtime source map>,
        "metadata": {
            "storage": [ ... ],
            "typeStates": [ ... ]
        }
    },
    "sourceList": ["/path/to/Contract.flint"],
    "version": <compiler version>
}
```

Listing 3.2: Format of a Flint compiler artifact.

3.4.1 Source map

The source map of a contract tells us which instruction in the compiled EVM bytecode is generated from which part of the source code, i.e. its source location. We define what a source map is as follows.

$$\text{Instructions} : [\text{EVMInstr}]$$
$$\text{InstrIndex} : \{0...\text{length}(\text{Instructions}) - 1\}$$
$$\text{SourceLocation} : (\text{Start}, \text{Length}, \text{File})$$

$$\text{SourceMap} : \text{InstrIndex} \rightarrow \text{SourceLocation}$$

We define *instructions* as the array of EVM instructions generated from a Flint contract. Then, the instruction index is defined as the zero-based index of this array. All instruction indices are mapped to a source location by the source map. Simply put, SourceMap[i] is the i^{th} instruction's corresponding source location. Using the source map, for any instruction, we can lookup which Flint source file it is generated from, and the start and end of the code fragment that generated it.

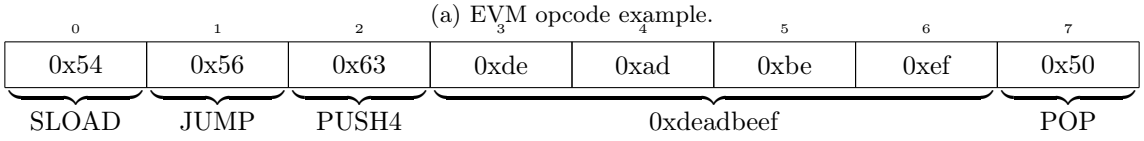
3.4.2 PC to instruction index conversion

From the trace, we know the program counter (PC) of the operation. The PC is the byte offset of the instruction in the bytecode. However, the PC is not equivalent to the instruction index. All instructions *except* `PUSHn` instructions take up 1 byte. A `PUSHn` instruction takes up $1 + n$ bytes: 1 for the opcode itself and n for the value to be pushed.

In the example in Figure 3.2, the instruction `PUSH4` pushes a 4-byte constant `0xdeadbeef` onto the stack. Unlike the other instructions, it takes up 5 bytes in the generated bytecode instead of just 1 byte. The extra 4 bytes are needed to store the constant to be pushed. Therefore, the 7th byte in the bytecode actually corresponds to the 3rd instruction.

```
0 SLOAD
1 JUMP
2 PUSH4 0xdeadbeef  // push the value 0xdeadbeef onto the stack (4 bytes)
3 POP
```

(a) EVM opcode example.



(b) Compiled bytecode.

Figure 3.2: Example of EVM opcodes and equivalent bytecode.

Therefore, when the source code manager is initialized, we first read the contract's bytecode and build a PC-to-instruction index conversion table, so that we can lookup the instruction index of a PC and hence the source location (using the source map).

3.4.3 State variables

The trace provides a snapshot of the contract's storage at each point of the execution. The contract's state variables are stored within the storage in the sequence they are declared [1]. Figure 3.3 shows how the state variables defined in the contract in Listing 3.3 are laid out in memory. `owner` takes up the 0th word, the fixed size array `arr` spans from the 1st word to the 4th word, and so on. The storage scheme of dynamic arrays and dictionaries is slightly less straightforward, which we will discuss below.

```
1 contract Example {
2   var owner: Address
3   var arr: Int[4]  // fixed size array
4   var arr2: [Int]  // dynamic array
5   var myVar: Int
6 }
```

Listing 3.3: Example contract with 4 variables defined.

The storage snapshot in the trace is a mapping from the storage word offset to the value stored at that position. The source code manager resolves the storage word offset to the name of the state variable defined in the contract.

Let's say we assign `arr[2] = 42`. In this trace, we would have a storage entry $(3 \mapsto 42)$, since `arr[2]` is located at the 3rd word in the storage. The source code manager resolves the offset 3 to its corresponding variable name (`arr`) and index (`[2]`). To do that, it needs to look at metadata from the compiler generated by the Flint compiler. As shown in Listing 3.4, the metadata contains the name, type, and size (if it is a fixed array) of each contract state variable, in the sequence they are declared. Then, the source code manager is able to work out which storage offset corresponds to which contract variable.

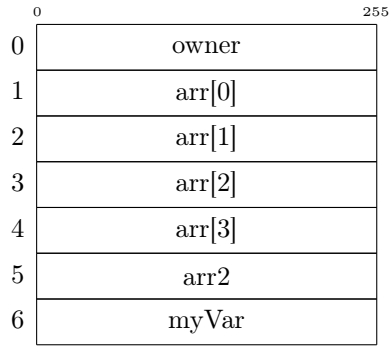


Figure 3.3: Storage layout of the contract in Listing 3.3. Note that the word size of EVM is 8 bytes, or 256 bits.

Knowing the type of the variable from the contract metadata also allows the debugger to format variable values in a user-friendly way, e.g. `Int`s are displayed as a base 10 number instead of a hex string, `Address`es are prefixed with "0x", etc.

```
{
  "contracts": {
    "Example": {
      "metadata": {
        "storage": [
          {
            "name": "owner",
            "type": "Address"
          },
          {
            "name": "arr",
            "type": "Int[4]",
            "size": 4
          },
          {
            "name": "arr2",
            "type": "[Int]"
          },
          {
            "name": "myVar",
            "type": "Int"
          }
        ]
      },
      // ...
    }
  }
}
```

Listing 3.4: Contract artifact generated from Listing 3.3

Limitations. The source code manager, however, does not support dynamic array and dictionary types. This is due to their dynamically sized nature.

Take the dynamically sized integer array `arr2` as an example from Listing 3.3. We can find the storage offset of the k^{th} element of `arr2` as such:

$$\text{Storage offset of } \texttt{arr2[k]} = \text{keccak256}(5 \cdot k),$$

where 5 is the base offset of `arr2`, as shown in Figure 3.3, and keccak256 is the cryptographic hash

function commonly used in Ethereum. Only knowing the storage offset, it is practically impossible to find out that it is hashed from $5 \cdot k$, since keccak256 is an irreversible cryptographic hash function. The same applies to dictionaries.

3.4.3.1 Type state

One unique feature of Flint is *type states* (see section 2.3). Flint uses a special internal variable to store the state of a contract, which is also in the contract's storage area. The source code manager also supports resolving the current state of a Flint contract.

To support this feature, one extra field "typeStates" is added to the contract's metadata, if type states are defined in the contract. Consider the example in Listing 3.6. In the contract's storage, the type state is stored as 0 if it is `Opened`, or 1 if it is `Closed`. This raw value is determined simply by the order in which the states are defined. This metadata allows the source code manager to convert the raw value of the state (0, or 1) to the name of the state (`Opened`, or `Closed`).

```
contract FlintSupermarket (Opened, Closed) {
```

Listing 3.5: Example type state declaration in a Flint contract.

```
{
  "contracts": {
    "FlintSupermarket": {
      "metadata": {
        "storage": [ /* ... */ ],
        "typeStates": ["Opened", "Closed"]
      },
      // ...
    }
  }
}
```

Listing 3.6: Type states in contract metadata.

3.4.4 Contract name detection

The trace itself does not contain information about the contract it calls. We need the contract's name in order to get the correct source map and metadata from the compiler artifact. This is because multiple contracts may be present together in the same compiler artifact, as the Flint compiler supports compiling multiple contracts at once (see example in Listing 3.7). The following steps are taken to resolve the contract name called by the transaction.

1. Use the Web3 method `eth_getTransactionByHash` to retrieve metadata of the transaction, which contains $addr_{to}$, the address the transaction was sent to, i.e. the contract address.
2. Call `eth_getCode` with the parameter $addr_{to}$ to retrieve the contract bytecode.
3. Compare the contract bytecode with the bytecode of each contract in the compiler artifact to find a matching contract.

```
{
    "contracts": {
        "ContractA": {
            "bin": <ContractA bytecode>,
            // ...
        },
        "ContractB": {
            "bin": <ContractB bytecode>,
```

```
            // ...
        },
        // ...
    },
    // ...
}
```

Listing 3.7: Example compiler artifact containing multiple contracts.

In step 3, the contract bytecodes cannot be *directly* checked for equality. The Solidity compiler attaches extra metadata to the end of the compiled contract bytecode [25]. The metadata could be different even if the bytecodes are compiled from the exact same contract. Therefore, the metadata must be first stripped from both bytecodes before they can be checked for equality.

This metadata is encoded as a byte string in the following format[3]:

```
a1 65 62 7a 7a 72 30 58 20 [...32-byte hash...] 00 29
```

Once this byte string is removed, we can directly compare two bytecodes for equality and find the relevant contract in the compiler artifact.

3.5 Stepping and Breakpoints

Although there is a general consensus and expectation of what each "standard" debugger feature does, the nuances still varies from debugger to debugger. In this section we discuss how they are implemented in the Flint debugger.

Step instruction. Step to the next EVM instruction. This is currently unsupported by the debugger GUI, as VS Code does not support this level of stepping granularity[4].

Step in. This is the second finest stepping granularity, after "step instruction". It uses subexpressions as a unit of stepping, i.e. it tries to go as deep as it can in the evaluation of an expression tree. It steps to the next subexpression that is evaluated.

It is possible for one fragment of source code to generate multiple instructions. It is in fact very common to have multiple instructions in a row that point to the same source location. Step in is implemented by stepping through instructions until the source location points to somewhere different from what we started with.

Step next. A step next action behaves identically to a step in action, *except* when the current position is a function call. In this case, it steps *over* the function call, instead of into the callee's function body. This is equivalent to stepping into a function then immediately stepping out.

The source map is used to identify whether or not the debugger is currently at the beginning of a function call. If the current instruction is marked with jump type "into", we know the program is about to jump into a function call. We record the current size of the stack, then "run" the program until the stack grows and shrinks back to the original size. This is how we know the function call has completed and the program has returned to the caller. More about jump types in subsection 4.3.1.

[3] https://solidity.readthedocs.io/en/v0.4.25/metadata.html#encoding-of-the-metadata-hash-in-the-bytecode

[4] The VS Code team is planning to add more levels of stepping granularity to the editor, but it has not been done yet (as of June 2020).

Step out. Run until the end of the function and stop at the caller's location. This is implemented by stopping right after it encounters a jump instruction that is marked with jump type "out", indicating the end of a function. More about jump types in subsection 4.3.1.

Continue. Run until the the end of the the transaction, unless a breakpoint is hit.

Breakpoints. Breakpoints can be set on any line within the source code. The debugger pauses if the debugger encounters a line with a breakpoint set on it.

Reverse debugging. All stepping commands as well as the continue command have a reverse counterpart *except* for step out, as the semantics for stepping out in reverse is unclear. It is done by simply reversing the direction in which the debugger traverses the transaction trace.

3.6 Concluding remarks

The core debugger is the heart of the Flint debugger, where debugging logic is implemented. It makes use of a Web3 client to communicate with an Ethereum client, from which it receives the transaction trace. This transaction trace is translated from the context of EVM to the context of Flint, by using a source map and contract metadata generated by the Flint compiler.

Chapter 4

Source map generation

For each EVM instruction in the compiled bytecode, we need to know which part of the Flint contract it is generated from, so that we can show the Flint programmer where in their code there is a problem. Therefore, during compile time, we need to generate a *source map*, which maps each instruction to a source location of the contract.

As mentioned in section 2.3, the Flint compiler compiles Flint code into YUL code, which is then compiled to EVM bytecode using the Solidity compiler `solc`. `solc` provides an option to emit as a compiler artifact, a source map which maps the EVM bytecode to the corresponding locations of the YUL code. In order to obtain a EVM-to-Flint source map, we generate an intermediate YUL-to-Flint source map during compilation, then merge it with the EVM-to-YUL source map generated by `solc`.

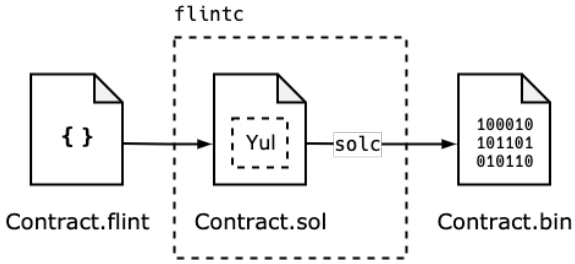


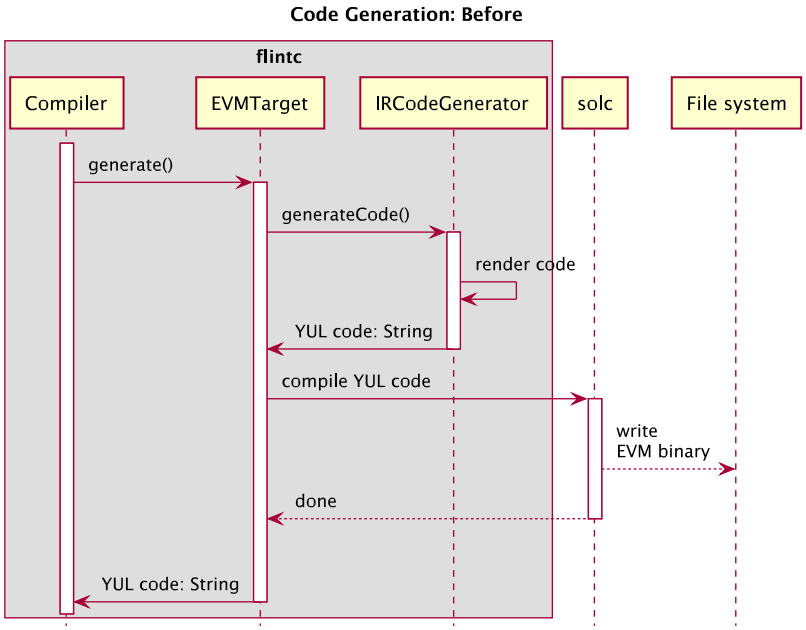
Figure 4.1: Flint code generation mechanism.

4.1 Extending the YUL code generation mechanism

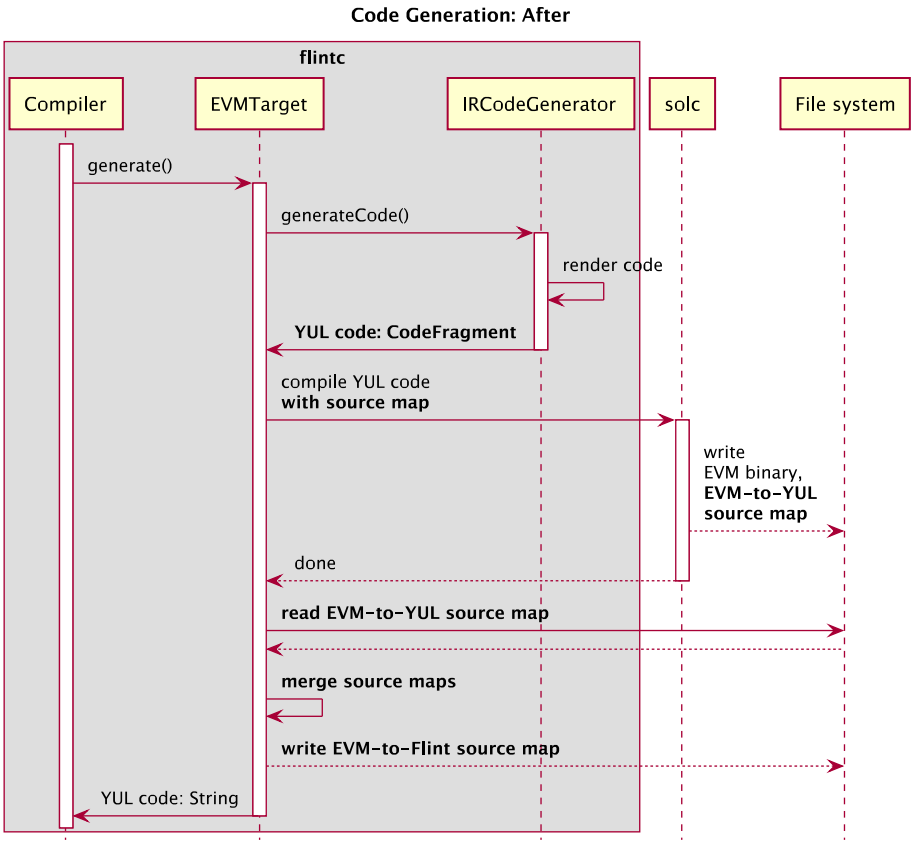
An overview of the existing Flint code generation process is shown in Figure 4.2a.

The `IRCodeGenerator` generates YUL code from the AST and returns it as a string. This YUL code is then compiled into EVM bytecode using `solc`.

When `IRCodeGenerator` renders the code, it internally transforms the AST into different intermediate structures, then converts those structures into the final string. Specifically, AST nodes are converted into `IR`*`Component`*`s` (e.g. `IRContract`, `IRFunction`, `IRStatement` etc.). The `IR`*`Component`*`s` are either (1) rendered directly to a string, or (2) rendered to another internal structure, `YUL.`*`Component`* (e.g. `YUL.Statement`). `YUL.`*`Component`*`s` can also be rendered to a string. All these rendered strings are aggregated recursively to form the final string. If this sounds confusing, it is because it is. This could be attributed to the fact that multiple people have worked on the codebase during disjoint time frames, without having a long term maintainer with a consistent vision for the project.



(a) Before.



(b) After. The bolded text indicates the changes made to the process.

Figure 4.2: Flint's code generation process before and after extended to support source map generation.

4.2 CodeFragment

To obtain the resulting EVM-to-Flint source map we want from the EVM-to-YUL source map generated by the Solidity compiler, we need to generate an intermediate YUL-to-Flint source map. To do so, we need to know which fragment of the resulting code is generated from which AST node, and hence its source location within the Flint file. In the current implementation, this information is lost, since the rendered string does not carry any extra information about where it came from.

We introduce a structure `CodeFragment`, which represents the rendered code as a trie structure. Each node stores an optional source location, signifying that the node and all its children are generated from that location. Each leaf node additionally stores the piece of text it represents.

Doing a depth-first search, we can work out where each fragment lies within the root fragment. Then, we can generate a mapping that maps the range of each fragment to the originating source location (if it exists).

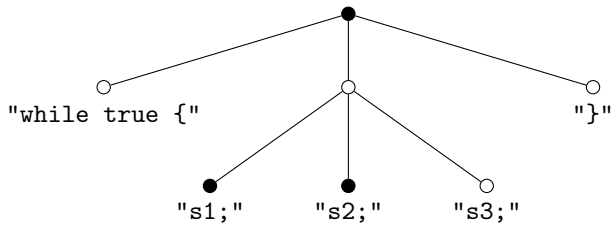


Figure 4.3: Example of a CodeFragment trie. Filled nodes represent code fragments with an attached source location, empty nodes represent those without.

4.2.1 API

Our CodeFragment API is designed to be semantically similar to Swift's built-in `String` class, so to maximize backwards compatibility with the existing code.

String literal A `CodeFragment` can be initialized with a string literal

```
let code: CodeFragment = "generated code"
// or
let code = "generated code" as CodeFragment
```

String interpolation A `CodeFragment` can be initialized via string interpolation. This example creates a `CodeFragment` same as the one in Figure 4.3.

```
// attach source location with .fromSource(...)
let stmt1 = ("s1;" as CodeFragment).fromSource(srcX)
let stmt2 = ("s2;" as CodeFragment).fromSource(srcY)
let stmt3 = "s3;" as CodeFragment

// "s1; s2; s3"
let body: CodeFragment = "\(stmt1) \(stmt2) \(stmt3)"

// "while true { s1; s2; s3 }"
let code: CodeFragment = "while true { \(body) }"

// alternative way of attaching source location
code.sourceLocation = srcZ
```

Operators The operators `+` and `+=` are overloaded to replicate the behavior of strings.

```
// "stmt1;stmt2;stmt3;"
var code: CodeFragment = "stmt1;" + "stmt2;" + "stmt3;"

// "stmt1;stmt2;stmt3;stmt4;"
code += "stmt4;"
```

RenderableToCodeFragment The protocol[1] `RenderableToCodeFragment` mimics the behavior of Swift's `CustomStringConvertible`[2]. Any object that implements `RenderableToCodeFragment` can be used directly within a `CodeFragment` interpolation.

The similarities to the `String` API allow us to preserve the existing structure and logic of the rendering code, while attaching more information to it. An example of refactoring the existing code to use the CodeFragment API is shown in Figure 4.4. The only changes made are:

- **Line 12:** Make `IRFunction` implement `RenderableToCodeFragment`.
- **Line 72:** Change the signature of `rendered()` from returning a `String` to return a `CodeFragment`.
- **Line 80, 84:** Attach the Flint source location to the fragment.

```
11    /// Generates code for a function.
12  - struct IRFunction {
13      static let returnVariableName = "ret"
14
15      var functionDeclaration: FunctionDeclaration
    @@ -69,19 +70,19 @@ struct IRFunction {
69        return functionDeclaration.signature.resultType.flatMap({ CanonicalType(from:
    $0.rawType)! })
70      }
71
72  -   func rendered() -> String {
73        let body = IRFunctionBody(functionDeclaration: functionDeclaration,
74                                  typeIdentifier: typeIdentifier,
75                                  callerBinding: callerBinding,
76                                  callerProtections: callerProtections,
77                                  environment: environment,
78                                  isContractFunction: isContractFunction).rendered()
79
80  -     return """
81        function \(signature()) {
82          \(body.indented(by: 2))
83        }
84  -     """
```

```
12    /// Generates code for a function.
13  + struct IRFunction: RenderableToCodeFragment {
14      static let returnVariableName = "ret"
15
16      var functionDeclaration: FunctionDeclaration
70        return functionDeclaration.signature.resultType.flatMap({ CanonicalType(from:
    $0.rawType)! })
71      }
72
73  +   func rendered() -> CodeFragment {
74        let body = IRFunctionBody(functionDeclaration: functionDeclaration,
75                                  typeIdentifier: typeIdentifier,
76                                  callerBinding: callerBinding,
77                                  callerProtections: callerProtections,
78                                  environment: environment,
79                                  isContractFunction: isContractFunction).rendered()
80
81  +     return ("""
82        function \(signature()) {
83          \(body.indented(by: 2))
84        }
85  +     """ as CodeFragment).fromSource(functionDeclaration.sourceLocation)
```

Figure 4.4: Changes needed to refactor existing code to use the CodeFragment API.

This approach allowed us to only make minimal changes, lowering the risk of breaking its logic.

An alternative approach would be to inject a StringBuilder-like service into each render method, but due to the recursive nature of the existing implementation, we believe the previous approach would be more appropriate.

4.2.2 Intermediate source map generation

A source range can be expressed as a tuple which specifies its starting position and length. For example, a source range (k, n) starts from the k^{th} character in the source file and spans over n characters.

$$\text{SourceRange} : (\text{Start}, \text{Length})$$
$$\text{Intermediate map} : \text{SourceRange} \rightarrow \text{OriginatingSourceLocation}$$

[1] A "protocol" in Swift is equivalent to an "interface" in other OOP languages.

[2] `CustomStringConvertible` requires a `description: String` property to be implemented. Its usage is similar to `toString()` in languages like Java.

We can generate an intermediate source map for a CodeFragment trie using the algorithm in Listing 4.1. The algorithm recursively generates source maps for the trie's sub-tries, merging them together, while keeping track of the current position within the string in order to compute the source range.

```
procedure generate(node, offset):
  srcMap = empty map

  if node is leaf:
    length = node.text.count
  else:
    length = 0
    for child in node.children:
      childSrcMap, childTextLength = generate(child, offset + length)
      srcMap += childSrcMap
      length += childTextLength

  if node has a source location:
    srcMap[(offset, length)] = node.sourceLocation

  return srcMap, length
```

Listing 4.1: Intermediate source map generation algorithm.

4.3 Merging source maps

In this section, we describe how a EVM-to-YUL source map is merged with a YUL-to-Flint intermediate source map to generate a EVM-to-Flint source map.

solc provides an option to output the source map(s) of the compiled contract(s) as a JSON-encoded compiler artifact. An example of a solc compiler artifact is shown in Listing 4.2. The `sourceList` is an array of the Solidity source files passed into solc. `contracts` is an object containing key-value pairs, where the key has a format `<Solidity source file path>:<contract name>`, and the value is an object containing information about the contract. `bin` contains the contract binary, encoded as a hex string. `srcmap` contains the source map of the contract, encoded as a string with the scheme described in subsection 4.3.1. `bin-runtime` and `srcmap-runtime` contains the same information about the contract, but for the "runtime code" only. The runtime code is essentially the code actually placed on the blockchain, which omits code in the constructor. The full code is only relevant to the creation of contracts. The runtime code is executed for all contract method calls.

```
{
  "contracts": {
    "/path/to/Contract.sol:ContractName": {
      "bin": "6080604[...truncated...]",
      "bin-runtime": "6080604[...truncated...]",
      "srcmap": "26:22771:0:-;;;45:9991;8:9:-1;[...truncated...]",
      "srcmap-runtime": "26:22771:0:-;;;10174:4;10168;[...truncated...]"
    },
  },
  "sourceList": [
    "/path/to/Contract.sol"
  ],
  "version": "0.4.26+commit.4563c3fc.Darwin.appleclang"
}
```

Listing 4.2: Example compiler artifact generated by solc.

4.3.1 Source map encoding

solc encodes the source map of a contract as a string [26]. Since the encoding scheme is sufficiently generic, our Flint source maps also follow the same scheme, so that we can reuse the encoding and decoding logic.

Consider a contract that compiles into n EVM instructions. Then, the source map of the contract would contain n mappings, separated by "`;`"s. Each mapping contains 4 fields, in the format `<start>:<length>:<source index>:<jump type>`. The i^{th} source map entry corresponds to the i^{th} EVM instruction. The start, length, and source index describes the location of the code fragment within the source code that corresponds to the generated EVM instruction. The jump type can be either `i`, `o`, or `-`, which describes whether the instruction jumps **i**nto a function, **o**ut of a function, or is a generic jump generated by control flow statements (e.g. if, for, etc.)

Example. Let's say we have a source mapping `4:42:0:i`. The following information is encoded in this mapping:

- `start`: The source location of the instruction starts from the 4th character within the source file.
- `length`: The source location spans 42 characters.
- `source index`: The source location is in the 0th file of the compiler artifact's `sourceList`.
- `jump type`: This instruction jumps into a Solidity function.

Example. Another possible mapping is `1:100:-1:-`. The `source index` here is -1, which signifies that the instruction does not correspond to any source location, i.e. it is from compiler-generated code. In this case, its `start` and `length` do not carry any meaning.

4.3.1.1 Compression

Since adjacent source mappings often contain the same or partially the same information, the source map is compressed according the following rules:

1. If a field is unchanged from the preceding mapping, it is replaced by an empty string.
2. Empty fields at the end of a mapping are omitted.

Example. Consider the following uncompressed mapping (spaces added for legibility).

```
1:2:3:- ; 1:2:0:- ; 10:20:0:-
```

Applying the first rule, we have

```
1:2:3:- ; ::0:    ; 10:20::
```

Applying the second rule, we have

```
1:2:3:- ; ::0     ; 10:20
```

As we can see, redundant information is removed. The decompression process can be done by reversing these two steps.

4.3.2 Merging

$$\textit{YULSourceRange} : (\text{Start}, \text{Length})$$
$$\textit{solc map} : \text{EVMInstrIndex} \to (\text{YULSourceRange}, \text{SourceIndex}, \text{JumpType})$$
$$\textit{Intermediate map} : \text{YULSourceRange} \to \text{FlintSourceLocation}$$
$$\textit{Merged map} : \text{EVMInstrIndex} \to \text{FlintSourceLocation}$$

For each instruction index in the solc source map, we find the corresponding FlintSourceLocation in the intermediate map by matching the YULSourceRange in both maps. An exact match is not required. We find the smallest possible range in the intermediate map that "contains" the instruction's range and use that, where "contains" is defined as follows.

$$\text{range1 } \textbf{contains } \text{range2} \iff (\text{range1.start} \leq \text{range2.start}) \wedge (\text{range1.length} \geq \text{range2.length})$$

This is so that we can fallback to an enclosing source range if an exact match is not found.

4.3.3 Fixing jump types

The solc compiler only generates the correct jump types for Solidity functions, but not for YUL functions. We have to inject into the merged source map the appropriate jump types with respect to Flint source code.

To do this, we search the Flint AST to find all function calls. We mark the instructions generated by functions calls as `i` (jump into function). Then, we search for all function declarations, and for each declaration, we mark the last instruction it generates as `o` (jump out of function).

4.4 Extra contract metadata

To facilitate state inspection, we augment the solc compiler artifact format with a metadata object for each contract. Inside each metadata object we store information about the contract's state variables. It is simply an array of the contract's state variables' name, type, and size (if the variable is a fixed array type), in the order the are declared in the contract. This is sufficient for the debugger to work out the layout of variables in the contract's storage.

Also, if the contract uses type states, the names of the states are also added to the metadata, so that the debugger can convert a contract's type state from its raw value to its name.

```
contract ContractName (TypeState1, TypeState2, TypeState3) {
  var variable1: Bool
  var variable2: Int[4]
  var variable3: Int
}
```

```
{
  "contracts": {
    "ContractName": {
      "bin": "...",
      "bin-runtime": "...",
      "srcmap": "...",
      "srcmap-runtime": "...",
      "metadata": {                // new metadata object
        "storage": [
```

```
          {
            "name": "variable1",
            "type": "Bool"
          },
          {
            "name": "variable2",
            "type": "Int[4]",
            "size": 4
          },
          {
            "name": "variable3",
            "type": "Int"
          }
        ],
        "typeStates": ["TypeState1", "TypeState2", "TypeState3"]
      }
    },
  },
  // ...
}
```

4.5 Concluding remarks

We extended the Flint compiler to be able to output a source map and contract metadata during compilation, so that it can be used by the debugger to look up source location of generated EVM instructions. To do so without restructuring and changing the logic of the existing Flint compiler, we introduced the CodeFragment API which is designed to attach source location information onto IR code segments, with only light refactorings to the codebase. CodeFragment API is also used to generate an intermediate source map, which is then merged with the source map generated by the Solidity compiler, solc, to give us the final EVM-to-Flint source map.

Chapter 5

VS Code Debugger Extension

5.1 Overview

In addition to the command line interface, the Flint debugger also provides a graphical interface in the form of a Visual Studio Code (VS Code) extension.

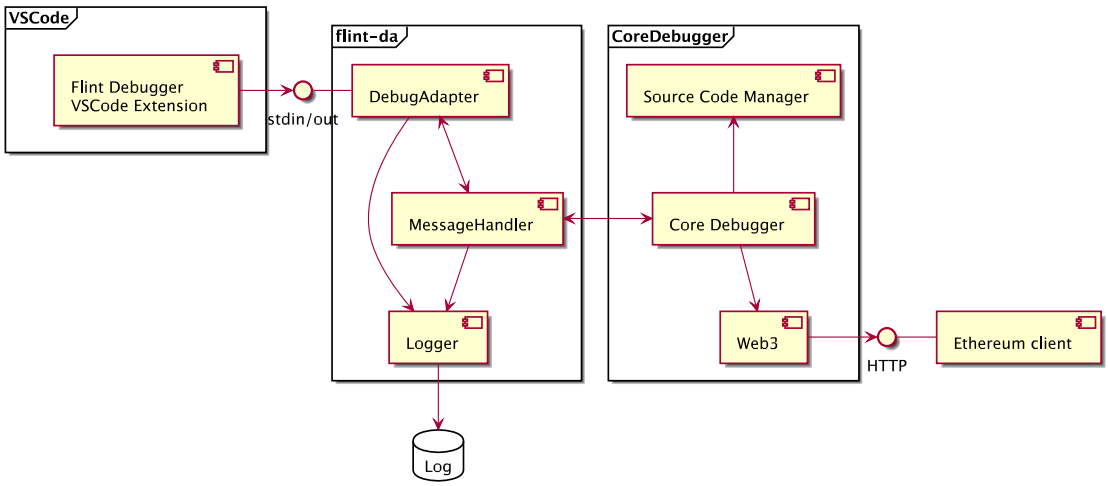


Figure 5.1: VS Code Flint debugger extension architecture.

Figure 5.1 shows the components that power the extension. The extension itself is a thin layer written in TypeScript that tells VS Code how it should communicate with the Flint debug adapter, flint-da, which implements the Debug Adapter Protocol (DAP) (see subsubsection 2.1.4.1). This thin layer is the only component in the entire project that is written in a language other than Swift, as it is a requirement by VS Code. The reasoning behind why all other components are written in Swift is explained in section 3.2.

Communication between VS Code and any debug adapter can be done via any of these 3 methods:

- Via the standard input/output of the adapter, if the adapter is an executable
- Via a port, if the adapter is run as a server
- Via direct function calls, if the adapter is written in JavaScript/TypeScript

Since the debug adapter is written in Swift, the third option is not possible. For simplicity, the adapter is implemented as an executable flint-da which reads from stdin and writes to stdout.

5.2 Extension

The extension's role is to tell VS Code how it should connect to flint-da. It essentially consists of one TypeScript file and one package.json manifest file, which are sufficient to specify the following:

- Settings for the extension, which are configurable in VS Code's settings page (see Figure 5.2):
  - Location of the flint-da executable
  - Enable flint-da's logging mechanism
  - Log file flint-da should log to (if logging enabled)
  - Logging level (if logging enabled)
- The arguments to pass to the debug adapter when launched
  - Transaction hash to debug
  - URL of the Ethereum client to use

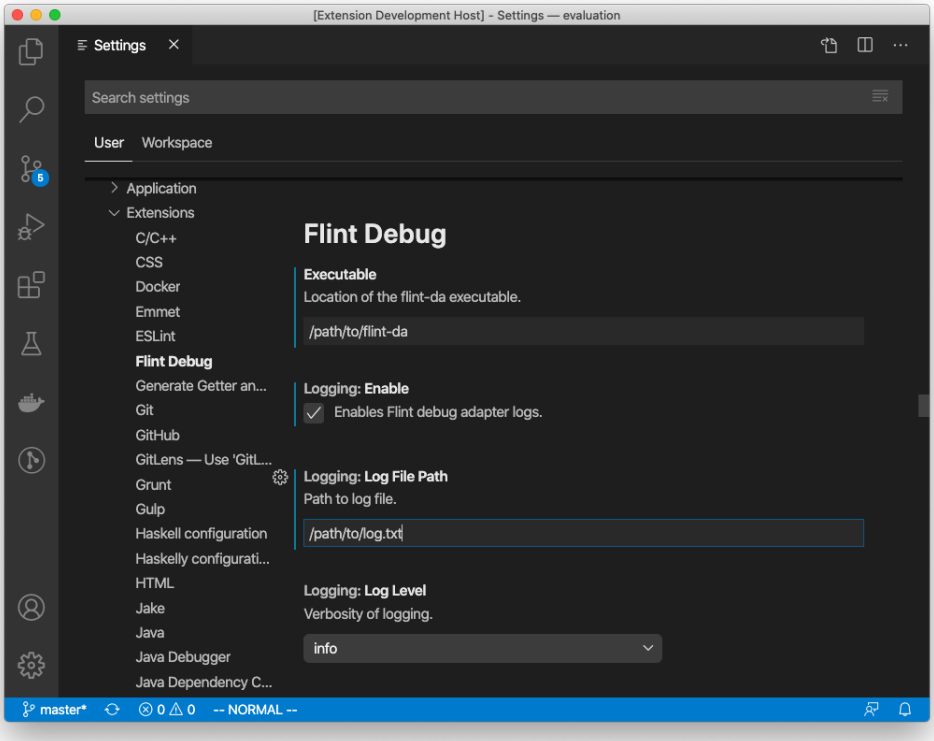


Figure 5.2: Flint debugger settings.

5.3 SwiftDAP

There are libraries/SDKs for DAP in other languages such as JavaScript and C#[1], but unfortunately not for Swift. Therefore, we implemented a DAP library in Swift, *SwiftDAP*[2], which provides the basic constructs to work with the protocol in Swift. It is designed to be a generic representation of the protocol, such that it could be used to build not only debug adapters, but also the IDE/client-side debug adapter protocol implementation.

5.3.1 The protocol

```
Content-Length: 120\r\n
\r\n
{
    "seq": 42,
    "type": "request",
    "command": "stepIn",
    "arguments": {
        "threadId": 3
    }
}
```

Listing 5.1: Example of a "step in" request.

Listing 5.1 shows an example of a DAP protocol message. It begins with a single Content-Length header, specifying the number of bytes in the message body. The message body follows the header, separated by 2 CRLF new line characters.

The message body is encoded in JSON. The fields "seq" and "type" are present in *all* protocol messages, the others are specific to the type of protocol message. "seq" represents the sequence number (or identifier) of the message. The "type" of the message is one of these 3: "request", "response", and "event". Request messages are sent from the IDE to the debug adapter [3]. The debug adapter responds back to request with "response" messages. "Event" messages are sent by the debug adapter to the IDE when certain debugger events occur, e.g. when it stops at a breakpoint or after a step action.

```
{
    "seq": 43,
    "type": "response",
    "request_seq": 42,
    "command": "stepIn",
    "success": false,
    "message": "An error occured",
    "body": {}
}
```

Listing 5.2: Example of a "step in" response messaage body.

```
{
    "seq": 44,
    "type": "event",
    "event": "stopped",
    "body": {
        "reason": "step"
    }
}
```

Listing 5.3: Example of a "stopped" event message body.

Requests. Request messages contain a "command" and an "arguments" object. Some examples of commands are *stepIn*, *launch*, *disconnect*, etc. The contents of the "arguments" object is determined by the type of command.

Responses. Response messages contain a "request_seq" and "command", which are, respectively, the sequence number and the command of the request it is responding to. "success" is a boolean

[1] https://microsoft.github.io/debug-adapter-protocol/implementors/sdks/
[2] https://github.com/noellee/SwiftDAP
[3] "Reverse requests" also exist, where the debug adapter sends certain requests to the IDE, but

specifying if the request has succeeded, "message" contains an error message if the request is not successful. "body" contains the data returned as the result of the request, which is specific to the request command.

Events. Event messages contains an "event", which specifies the type of the event, and "body", which includes more information about the event. Some examples of events are *stopped*, *initialized*, *terminated*, etc.

5.3.2 API

Here we will discuss how DAP protocol messages as described in subsection 5.3.1 are represented in SwiftDAP, as well as components provided by SwiftDAP to make working with DAP easier.

5.3.2.1 ProtocolMessage

Protocol messages are represented as enums with associated values. Listing 5.4 shows the equivalent Swift representation of the example request in Listing 5.1. Response and event messages follow a similar structure.

```
var msg: ProtocolMessage
msg = ProtocolMessage.request(
    seq: 42,
    request: RequestMessage.stepIn(StepInArguments(threadId: 3))
)
// exactly the same, but with shorthands
msg = .request(seq: 42, request: .stepIn(StepInArguments(threadId: 3)))
```

Listing 5.4: Example of a request `ProtocolMessage`.

5.3.2.2 DebugAdapter

The DebugAdapter class reads continuously from a given input file (standard input by default), parses and deserializes incoming messages, then delegates the handling of successfully parsed messages to its ProtocolMessageHandler. The DebugAdapter also passes a ProtocolMessageSender function to the ProtocolMessageHandler, which it can use to send messages to the DebugAdapter's output.

```
// typealias MessageHandlerFactory
//     = (ProtocolMessageSender) -> ProtocolMessageHandler

let input: FileHandle = ...
let output: FileHandle = ...
let logger: Logger = ...
let messageHandlerFactory: MessageHandlerFactory = ...

// run the debug adapter
DebugAdapter(input, output, logger)
    .withMessageHandler(messageHandlerFactory)
    .start()
```

Listing 5.5: Example usage of DebugAdaapter.

```
class ExampleMessageHandler: ProtocolMessageHandler {
    func handle(message: ProtocolMessage) {
        switch message {
        case .request(let seq, let request):
            // do something about the request
```

```
6           case .response(let seq, let response):
7               // do something about the response
8           case .event(let seq,  event):
9               // do something about the event
10          }
11      }
12  }
```

Listing 5.6: Example ProtocolMessageHandler implementation.

5.3.2.3 Logger

Optionally, the DebugAdapter takes in a logger. This is especially useful during development. Since the debug adapter executable is directly launched by VSCode, we cannot see the text sent to and from the adapter, making it hard for a debug adapter developer to debug the debug adapter.

The user can define a logger by implementing the following protocol (interface). As an example, the Flint debugger implements a logger that logs to a file.

```
1  public protocol Logger {
2    func log(_ message: CustomStringConvertible)
3    func log(_ message: CustomStringConvertible, level: LoggingLevel)
4  }
5
6  public enum LoggingLevel {
7    case error, warning, info, debug
8    // ...
9  }
```

Listing 5.7: Logger interface. Note: `CustomStringConvertible` is a builtin Swift protocol which has a `toString()`-like method.

5.4 flint-da

flint-da is a debug adapter for Flint created during this project, in the form of an executable. It acts as a wrapper around the core debugger which we have discussed in chapter 3.

Using SwiftDAP, flint-da listens to incoming DAP requests and responds to them. When the core debugger generates an event (e.g. when a breakpoint is hit, when debugging is done, etc.), the event is relayed to VS Code as a DAP event message.

Figure 5.3 shows an example of how flint-da communicates back and forth with VS Code within a debug session. When the user initiates a debug session, VS Code launches the flint-da executable. It then sends the adapter an "initialize" request to which the debug adapter responds with an "initialize" response. These "initialize" messages inform each other of the DAP features they support. Then, VS Code sends a "launch" request, with arguments needed by the debugger, such as the transaction hash to debug. The debug adapter responds with a "launch" response as an acknowledgement, initializes the core debugger, then sends VS Code an "initialized" event once the core debugger is done initializing.

At this point, the user is able to use debugging controls in VS Code to start debugging. All user actions follow the same pattern: (1) user interacts with VS Code; (2) VS Code sends the debug adapter a request; (3) the adapter acknowledges the request immediately; (4) the adapter calls the relevant function of the core debugger; (5) the core debugger emits an event; (6) the adapter captures the event and sends it to VS Code; (6) VS Code receives the event and updates the UI accordingly to reflect the change.

5.5 Concluding remarks

We have integrated the Flint debugger with VS Code using the Debug Adapter Protocol (DAP). In the process of doing so, we built a DAP library, SwiftDAP, that can be used by any Swift programmer to easily build debug adapters and/or other tools on top of DAP.

We have used VS Code to demonstrate how one can integrate the Flint debug adapter with an existing editor. However, the adapter can easily be integrated with any editor/IDE that supports DAP, without having to be modified.

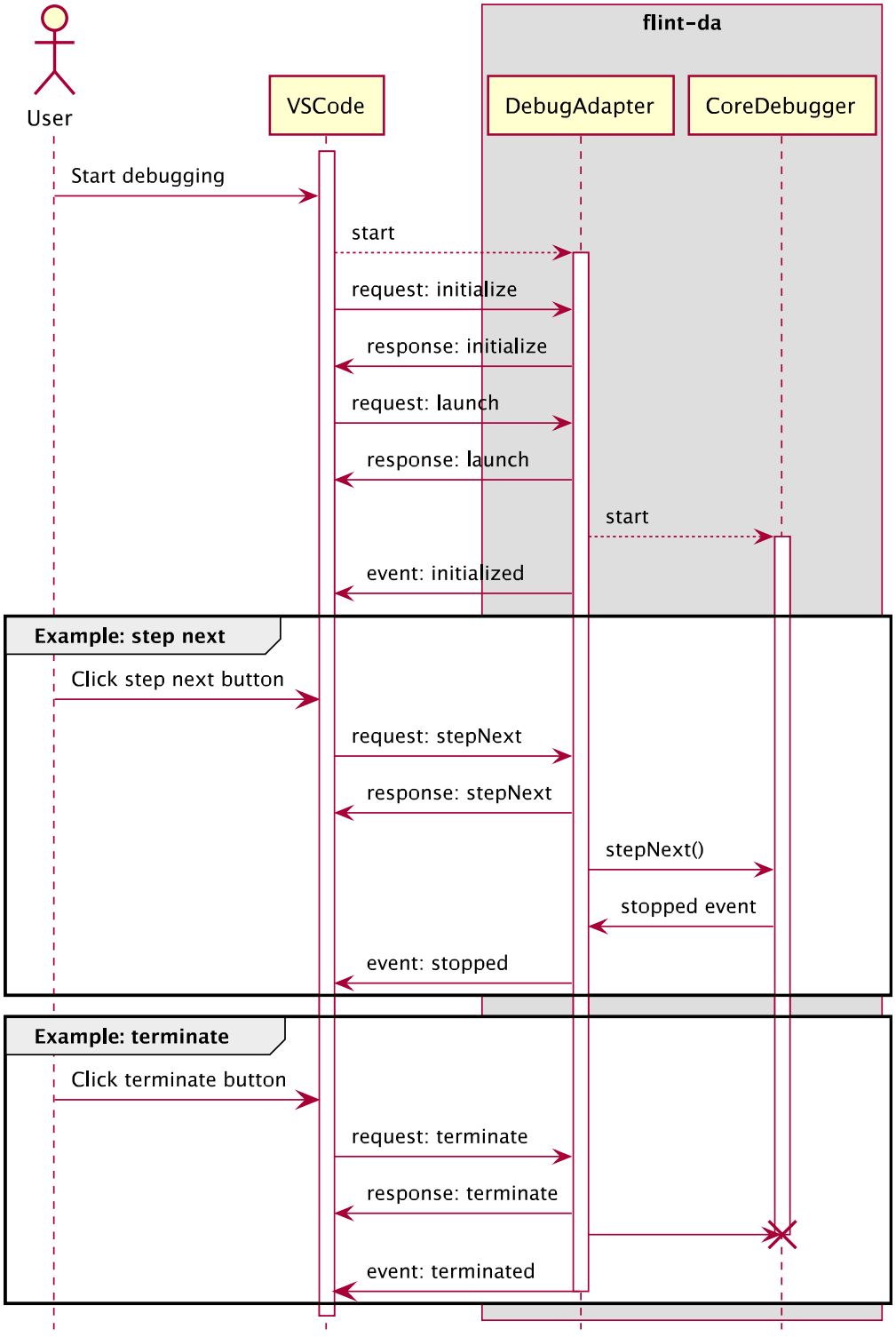


Figure 5.3: Example sequence diagram of a Flint debug session.

Chapter 6

Evaluation

In this chapter we evaluate the impact of our code changes to the Flint compiler, by looking at its test suite and measuring the performance of the compiler after the changes. We will also compare both the CLI and the GUI of our debugger against existing Solidity debuggers, Truffle (a CLI) and Remix (a GUI).

6.1 System specification

The following is the specification of the system used for this evaluation (unless specified otherwise).

	Version
OS	macOS 10.15.5
VS Code	1.45.1
Ganache	2.4.0.1317
solc	0.4.26+commit.4563c3fc.Darwin.appleclang
Truffle	5.1.12

6.2 Flint tests

As mentioned in previous sections, changes have been made to the Flint compiler to support debugging. The Flint project contains a suite of syntax, semantic, and functional tests. All tests still pass after the implementation of new compiler features, including 5 new unit tests for CodeFragments. No compilers were harmed in the development of this debugger.

The tests are run on Travis CI, which is the continuous integration server used by the Flint project. Tests on Travis CI run on both Linux and macOS servers.

6.3 Compiler overhead

As we have changed the code generation process from using plain strings to using CodeFragments (see section 4.2), it is expected that some overhead to the compilation time would be added.

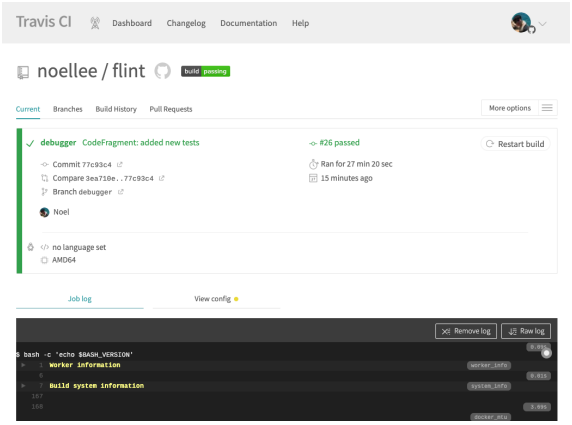


Figure 6.1: All automated tests pass on Travis CI[1].

6.3.1 Methodology

First, we compile the Flint project without applying our changes to obtain an executable $\text{flintc}_{\text{before}}$. Then, we apply our changes and compile the project again to obtain executable $\text{flintc}_{\text{after}}$.

Using both versions of flintc, we will compile a sample contract (Supermarket.flint, Listing B.1) and record the time taken to complete the compilation.

We compile the contract with the flags `--skip-verifier` and `--skip-holistic` such that our results are not influenced by the contract verification process, which itself takes some time to complete. The following is the compilation command and an example of the output it produces.

```
$ flintc --skip-verifier --skip-holistic Supermarket.flint

Produced binary in /path/to/directory/bin.
```

To record command's execution time, we use Unix's `time` utility. This outputs 3 extra lines: the total wall clock (real world) time taken (`real`), CPU time spent in user mode (`user`), and CPU time spent in kernel mode (`sys`) [27]. In this evaluation we will compare the total CPU time used by the flintc process (user + sys) since we are only interested in timing changes within flintc.

```
$ time flintc --skip-verifier --skip-holistic Supermarket.flint

Produced binary in /path/to/directory/bin.

real    0m0.390s
user    0m0.304s
sys     0m0.014s
```

We will run and time the compilations 5 times each for $\text{flintc}_{\text{before}}$ and $\text{flintc}_{\text{after}}$, and take the mean user + sys time for those 5 executions. These two means are the final values we will be comparing.

Source map generation. To measure the overhead of generating a source map, we repeat the same steps (only on $\text{flintc}_{\text{after}}$) with the flintc flag `--emit-srcmap`:

```
$ time flintc --skip-verifier --skip-holistic --emit-srcmap \
    Supermarket.flint
```

6.3.2 Results

Table 6.1 contain the results of running flintc as described above, with source map generation disabled. The mean execution time saw a 6.18% increase (0.019s) from 0.3074s to 0.3264s.

#	Type	Time (s)	user + sys (s)
1	real	0.751	
	user	0.296	0.313
	sys	0.017	
2	real	0.381	
	user	0.293	0.307
	sys	0.014	
3	real	0.374	
	user	0.291	0.304
	sys	0.013	
4	real	0.373	
	user	0.289	0.304
	sys	0.015	
5	real	0.379	
	user	0.296	0.309
	sys	0.013	
		Mean (s)	**0.3074**

(a) Before

#	Type	Time (s)	user + sys (s)
1	real	0.390	
	user	0.304	0.318
	sys	0.014	
2	real	0.391	
	user	0.306	0.32
	sys	0.014	
3	real	0.393	
	user	0.309	0.324
	sys	0.015	
4	real	0.391	
	user	0.302	0.317
	sys	0.015	
5	real	0.488	
	user	0.329	0.353
	sys	0.024	
		Mean (s)	**0.3264**

(b) After

Table 6.1: Execution times before and after making changes to flintc (no source map generation).

Table 6.2 shows the results of running flintc with source map generation enabled. There is a slowdown of more than 8x when the option is enabled, adding 2.6 seconds to the compilation time.

#	Type	Time (s)	user + sys (s)
1	real	3.166	
	user	2.514	2.993
	sys	0.479	
2	real	2.989	
	user	2.458	2.906
	sys	0.448	
3	real	2.982	
	user	2.457	2.905
	sys	0.448	
4	real	2.972	
	user	2.452	2.901
	sys	0.449	
5	real	3.047	
	user	2.507	2.966
	sys	0.459	
		Mean (s)	**2.9342**

Table 6.2: Execution times before and after making changes to flintc (with source map generation).

6.3.3 Summary

In general, the performance of flintc is only slightly affected by a slowdown of 6.18%. However, source map generation is very slow, **significantly** adding to the execution time by a huge factor if enabled (800% slowdown). Therefore, at this stage of development, it is essential to have source map generation disabled by default, and only enabled explicitly if necessary to avoid the overhead when the feature is not needed.

6.4 Comparison with Solidity debugging tools

Since there is no existing Flint debugger, we will compare our debugger with Solidity debuggers. The Solidity debuggers will act as a benchmark for features a smart contract developer would need. They provide the closest comparison possible as both languages use the same execution model and are bound by the same limitations set on by Ethereum. Furthermore, compared to other smart contract languages (e.g. Vyper etc.), Solidity is more mature as a project and has the largest user base, hence the tools developed for it are better supported and are richer feature-wise.

We chose Truffle and Remix as our Solidity debuggers since they both have large user bases, and their interfaces are very similar to the Flint debugger interfaces we implemented. For reference, the Truffle debugger has been in development since 2017 and Remix's debugger has been in development since 2018, according to their GitHub repositories [28, 29]. Both have been worked on by multiple people. We will make the following comparisons:

1. flint-debug CLI tool vs. Truffle debug
2. VSCode Flint debug extension vs. Remix IDE

A better comparison to our Flint debugger extension than Remix IDE would be Etheratom, an extension for the code editor Atom which makes of Remix's backend. However, we are unable to test Etheratom's debugger functionality due to a compilation bug[2]. The extension gets stuck when it tries to compile our test contracts. Because the contracts cannot be successfully compiled, it cannot proceed to the debugging process. According to Etheratom's website, its debugger provides the same functionality as Remix (as it uses Remix's backend as its backend).

6.4.1 Methodology

We will use a sample Flint contract "`FlintSupermarket`" in this evaluation. This contract is designed to exercise different basic control flow constructs such as function calls, for loops etc, as well as Flint-specific features like caller groups and type states (see section 2.3). A functionally equivalent contract, "`SolSupermarket`", is also implemented in Solidity, which will also be used in this evaluation.

Table 6.3, Table 6.4, and Table 6.5 outline the functionality provided by the Supermarket contract. The full implementations of the contract in both languages are available in B.1. They are also published on GitHub in this repository: https://github.com/noellee/flint-debugger-evaluation, which also contains scripts that automates some steps of the evaluation process.

Name	Type	Description
`staff`	`address[3]`	Ethereum addresses of staff members.
`manager`	`address`	Ethereum addresses of the manager.
`stock`	`int`	The number of "items" in stock.
`customers`	`int`	The number of customers who visited the supermarket.

Table 6.3: State variables in the Supermarket contract.

Name	Description
`incrementCustomers(count)`	Increment `customers` by `count`.
`resetCustomers()`	Reset `customers` to 0.

Table 6.4: Private methods in the Supermarket contract.

[2] https://github.com/0mkara/etheratom/issues/294, tested on Atom 1.47.0, Etheratom 4.6.0.

Name	Description	Requirements
`deposit()`	Deposit Ether into the contract's balance.	Any state, anyone
`payday()`	Loop through all `staff` and send them Ether.	Any state, manager only
`employ(n, employee)`	Set `staff[n] = employee`.	
`buy(amount)`	Remove `amount` from `stock` if there is enough stock and `amount <= 2`. Then, call `incrementCustomers(1)`.	Opened, anyone
`close()`	Call `resetCustomers()` then change state to Closed.	Opened, manager only
`open()`	If there is enough stock, change state to Opened.	Closed, manager only
`restock(amount)`	Add `amount` to `stock`.	Closed, manager or staff

Table 6.5: Description of the Supermarket contract's public interface.

Steps taken. For both contracts, these are the steps taken to obtain Ethereum transactions for each contract which we can evaluate our debuggers against.

1. Compile the Flint contract (with `-emit-srcmap -emit-ir`)
2. Deploy the Flint contract's IR code (YUL code wrapped in a Solidity file) to Ganache
3. Compile the Solidity contract
4. Deploy the Solidity contract to Ganache
5. For both contracts, call contract methods in the following sequence:
   (a) manager: `supermarket.deposit()`, with 1,000,000 Wei
   (b) manager: `supermarket.employ(0, <employee0>)`
   (c) manager: `supermarket.employ(1, <employee1>)`
   (d) manager: `supermarket.employ(2, <employee2>)`
   (e) employee0: `supermarket.restock(1005011)`
   (f) manager: `supermarket.open()`
   (g) customer: `supermarket.buy(2)`, with 20 Wei
   (h) manager: `supermarket.payday()`
   (i) manager: `supermarket.close()`

Once we have transaction hashes, $\{tx_a^{flint}, ..., tx_i^{flint}, tx_a^{sol}, ..., tx_i^{sol}\}$, we debug the transaction for each contract using our Flint debugger and the reference Solidity debugger, and compare the debugging experience.

1. Debug each tx_x^{flint} against Flint source code using **our Flint debugger**
2. Debug each tx_x^{sol} against Solidity source code using a **Solidity debugger**
3. Debug each tx_x^{flint} against YUL IR code (generated by the Flint contract) using a **Solidity debugger**

These 3 debugging sessions capture 3 different scenarios:

1. Debugging a Flint contract with our dedicated Flint debugger
2. Debugging a Solidity contract — this gives us a reference of what a standard smart contract debugging experience would be like.
3. Debugging a Flint contract, where there is no Flint debugger

6.5 CLI Comparison

6.5.1 flint-debug

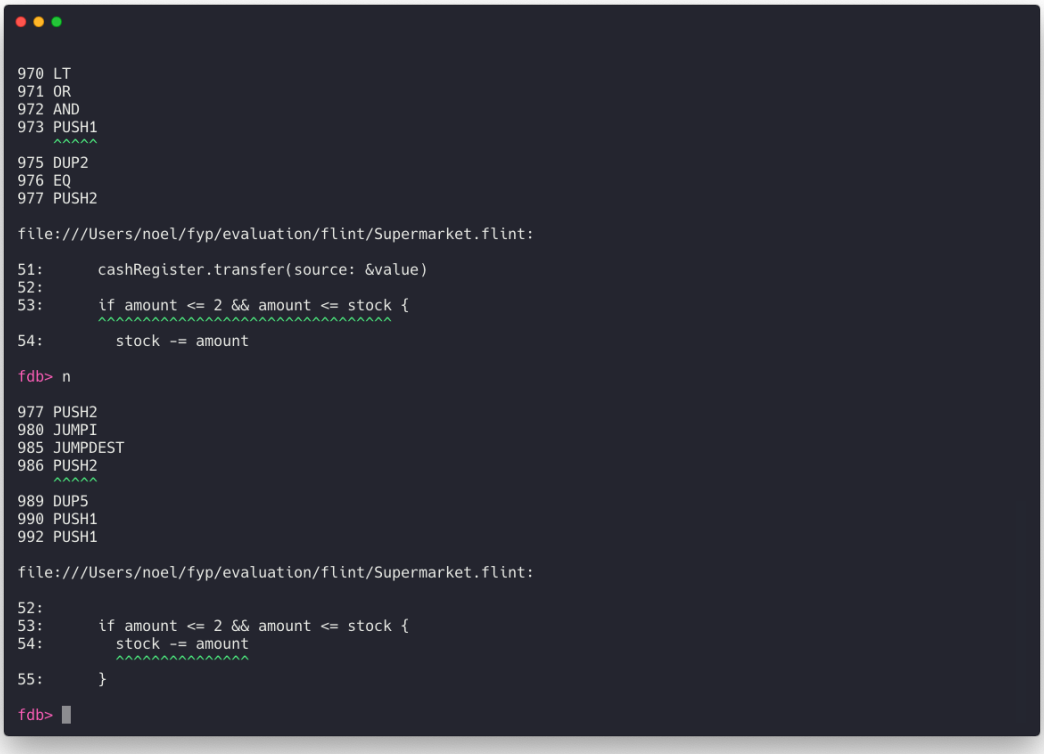


Figure 6.2: Debugging with flint-debug.

In this section we will discuss flint-debug's support for debugger features.

Launching the debugger. The following command starts a Flint debug session, where 0xa723b... is the transaction to be debugged.

```
flint-debug 0xa723b...
```

Optionally, `--artifacts` and/or `--rpc-url` can be passed in as arguments to specify the directory containing compiler artifacts and the Ethereum client's url respectively, if they are different from the default.

Breakpoints. A breakpoint can be set by entering the command "`b <line number>`", and can be removed by the command "`B <line number>`". For example, "`b 42`" sets a breakpoint at line 42, and "`B 42`" removes that breakpoint.

Stepping. Table 6.6 describes all stepping commands supported by flint-debug.

State inspection. The state of the contract can be inspected by entering the command "`p <category>`", as shown in Figure 6.3, where `<category>` is any one of the following: `stack`, `memory`, `storage`, `flint`, `evm` (see Table 6.7).

Command	Description
`n`	**Step next**
`i`	**Step into**
`o`	**Step out** of the current function.
`;`	**Step instruction**
`c`	**Continue** until a breakpoint is hit or there is no more instructions to step.
`r`	Toggle **reverse** debugging. All the above commands *except* step out are executed in reverse when reverse debugging is on.

Table 6.6: flint-debug stepping commands.

It is worth noting that "`p storage`" does not always show *all* state variables of the contract.

Category	Description
`stack`	Raw contents in the contract's stack.
`memory`	Raw contents in the contract's memory.
`storage`	State variables in the contract's storage.
`flint`	Flint-specific information. Currently displays only one piece of information: the type state of the contract.
`evm`	EVM instruction information: op code, program counter (pc), gas, gas cost, and jump type.

Table 6.7: flint-debug state inspection categories.

Expression evaluation. Not supported.

6.5.2 Truffle debug

Launching the debugger. The following command starts a Truffle debug session, where 0xa723b... is the transaction to be debugged.

```
truffle debug 0xa723b...
```

Truffle debug accepts exactly one argument: the transaction hash to debug. Additional configuration (e.g. Ethereum network, Solidity compiler, project directory structure etc.) is specified in a configuration file, truffle-config.js, which is located at the root of the Truffle project. Listing 6.1 shows an example of a truffle-config.js file where a development Ethereum network and solc version version are specified.

```
module.exports= {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 8545,
      network_id: "*",
    },
  },
  compilers: {
    solc: {
      version: "0.4.26",
    }
  }
}
```

Listing 6.1: Example truffle-config.js

```
fdb> p stack
0: 0000000000000000000000000000000000000000000000000000000d96a094a
1: 0000000000000000000000000000000000000000000000000000000000000001
2: 000000000000000000000000000000000000000000000000000000000000013a
3: 0000000000000000000000000000000000000000000000000000000000000002
4: 0000000000000000000000000000000000000000000000000000000000000060
5: 0000000000000000000000000000000000000000000000000000000000000001
6: 0000000000000000000000000000000000000000000000000000000000000001
```

(a) Raw contents in the stack.

```
fdb> p memory
0: 0000000000000000000000000000000000000000000000000000000000000000
1: 0000000000000000000000000000000000000000000000000000000000000000
2: 0000000000000000000000000000000000000000000000000000000000000080
3: 0000000000000000000000000000000000000000000000000000000000000000
```

(b) Raw contents in the memory.

```
fdb> p storage
stock: 1005011
cashRegister: 1000020 Wei
```

(c) Contract state variables in storage.

```
fdb> p flint
state: Opened
```

(d) Contract type state.

```
fdb> p evm
op: PUSH2
pc: 986
gas: 55299
gasCost: 1
jump: -
```

(e) Ethereum metadata

Figure 6.3: State inspection in flint-debug.

6.5.2.1 Debugging a Solidity contract

Breakpoints. Breakpoints can be set on any line in the contract, and works as expected, i.e. when the user issues a "continue" command, the debugger breaks if/when the breakpoint is hit.

Stepping. Truffle supports 5 types of stepping commands [30]:

1. **Step next.** Steps to the next Solidity subexpression. Equivalent to flint-debug's "step into".
2. **Step instruction.** The most fine-grained stepping option - uses EVM instructions as a unit of stepping. Equivalent to flint-debug's "step instruction".
3. **Step over.** Steps over the current line. No direct equivalent flint-debug command.
4. **Step into.** Similar to step next, except on a method call, it jumps straight into the callee method body, instead of stepping through the evaluation of method call arguments. No direct equivalent flint-debug command.
5. **Step out.** Run through the current method and stop at the calling method. Equivalent to flint-debug's "step out". Equivalent to flint-debug's "step out".

State inspection. It is possible to view transaction details and other Ethereum metadata, as well as the raw contents of the contract's stack memory and storage. In addition, Truffle shows the state variables and local variables of the Solidity contract. In Figure 6.5a, the first 4 variables are defined in the contract. `enoughCash` is a locally defined variable; `amount` is the argument of `buy()`; `state`, `staff`, `manager`, `stock` and `customers` are state variables of `SolSupermarket`.

As shown in Figure 6.5b, instruction information and raw contents of the stack, memory, and storage can also be displayed.

However, state variables are not always accurate. Variables only become accurate **after** they are written to. In our example, before `customers` is written, its value is *always* shown as 0. This

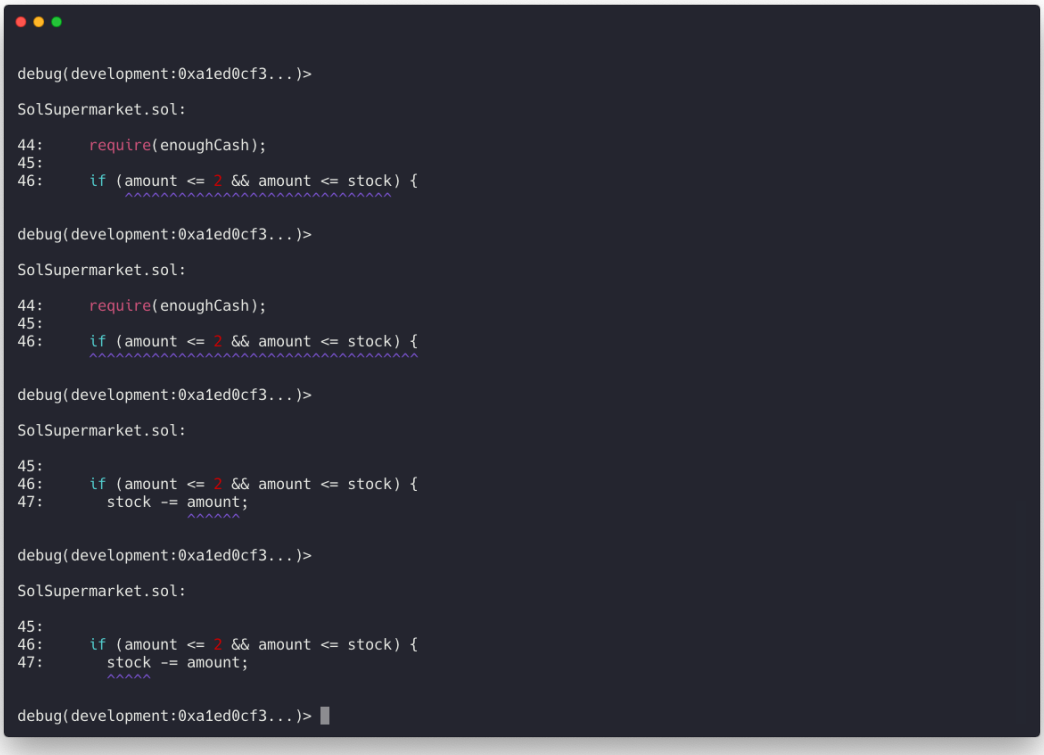


Figure 6.4: Debugging with Truffle.

also applies to arrays, i.e. all elements that are not read/written in the transaction are shown as 0. Consider the situation where we call `buy()` for the second time. When we debug this second transaction and land on the line `incrementCustomers(1)`, the value of `writeCount` shown in Truffle jumps from 0 to 2. This can be incredibly misleading to the user.

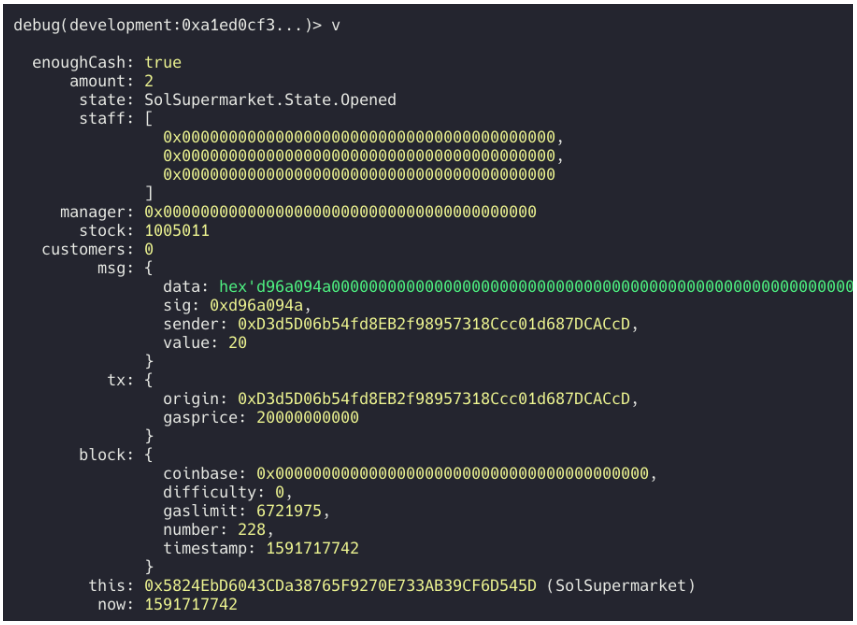
Expression evaluation. As seen in Figure 6.6, Truffle supports basic expression evaluation using contract variables. Expressions can be "watched", whereby they will be evaluated and displayed on every step.

6.5.2.2 Debugging a Flint contract

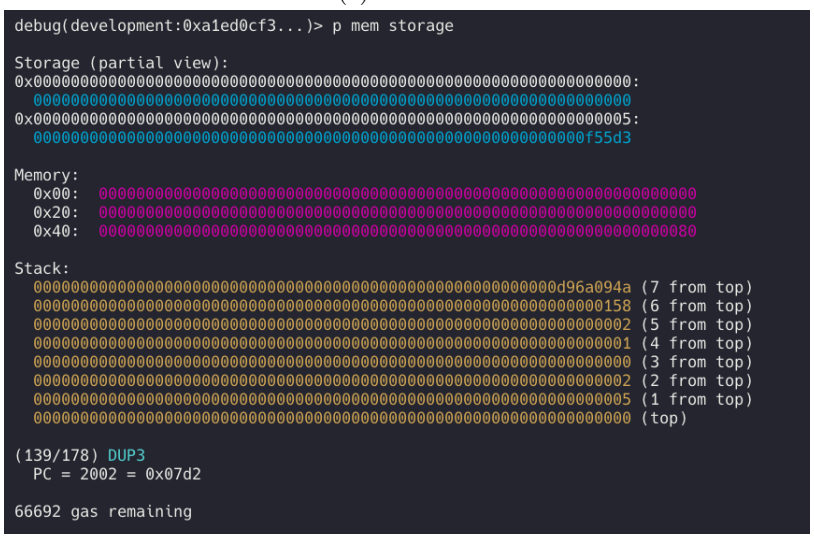
Truffle supports debugging YUL code (with limited capabilities), as it is embedded in a Solidity file. However, since the code is generated by a compiler, i.e. Flint, and not written by a person, it is not very readable. To be able to understand the YUL code, one must be familiar with how the compiler generates YUL code from Flint code. Even then, there are many additional checks placed in the code by the compiler, which obfuscates domain specific code even further.

As an example, Listing 6.2 shows the `FlintSupermarket.buy()` function as implemented in Flint. Listing 6.3 shows the snippet of code from the generated YUL code for the same function. The logic expressed by the YUL code is not immediately obvious, compared to that by the Flint code.

```
46 FlintSupermarket @(Opened) :: (any) {
47   @payable
48   public func buy(amount: Int, implicit value: Wei) mutates (stock,
       ↪ customers) {
49     let enoughCash: Bool = value.getRawValue() == amount * 10
50     assert(enoughCash)
51     cashRegister.transfer(source: &value)
```



(a) Variables



(b) Contents in the contract's stack, memory, and storage.

Figure 6.5: State inspection in Truffle, with a **Solidity contract**.

```
52
53      if amount <= 2 && amount <= stock {
54        stock -= amount
55      }
56
57      incrementCustomers(count: 1)
58    }
59  }
```

Listing 6.2: `FlintSupermarket.buy()` implementation.

```
1   pragma solidity ^0.4.25;
2
3   contract FlintSupermarket {
        ...
309   function () public payable {
310     assembly {
        ...
```

(a) Evaluating a JavaScript expression involving contract variables.

(b) Expressions can be added as a "watch". e.g. `stock*5`.

Figure 6.6: Truffle expression evaluation.

```
428       function FlintSupermarket$buy$Int_Wei(_amount)  {
429         let _value := flint$allocateMemory(32)
430         Wei$init$Bool_Int(_value, 1, 1, callvalue())
431         let _enoughCash := eq(Wei$getRawValue(_value, 1), flint$mul(_amount
        ↪ , 10))
432         Flint$Global$assert$Bool(_enoughCash)
433         Wei$transfer$$inoutWei(add(0, 6), 0, _value, 1)
434         switch and(or(lt(_amount, 2), eq(_amount, 2)), or(lt(_amount, sload
        ↪ (add(0, 4))), eq(_amount, sload(add(0, 4)))))
435         case 1 {
436           sstore(add(0, 4), flint$sub(sload(add(0, 4)), _amount))
437         }
438         FlintSupermarket$incrementCustomers$Int(1)
439       }
      ...
877     }
878   }
879 }
```

Listing 6.3: Snippet of the YUL code (embedded in Solidity) generated from `FlintSupermarket.buy()`.

Breakpoints. A limitation of debugging a Flint contract with Truffle is that it is not possible to set breakpoints on YUL code. In Listing 6.3, we can see that the `buy()` function is wrapped inside an `assembly { ... }` block. When we try to set a breakpoint on any line inside an `assembly { ... }` block, Truffle fails to set the breakpoint at the correct location, as seen in Figure 6.7.

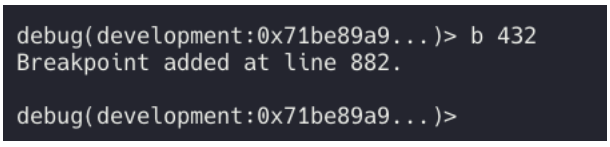


Figure 6.7: Truffle does not support YUL breakpoints. When attempting to set a breakpoint at a YUL position (line 432), the breakpoint is incorrectly set at line 882 instead.

Stepping. Stepping generally works as expected with YUL code. Reverse stepping is not supported. There are 5 types of "steps": step over, step into, step out, step next, and step instruction (the nuances of each step type is discussed in subsubsection 6.5.2.1). Step into and step out are designed to work in the context of Solidity code, so the debugger steps into and out of the Solidity function enclosing the YUL code, instead of a YUL function. Step over, step next, and step instruction works well with YUL code.

However, because of the volume of compiler-generated code, it takes significantly more steps to reach "interesting" parts of the code, i.e. the method called in the transaction. With the example contract, 65 "step next"s are needed to reach the start of the `buy()` function body, as opposed to only 1 when using flint-debug. Without the help of breakpoints, it is an impractical way to debug Flint contracts.

State inspection. The same Ethereum metadata as with debugging Solidity code is displayed (see subsubsection 6.5.2.1). However, user-defined variables are not shown.

Expression evaluation. Since Truffle does not support user-defined variables in YUL code, there are no variables to evaluate expressions with. Only "constant" expressions can be evaluated, such as `40 + 2`, or `"hello" + "world"` etc.

6.5.3 Summary

Interface. Visually, Truffle provides a richer user interface, as code is displayed on screen with syntax highlighting. In terms of controls, both debuggers are similar. Both command prompts accepts commands in the form of "`<command> <arguments>`".

Launching the debugger. The process of launching both debuggers are very similar, except for two slight differences.

1. **Configuration.** flint-debug accepts configuration parameters as command line arguments, while Truffle uses a configuration file. Both approaches have their own strengths. flint-debug gives the user more freedom to run the program anywhere, instead of requiring extra project setup before running. On the other hand, persisting the configuration on disk allows users to avoid having to pass in arguments every time they call the program, which is especially helpful with more complicated configurations.
2. **Loading time.** Truffle compiles all contracts on the launch of the debugger. Even with a caching mechanism, this takes a few seconds even for short contracts like the ones we use in this evaluation. In comparison, flint-debug's startup overhead is minimal and takes no more than one second to launch. This is because it assumes the contract has already been compiled.

Breakpoints. Breakpoints generally behave the same in both debuggers.

Stepping. Stepping commands are fairly similar for both debuggers, except for some semantic differences. flint-debug supports reverse debugging, which Truffle does not.

State inspection. Both support viewing the raw contents of the contract's stack and memory. However, when showing contract variables in storage, Truffle shows both unconfirmed and confirmed variables, which means it displays incorrect values sometimes. For flint-debug, we have instead opted to not show unconfirmed variables rather than showing incorrect values. We believe this would less confusing and misleading to the user.

Expression evaluation. Expression evaluation and variable watches are useful features provided by Truffle, especially for a CLI application since variables are not always shown on screen like in a GUI debugger. flint-debug does not support these features.

6.6 GUI Comparison

6.6.1 VSCode Flint debug extension

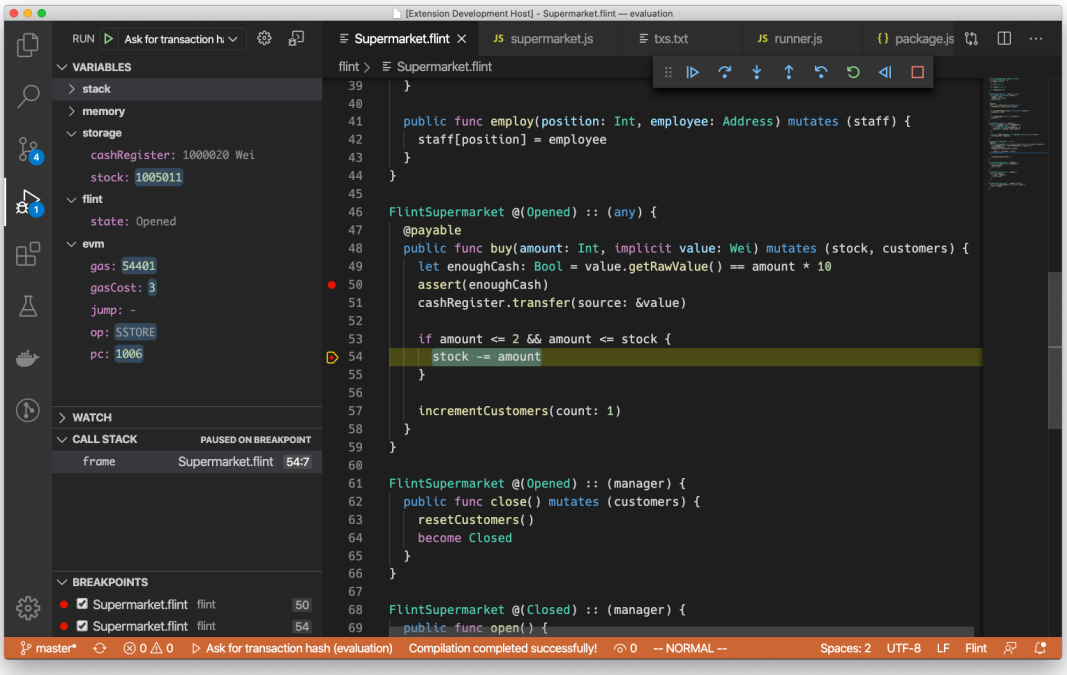


Figure 6.8: Debugging in VSCode.

Launching the debugger. VSCode requires the user to set up a `launch.json` file in their project directory before they can run/debug a program. This is where the user can specify launch configurations, i.e. how they want to launch a debug session. Listing 6.4 shows the default launch configuration for Flint.

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "flint",
            "request": "launch",
            "name": "Ask for transaction hash",
            "txHash": "${command:AskForTransactionHash}",
            "artifactDirectory": "${workspaceFolder}/bin",
            "rpcUrl": "http://localhost:8545"
        }
    ]
}
```

Listing 6.4: Sample `launch.json` file, with the default Flint debug configuration.

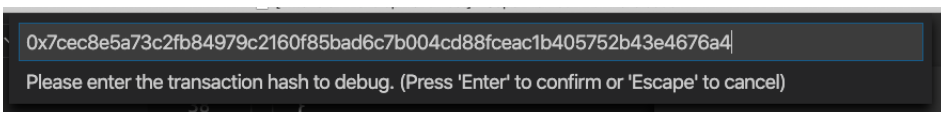


Figure 6.9: A pop-up asks the user for a transaction hash for the debug session.

Launch configurations allows users to customize their workflows to their liking. They can replace the value of `rpcUrl` with the url of their Ethereum client, if it is different from the default one. If

they don't want to get the "Please enter a transaction hash..." pop up, they can manually set the `txHash` in the configuration.

Breakpoints. Breakpoints can be set by clicking on line numbers in the main editor window, as seen in Figure 6.10.

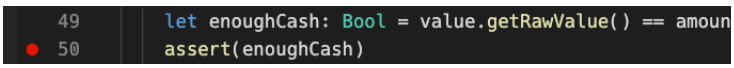


Figure 6.10: Breakpoints in VS Code.

Stepping. Stepping controls can be accessed at a bar on the top of the editor, as seen in Figure 6.11.

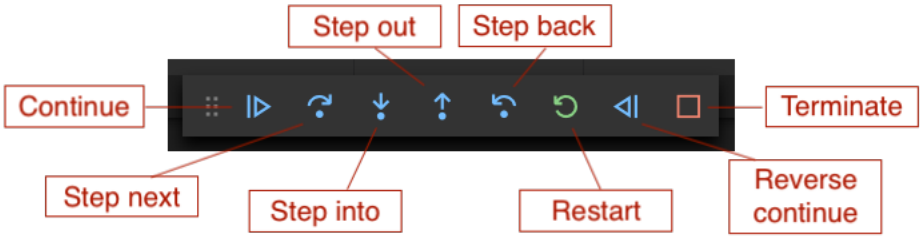


Figure 6.11: Controls in VSCode.

State inspection. As shown in Figure 6.12, 5 categories of variables are shown in the side bar within their own sections: stack, memory, storage, flint, and evm.

These categories are the same as what is shown in flint-debug, an explanation of each category can be found in Table 6.7.

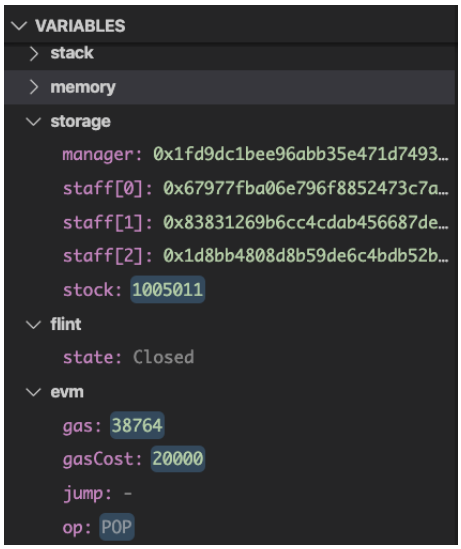


Figure 6.12: State inspection in VSCode.

Expression evaluation. Expression evaluation and watch expressions are not supported.

6.6.2 Remix IDE

Remix allows the user to connect to any Ethereum client (e.g. Ganache), *or* use its internal JavaScript VM. In our experiments, Remix's debugger sometimes gives different results when used with Ganache and its JavaScript VM. These differences will also be briefly discussed.

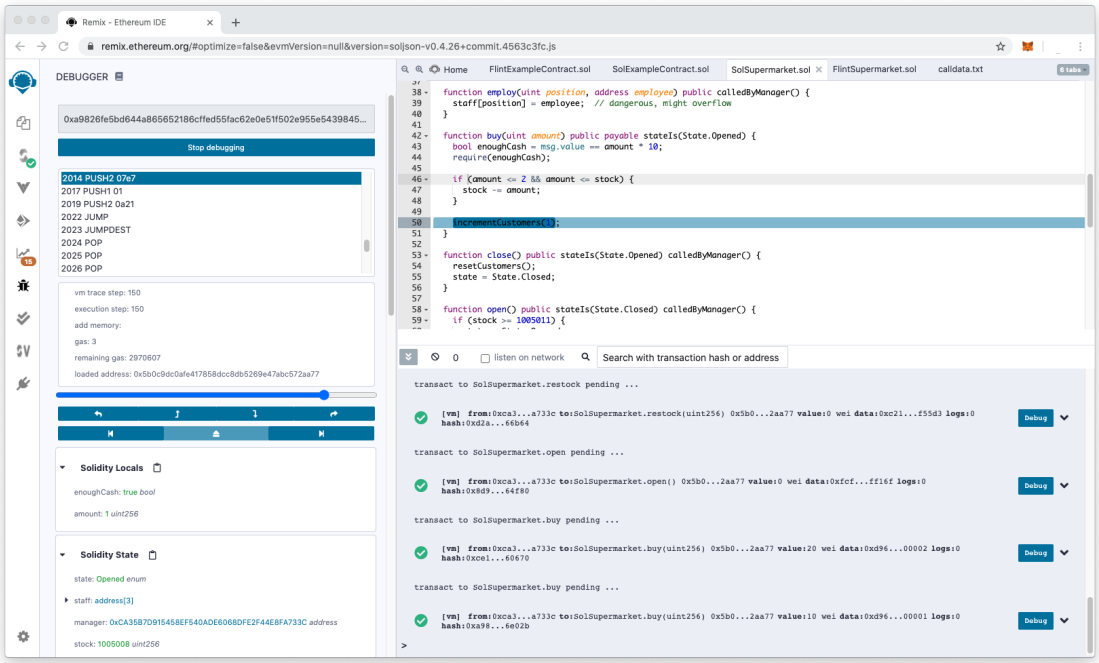


Figure 6.13: Debugging in Remix.

Launching the debugger. To start debugging, the user can enter a transaction hash in Remix's debug sidebar (shown on the left in Figure 6.13).

6.6.2.1 Debugging a Solidity contract

Breakpoints. To set a breakpoint, one can click on a line number in the main editing panel. The user can jump forwards or backwards to the closest breakpoint by clicking the relevant buttons shown in Figure 6.14.

There is however a performance issue when testing against Ganache, instead of the JavaScript VM. There is a noticeable delay (>1s) for it to hit a breakpoint at line 14 of our example contract, but there is no delay with the JavaScript VM.

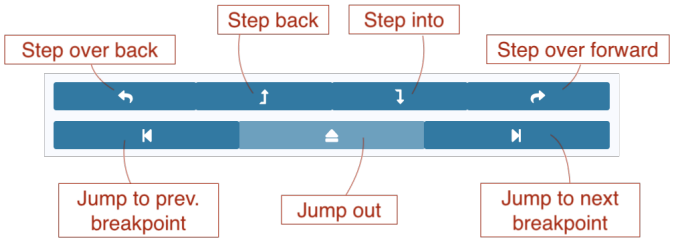


Figure 6.14: Controls in Remix.

Stepping. Stepping commands are available in the panel shown in Figure 6.14. The following descriptions of stepping commands are conclusions from testing our example contract, since it is not documented in Remix what each command does. The behavior is identical with Ganache and the JavaScript VM.

- **Step over back/forward.** Step back/forward one EVM instruction.
- **Step back/into.** Identical to step over back/forward.
- **Jump out.** Unknown. The button is disabled during the debugging session.

Additionally, Remix includes a slider to quickly navigate through the entire transaction trace.

State inspection. As shown in Figure 6.15, contract state variables and local variables are displayed. When testing with Ganache, Remix suffers from the same problem Truffle does. Contract state variables are inaccurate until they are written to. However, when used with Remix's JavaScript VM, the state variables are always accurate, even before they are written to.

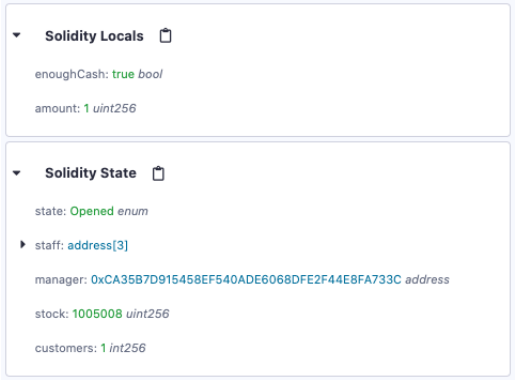
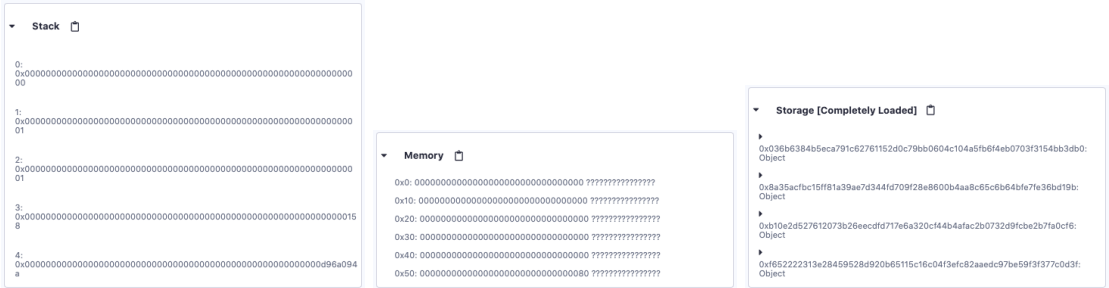


Figure 6.15: Contract variables in Remix.

Raw contents of the contracts' stack, memory and storage are also displayed, similar to what the Flint debug extension displays, as shown in Figure 6.16.



(a) Stack. (b) Memory. (c) Storage.

Figure 6.16: Contents of the stack, memory, and storage in Remix.

Expression evaluation. Expression evaluation and watch expressions are not supported.

6.6.2.2 Debugging a Flint contract

Breakpoints. Breakpoints in YUL code work as expected unlike Truffle, but still suffers from the same performance issue with Ganache.

Stepping. Stepping works as described in the previous section.

State inspection. Remix does not show variables defined in YUL code. Raw contents of the stack, memory, and storage are displayed the same way with Solidity contracts.

Expression evaluation. Expression evaluation and watch expressions are not supported.

6.6.3 Summary

Interface. Integrating the Flint debugger with a generic editor i.e. VS Code allows the user to stay in a familiar code editing environment, with keyboard shortcuts, plugins etc. configured to their liking, while Remix is a highly specialized IDE which developers might have to get used to.

Launching the debugger. Both provides the flexibility to configure launch configuration to use different Ethereum clients.

Breakpoints. Breakpoints generally behave the same in both debuggers.

Stepping. It is hard to compare and make a conclusion about the semantics of stepping commands in both debuggers, as the semantics are not documented in Remix and are unclear from basic testing. Both debuggers support reverse debugging. Remix provides an additional feature to quickly navigate around the transaction trace.

State inspection. Both support viewing the raw contents of the contract's stack and memory. Remix can accurately show state variables that would be unconfirmed in the Flint debugger, only if its internal JavaScript EVM is used.

Expression evaluation. Unsupported by both debuggers.

6.7 Comparison: Conclusion

As explained and shown in previous sections, attempting to debug a Flint contract using existing Solidity debuggers gives a non-ideal experience, rightfully so, as they are designed with Solidity in mind.

The Flint debugger implements features provided by existing Solidity debuggers but within the context of Flint. Currently, it also provides one Flint-specific feature, type state inspection. Table 6.8 summarizes features supported by the Flint debugger (we consider the CLI and GUI as one debugger in this summary, as they are the same feature-wise), compared with those supported by Truffle and Remix.

One feature that is Flint-specific is the ability to inspect the type state of a contract. Take the following code snippet as an example. In a function like `open()`, we know that the contract's type state must be `Closed`, as the function can only be called when the state is `Closed`. In this case, type state inspection is not very useful. However, some functions can be called in any state, such as `payday()` from the example. In this case, it could be helpful to the programmer if they need to know what the contract's type state is when the function is called.

	Flint debugger + Flint	**Truffle + Flint**	**Truffle + Solidity**	**Remix + Flint**	**Remix + Solidity**
Stepping	Yes	Yes (limited)	Yes	Yes	Yes
Reverse debugging	Yes	No	No	Yes	Yes
Breakpoints	Yes	No	Yes	Yes	Yes
Contract variables	Yes	No	Yes (buggy[3])	No	Yes
Raw state inspection	Yes	Yes	Yes	Yes	Yes
Expression evaluation	No	No	Yes	No	No
Type states	Yes	No	N/A	No	N/A

Table 6.8: A summary of features provided by the Flint debugger, Truffle, and Remix.

```
contract FlintSupermarket(Opened, Closed) {
  // ...
}

FlintSupermarket @(Closed) :: (any) {
  public func open() {
    // state must be Closed
    // ...
  }
}

FlintSupermarket @(any) :: (manager) {
  public func payday() mutates (cashRegister) {
    // state could be anything
    // ...
  }
}
```

6.8 Concluding remarks

We have preserved the Flint compiler's functionalities after refactoring and extending its code generation mechanism. However, the performance of source map generation can be an issue, therefore it is not enabled by default.

By comparing our debugger with existing Solidity debuggers, Truffle and Remix, we have identified their differences in features and how they affect the user experience. We also simulated a possible Flint debugging process without a dedicated Flint debugger, using the Solidity debuggers, and showed that our Flint debugger vastly improves the debugging experience from not having a Flint debugger. All in all, both user interfaces of our debugger generally give a similar experience to the experience given by Truffle and Remix, which is what a smart contract developer would expect from a smart contract debugger.

Our debugger is designed to be integrated well with a smart contract developer's existing development environment. For one, it allows them to be flexible with the testing blockchain they use. Also, the architecture of the debugger allows easy integration with more code editors and IDEs in the future, and not just VS Code, allowing even more developers to debug without leaving their development tool of choice. We believe this greatly adds to the Flint debugging experience.

Chapter 7

Conclusion

We present the Flint debugger, which provides two user interfaces: a command line interface (CLI), *flint-debug*; and a graphical user interface (GUI), *Flint Debug VS Code extension*. The debugger improves the developer experience of debugging Flint contracts, helping smart contract developers write safer code with Flint.

As side products of developing the Flint debugger, we extended the open source library Web3.swift to support Ethereum's debug API, and also built a new library for Swift developers to work with the Debug Adapter Protocol (DAP).

7.1 Future work

7.1.1 Debugger

Expression evaluation. The debugger lacks the ability to evaluate expressions on the fly during debugging, which is a function provided by truffle-debug. Flint has its own REPL, which could be integrated into the debugger.

Ewasm. Since EVM will slowly be replaced by Ewasm, it is necessary for Flint and the debugger to support it. Due to various reasons such as (1) Flint using a very old version of Solidity/YUL as its IR, which does not support compilation to Ewasm; (2) Ewasm's integration into Ethereum is still in active development; right now is not the best time to support Ewasm. The entire debugger architecture is however designed to be modular, such that the debug adapter and hence the extension would not have to be changed if the implementation of the core debugger changes.

Real-time debugging. None of the current smart contract debuggers support real-time debugging with a standard Ethereum client like geth. It is worth investigating if it is possible to implement the debugger as a traditional real-time debugger instead of a trace-based debugger, and how it would change the smart contract debugging experience. This would likely require modifying the innerworkings of an existing Ethereum client.

Support more IDEs/editors. Since a debug adapter is implemented as part of the project, it should be easy to add support for IDEs and editors that support the Debug Adapter Protocol.

Improve source map generation performance. As mentioned in section 6.3, source map generation is very slow. It could significantly impact the developer experience if the compiler takes

seconds to compile.

7.1.2 SwiftDAP.

Async I/O. The debug adapter currently reads and writes to its input and output synchronously. Asynchronous I/O would be more appropriate given the asynchronous, event-driven nature of debug adapters. An asynchronous approach to implementing the adapter had been attempted during development. However, due to unknown reasons, messages would mysteriously disappear during transport. For the sake of this project, we fell back to a simpler, but naive synchronous approach.

Implement all protocol messages. Currently, not all protocol messages are implemented. SwiftDAP would not be a fully-fledged library without the full protocol implemented.

7.2 Challenges

Working on a legacy codebase. Just like any legacy codebase, the Flint codebase takes a lot of time and effort to understand. Especially because multiple people have worked on the codebase on a project-by-project basis, without having a consistent maintainer of the codebase. It was challenging to implement new features on top of the existing code. Moreover, the build system is rather complicated due to its many dependencies in multiple languages – it took more than a week to successfully compile the project.

Limited library support for Swift. Due to reasons mentioned in chapter 3, we have decided to implement as much of the project in Swift. However, it is not a popular language for Ethereum development or debugger development and hence there is not much library support for the protocols we need to work with. It was necessary for us to implement our own library and extend a 3rd party library.

Lack of documentation. Although there is a vast amount of resources on implementations of general debuggers, little to no resources are available on debuggers for code run on blockchains. Many concepts coming from general debuggers cannot be applied directly to this scenario. There are only a few publicly available smart contract debuggers, which implementations are not documented extensively, or at all. The only way to understand their implementations is by studying their source code, which is a huge challenge on its own.

7.3 Final remarks

Debuggers are a valuable asset in a programmer's toolbox. This is especially true for smart contract programmers, since an Ethereum blockchain is essentially a black box environment. Our Flint debugger, like other smart contract debuggers, help programmers gain visibility within this black box, helping them solve issues with their code.

Working with a legacy codebase is known to be a difficult task. It could take weeks to fully familiarize oneself with a codebase like this. In a time constrained environment, we cannot afford to completely understand the code before starting to work on it. Sometimes it is necessary to instead spend time to strategically work around the existing structure of the code, as demonstrated with our design of the CodeFragment API section 4.2, so that we can avoid spending extra time fixing broken code.

Bibliography

[1] Schrans F, Hails D, Harkness A, Drossopoulou S, Eisenbach S. Flint for Safer Smart Contracts; 2019. (Accessed on 2020/05/15). Available from: https://arxiv.org/abs/1904.06534.

[2] Chowdhury M, Eisenbach S, Chatley R. flint/MohammadChowdhury.pdf at 81d3266f30ae127836c6711119e170665d80b003 · flintlang/flint;. (Accessed on 2020/01/23). Available from: https://github.com/flintlang/flint/blob/81d3266f30ae127836c6711119e170665d80b003/docs/pdfs%20(student%20theses)/ecosystem/MohammadChowdhury.pdf.

[3] Hunt Andrew. The pragmatic programmer : from journeyman to master. Reading, Mass ; Harlow: Addison-Wesley; 2000.

[4] Gauss EJ. The "Wolf Fence" Algorithm for Debugging. Commun ACM. 1982 Nov;25(11):780. Available from: https://doi.org/10.1145/358690.358695.

[5] Engblom J. A review of reverse debugging. In: Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference. IEEE; 2012. p. 1–6.

[6] Rosenberg Jonathan B. How Debuggers work : algorithms, data structures, and architecture. New York ; Chichester: John Wiley; 1996.

[7] ptrace(2) - Linux manual page;. (Accessed on 2020/01/21). Available from: http://man7.org/linux/man-pages/man2/ptrace.2.html.

[8] Overview;. (Accessed on 2020/01/21). Available from: https://microsoft.github.io/debug-adapter-protocol/overview.

[9] Tools supporting the DAP;. (Accessed on 2020/01/21). Available from: https://microsoft.github.io/debug-adapter-protocol/implementors/tools/.

[10] White Paper · ethereum/wiki Wiki;. (Accessed on 2020/01/23). Available from: https://github.com/ethereum/wiki/wiki/White-Paper.

[11] Go Ethereum;. (Accessed on 2020/01/22). Available from: https://geth.ethereum.org/.

[12] Clients - ethernodes.org - The Ethereum Network & Node Explorer;. (Accessed on 2020/01/22). Available from: https://www.ethernodes.org/.

[13] McCallum T. Diving into Ethereum's Virtual Machine(EVM): the future of Ewasm - By Timothy McCallum;. (Accessed on 2020/01/20). Available from: https://hackernoon.com/diving-into-ethereums-virtual-machine-the-future-of-ewasm-wrk32iy.

[14] JSON RPC · ethereum/wiki Wiki;. (Accessed on 2020/01/20). Available from: https://github.com/ethereum/wiki/wiki/JSON-RPC.

[15] JSON-RPC Working Group. JSON-RPC 2.0 Specification; 2013. (Accessed on 2020/01/21). Available from: https://www.jsonrpc.org/specification.

[16] Remix - Ethereum IDE;. (Accessed on 2020/01/13). Available from: https://remix.ethereum.org/.

[17] debug Namespace | Go Ethereum;. (Accessed on 2020/01/22). Available from: https://geth.ethereum.org/docs/rpc/ns-debug.

[18] Truffle | Overview | Documentation | Truffle Suite;. (Accessed on 2020/01/13). Available from: https://www.trufflesuite.com/docs/truffle/overview.

[19] truffle-npm;. (Accessed on 2020/01/16). Available from: https://www.npmjs.com/package/truffle.

[20] npm | build amazing things;. (Accessed on 2020/01/16). Available from: https://www.npmjs.com/.

[21] dapp.tools;. (Accessed on 2020/01/16). Available from: https://dapp.tools/.

[22] dapp.tools - hevm;. (Accessed on 2020/01/23). Available from: https://dapp.tools/hevm/.

[23] flint/state_of_flint.md at 81d3266f30ae127836c6711119e170665d80b003 · flintlang/flint;. (Accessed on 2020/01/23). Available from: https://github.com/flintlang/flint/blob/81d3266f30ae127836c6711119e170665d80b003/docs/state_of_flint.md.

[24] Blackshear S, Cheng E, Dill DL, Gao V, Maurer B, Nowacki T, et al.. Move: A language with programmable resources; 2019. Available from: https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources.pdf.

[25] Contract Metadata — Solidity 0.4.25 documentation;. (Accessed on 2020/06/04). Available from: https://solidity.readthedocs.io/en/v0.4.25/metadata.html#encoding-of-the-metadata-hash-in-the-bytecode.

[26] Miscellaneous — Solidity 0.4.25 documentation;. (Accessed on 2020/06/15). Available from: https://solidity.readthedocs.io/en/v0.4.25/miscellaneous.html#source-mappings.

[27] time(1) - Linux manual page;. (Accessed on 2020/06/11). Available from: https://man7.org/linux/man-pages/man1/time.1.html.

[28] trufflesuite/truffle-debugger: Core functionality for debugging Solidity files built with Truffle;. (Accessed on 2020/06/11). Available from: https://github.com/trufflesuite/truffle-debugger.

[29] ethereum/remix: Ethereum IDE and tools for the web;. (Accessed on 2020/06/11). Available from: https://github.com/ethereum/remix.

[30] Truffle | Debugging Your Contracts | Documentation | Truffle Suite;. (Accessed on 2020/06/04). Available from: https://www.trufflesuite.com/docs/truffle/getting-started/debugging-your-contracts.

Appendix A

User manuals

User manuals and documentation for each component of the project can be found in their respective repositories/websites.

A.1 Project URLs

A.1.1 Flint (fork).

Repository: https://github.com/noellee/flint

A.1.2 VS Code Extension.

Repository: https://github.com/noellee/vscode-flint-debug

VS Code Marketplace page: https://marketplace.visualstudio.com/items?itemName=noellee-doc.flint-debug

A.1.3 SwiftDAP.

Repository: https://github.com/noellee/SwiftDAP

A.1.4 Web3.swift (fork).

Repository: https://github.com/noellee/Web3.swift

A.1.5 Contracts and scripts used for evaluation.

Repository: https://github.com/noellee/flint-debugger-evaluation

Appendix B

Code examples

B.1 Full contracts for evaluation

B.1.1 Flint

```
contract FlintSupermarket (Opened, Closed) {
  var staff: Address[3] = []
  let manager: Address

  var stock: Int = 0

  var customers: Int = 0

  var cashRegister: Wei
}

FlintSupermarket @(any) :: caller <- (any) {
  public init() mutates (Wei.rawValue) {
    manager = caller
    cashRegister = Wei(0)
    become Closed
  }

  @payable
  public func deposit(implicit value: Wei) {
    cashRegister.transfer(source: &value)
  }

  func incrementCustomers(count: Int) mutates (customers) {
    customers += count
  }

  func resetCustomers() mutates (customers) {
    customers = 0
  }
}

FlintSupermarket @(any) :: (manager) {
  public func payday() mutates (cashRegister) {
    for let employee: Address in staff {
      var salary: Wei = Wei(&cashRegister, 42)
      send(address: employee, value: &salary)
    }
  }
```

```
40
41    public func employ(position: Int, employee: Address) mutates (staff) {
42      staff[position] = employee
43    }
44  }
45
46  FlintSupermarket @(Opened) :: (any) {
47    @payable
48    public func buy(amount: Int, implicit value: Wei) mutates (stock,
        ↪ customers) {
49      let enoughCash: Bool = value.getRawValue() == amount * 10
50      assert(enoughCash)
51      cashRegister.transfer(source: &value)
52
53      if amount <= 2 && amount <= stock {
54        stock -= amount
55      }
56
57      incrementCustomers(count: 1)
58    }
59  }
60
61  FlintSupermarket @(Opened) :: (manager) {
62    public func close() mutates (customers) {
63      resetCustomers()
64      become Closed
65    }
66  }
67
68  FlintSupermarket @(Closed) :: (manager) {
69    public func open() {
70      if stock >= 1005011 {
71        become Opened
72      }
73    }
74  }
75
76  FlintSupermarket @(Closed) :: (manager, staff) {
77    public func restock(amount: Int) mutates (stock) {
78      stock += amount
79    }
80  }
```

Listing B.1: Example Flint contract: Supermarket.flint

B.1.2 Solidity

```
1  pragma solidity ^0.4.25;
2
3  contract SolSupermarket {
4    enum State {Opened, Closed}
5    State state;
6
7    address[3] staff;
8    address manager;
9
10   uint stock = 0;
11
12   int customers = 0;
13
14   constructor () public {
15     manager = msg.sender;
16     state = State.Closed;
```

```
17  }
18
19  function deposit() public payable {
20    // do nothing
21  }
22
23  function incrementCustomers(int count) private {
24    customers += count;
25  }
26
27  function resetCustomers() private {
28    customers = 0;
29  }
30
31  function payday() public calledByManager() {
32    for (uint i = 0; i < staff.length; i++) {
33      uint salary = 42;
34      staff[i].transfer(salary);
35    }
36  }
37
38  function employ(uint position, address employee) public calledByManager()
      ↪ {
39    staff[position] = employee;  // dangerous, might overflow
40  }
41
42  function buy(uint amount) public payable stateIs(State.Opened) {
43    bool enoughCash = msg.value == amount * 10;
44    require(enoughCash);
45
46    if (amount <= 2 && amount <= stock) {
47      stock -= amount;
48    }
49
50    incrementCustomers(1);
51  }
52
53  function close() public stateIs(State.Opened) calledByManager() {
54    resetCustomers();
55    state = State.Closed;
56  }
57
58  function open() public stateIs(State.Closed) calledByManager() {
59    if (stock >= 1005011) {
60      state = State.Opened;
61    }
62  }
63
64  function restock(uint amount) public stateIs(State.Closed)
      ↪ calledByManagerOrStaff() {
65    stock += amount;
66  }
67
68  // modifiers to approximate flint features
69
70  modifier stateIs(State requiredState) {
71    require(state == requiredState, "Not in appropriate state.");
72    _;
73  }
74
75  modifier calledByManager() {
76    require(msg.sender == manager, "Not an authorized caller.");
77    _;
```

```
78    }
79
80    modifier calledByStaff() {
81      require(isStaff(), "Not an authorized caller.");
82      _;
83    }
84
85    modifier calledByManagerOrStaff() {
86      require(msg.sender == manager || isStaff(), "Not an authorized caller."
        ↪ );
87      _;
88    }
89
90    function isStaff() private view returns (bool) {
91      for (uint i = 0; i < staff.length; i++) {
92        if (msg.sender == staff[i]) {
93          return true;
94        }
95      }
96      return false;
97    }
98  }
```

Listing B.2: Example Solidity contract: Supermarket.sol