Ethereum Debug (EDB)

Proposal for Computer Projects 2018
University of Scranton

Andrew Plaza

May 14, 2018

Contents

1 Context

1.1 Who Needs It?

In recent years, blockchain and digital currencies have rocketed in popularity. Many have realized the potential blockchain technology has in revolutionizing society. In general, blockchain can be thought of as a decentralized value-transfer system shared across the world via the internet, while remaining almost free to use.[1] At it's core, blockchain currencies like Bitcoin operate as a cryptographically secure state-machine. Ethereum takes this concept a step further; Ethereum, another blockchain currency, allows any developer the ability to create their own software on top of Ethereum's own core state machine. Therefore, in order to facilitate the developers ability to write software for Ethereum, a set of specialized programming languages were built.

The Ethereum Open Source community, hobbyist and early-adopter developers, however, need an intuitive and familiar way of testing their applications. A debugger with a familiar interface and an emphasis on ease of use, extensibility, and maintainability needs to be developed.

The core of this project concerns the debug library. This library serves as the engine that enables many plugins and client-facing interfaces to be built. In essence, this library will allow the launch of a process that exposes its debug functionality via an inter-process communication (IPC) protocol. Documentation of the IPC protocol allows the development of plugins by third-parties for various development environments and workflows. For demonstration purposes, two simple interfaces will be developed in addition to the library. These are the command-line interface and the Visual Studio Code plugin. Therefore, the 'debugger' references some interface and the core library working together.

1.2 Reasons For A New System?

Currently, debuggers for specialized blockchain programming languages remain either non-existent or extremely limited. The most popular current debugger is an online tool known as an Integrated Developer Experience (IDE) and named 'Remix'.[2] Although providing some popular functions of debugging, Remix is cumbersome and complicated to use. Despite this lack of debuggers, however, applications written in Ethereums programming languages greatly benefit from extensive testing and debugging in order to better ensure security and a seamless user experience.

1.3 Where Will It Be Used?

This new debugger will be used locally on the developer's machine. The debug library, however, may be used in conjunction with any popular IDE plugin framework to build a customized debugging interface for mainstream IDE's.

[1] Gavin Wood, *Ethereum Whitepaper* http://gavwood.com/Paper.pdf
[2] *Remix* https://remix.ethereum.org/

1.4 Software Requirements

In order to develop the debug library, a robust and tested Ethereum Virtual Machine (EVM) implementation is needed.[3] In addition to this, a compiler for these languages is required to be used. During development, workflow will consist of vim as an editor, Rust document generation for documentation of the debug API, and native Rust testing constructs for integration and unit testing.

1.5 Intended Target Demographic

Developers who work for organizations such as Augur or Enjin, hobbyist/early-adopters interested in blockchain programming and applications.

2 Overview

2.1 Problem

Current testing solutions are either non-existent or too complicated and cumbersome in use to be truly a part of a developers workflow. There is currently no integration into mainstream IDE's such as Visual Studio Code, or any debug libraries in existence. As these specialized languages become more and more popular, more advanced debugging techniques will become more important in order to test applications growing in size and complexity.

2.2 Objectives

- Create a debug library supporting at least the Solidity language with a robust and clear API that is well-documented
- Implement basic debug functions (step over, step into, next, info, backtrace, print, quit, kill)
- Implement an read-eval-print loop (REPL) for testing code on-the-fly
- Create a VSCode plugin implementing the debug library
- Create a simple command-line interface implementing debug library
- Document debug library API and make documentation public by publishing/hosting it on the web via Github Pages

2.3 Inputs

The debugger will support the main languages of LLL (Low-level Lisp-like language), Solidity, Serpent, and Vyper. All these languages compile down to the same bytecode processed by the EVM. In addition to bytecode being output

[3] *Parity EVM* https://github.com/paritytech/parity/tree/master/ethcore/evm/src

during compilation, the abstract syntax tree of the program is also provided. LLL is closest in it's level of abstraction, similarity, and direct access to instructions represented by the bytecode, while the other languages are more abstract and unique in their style, and are often compared to Javascript or Python.

Users who input these languages are given the option to execute debug features through their chosen interface (VSCode or CLI). These features correspond to functions which are handled and included by the debug library.

2.4 Outputs

The interface will show pertinent information given by the debug library in a appealing format. The information that will be displayed includes breakpoints, variables, stack traces, execution information and the REPL. The library will manage and output this information via IPC to the interface.

2.5 Features

- Resolution of imports present in program being debugged
- Compilation of code from parent language (LLL/Solidity/Serpent/Vyper) to bytecode. This compilation includes generation of the Abstract Syntax Tree
- Source mapping of bytecode to parent language
- Launching of an emulated VM/REPL environment
- Setting/Enabling/Disabling of breakpoints
- Halting execution of VM at breakpoints
- Providing information about execution (variables, stack)
- Options to step over, step into, continue, print, quit, kill, or restart execution
- On-the-fly code interpretation and testing (REPL)

2.6 Constraints

The debug library will be written in the Rust programming language.[4] Fortunately, support for Rust among the Ethereum Open Source community is strong, particularly with the popularity of the Parity Ethereum application.[5] However, compilers for all but one of the programming languages are written with C. Rust bindings for the Solidity compiler already exist, but bindings for LLL, Serpent, and Vyper will have to be generated.[6]

[4] *Rust* https://www.rust-lang.org/en-US/
[5] *Parity* https://www.parity.io/
[6] *Rust-Bindgen* https://github.com/rust-lang-nursery/rust-bindgen

Since the debugger will act with a local version of the EVM, performance impact of working with a test or live blockchain does not exist. In consideration of program runtime during debugging, however, performance may be impacted depending on the number of trace calls on function definitions being debugged, or number of checks for breakpoints on code instructions. In addition, the method of IPC communication chosen may impact the speed of communication and responsiveness with IDE plugin frameworks.

3 Feasibility

Crucial work on this project has already been underway, with some parts of the project completed.[7] These completed parts of the debugger, however, were created using NodeJS instead of Rust. Particularly, source mapping, code execution control, EVM hooks/interaction with the EVM, and a VSCode debugging plugin written in Typescript have already been completed.[8]

The choice to use Rust for this project came out of concerns for performance, maintainability, documentation, and API usability. NodeJS proved to be a poor choice in all three of these aspects, while Rust excels at these three areas. Rust provides several important advantages in terms of including intuitive IPC features, data structures, library management, package management, along with testing and documentation tooling. I am also fairly confident in my experience with Rust, having worked with a number of previous projects written in the language, including an Open Source Operating system and Linux Window Manager. Therefore, much of the first phase of the project will simply include porting the existing and working NodeJS debug library logic to Rust.

Once this is complete, work on implementing the missing features from the first version of the debug library will begin. This includes implementing functions which take further advantage of the execution control already present in the first version of the debugger. These are some of the basic functions required for debugging, such as 'next', 'continue', 'kill', and 'step into'. In addition to this, the inspection and outputting of variables, and a REPL written in Rust must be completed.

Once these features are complete, a simple command line interface can be created, and the VSCode plugin will have to be modified to work with the new IPC interface.

Considering the time-frame of this project, and my previous experience writing applications in Rust, it seems very likely the project will be completed in the time given.

[7]Andrew Plaza, Sean Batzel *ethdbg* https://github.com/ethdbg/ethdbg
[8]Andrew Plaza, Sean Batzel *VSCode Plugin* https://github.com/ethdbg/vscode-ethdbg