Ethereum Debug (EDB)

Proposal for Computer Projects 2018

University of Scranton

Andrew Plaza

May 13, 2018

Contents

1 Context

1.1 Who Needs It?

The Ethereum Community Open Source community, hobbyist and early-adopter developers need an intuitive and familiar way of testing their applications written in niche languages. A familiar interface and underlying software with an emphasis on ease of use, extensibility, and maintainability needs to be developed.

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 interactive IDE [1]. Although providing some popular functions of debugging, it is cumbersome and complicated to use. Despite this lack of , however, applications written in these 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 and debug library will be used locally on the developers machine in command-line interface form similar to GDB, or in the form of a Visual Studio Code plugin. As a debug library, it may be used in conjunction with any popular IDE plugin framework to build a customized debugging plugin for mainstream IDE's.

1.4 Software Requirements

In order to develop this plugin a robust and tested EVM (Ethereum Virtual Machine) implementation is needed [2]. In addition to this, a compiler for these languages is required to be used. During development, work-flow will consist of the use 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.

[1] Remix Online IDE: https://remix.ethereum.org/

[2] Parity EVM will be used: https://github.com/paritytech/parity/tree/master/ethcore/evm/src

2 Overview

2.1 Problem

Current testing solutions are either non-existent or too complicated and cumbersome in use to be truly apart of a developers work-flow. There is currently no integration into mainstream IDE's such as Visual Studio Code or IntelliJ, nor any local 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 (breakpoint information), backtrace, print, quit, kill)
- Implement an interactive REPL for testing/interpreting 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 it public by publishing/hosting it on the web via Github Pages

2.3 Inputs

The main languages to be supported by the debugger are 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 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

Breakpoints requested by the user will be highlighted and/or indicated in some way by the interface application (VSCode or CLI). In addition, the interface will show pertinent information given by the debug library in a appealing format. The library will manage and output variables, stack traces, current line numbers/execution location and REPL 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 [3]. Fortunately, support for Rust among the Ethereum Open Source community is strong, particularly with the popularity of the Parity Ethereum application [4]. 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 [5].

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

3 Feasibility

Crucial work on this project has already been underway, with some parts of the project completed [6]. 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 [7] written in Typescript have already been completed.

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 debugger logic to Rust.

Once this is complete, work on implementing the missing features from the first version of the debugger 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 CLI debugger can be created, and the VSCode plugin will have to be modified to work with the new IPC interface, with the missing features added.

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.

[6] EDBv1: https://github.com/ethdbg/ethdbg
[7] VSCode Plugin: https://github.com/ethdbg/vscode-ethdbg