Echidna: A Practical Smart Contract Fuzzer

John-Paul Smith [1] Alex Groce[1] Gustavo Grieco[1] Josselin Feist[1]

Trail of Bits, New York NY

Objectives

Designing a smart contract fuzzer poses numerous challenges. Traditional fuzzing targets receive input via a buffer of bits, explore a complex control flow graph, and detect notable inputs by observing program crashes. Smart contracts in contrast have a structured, transactional input, almost no control flow, and largely cannot "crash" in conventional ways. Nonetheless, fuzz testing is highly effective, especially in cases where symbolic execution or formal verification are not tractable. We wish to design and implement a fuzzer for Ethereum smart contracts, then evaluate its performance on deployed contracts.

Smart Contracts

Ethereum smart contracts are small programs designed to be executed in consensus by a network of mutually distrusting nodes. They run on the *Ethereum Virtual Machine* (EVM), a small, stack-based ISA inspired by Bitcoin Script. Most smart contracts are written in the Solidity language. Solidity is a statically-typed, compiled language inspired by Javascript.

Typically, contracts present an API consisting of a number of functions that can be called by humans or other contracts. Each function in this API typically has an associated function type, and there is a standard Solidity ABI for encoding values of each type. Contracts cannot execute code unless they are called, so the API describes the entire functionality of the contract.

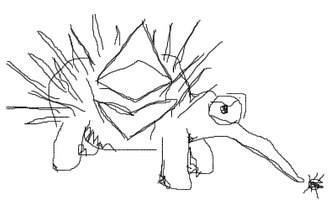

Figure 1:Echidna's logo (it's eating bugs)

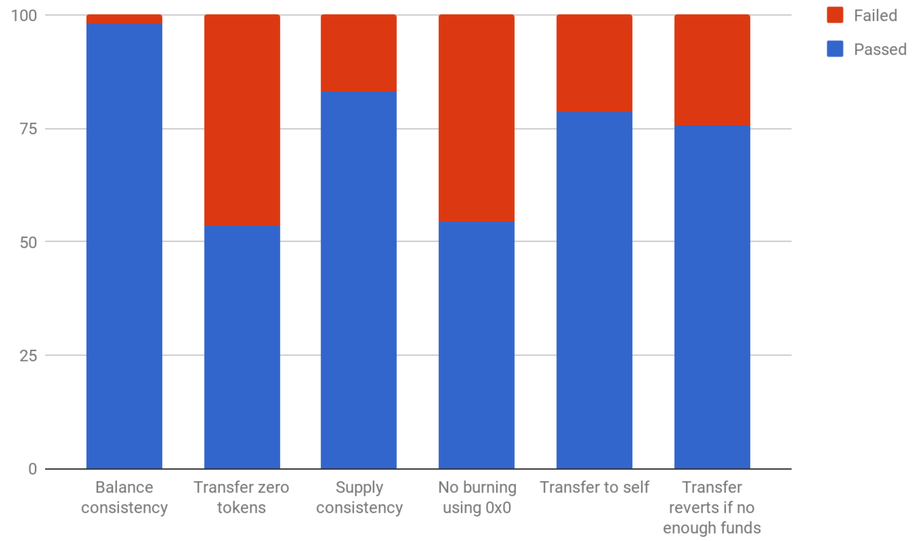

Figure 2:Bugs found in deployed ERC20 contracts

Echidna's Design

Echidna "fuzzes" smart contracts via the addition of extra "test" functions to the API. These functions must take no arguments and return a Boolean indicating whether the contract is functioning as intended. For instance, when testing a token contract, one might test whether all balances sum to the intended total supply. It then generates a random sequence of calls to the contract's API, and, after each call, checks the return of each test. This allows contracts to fail observably, much like a conventional program might segfault.

Typically, to test a smart contract, one inherits from it, modifies the constructor such that the contract initializes in a realistic state, then adds a number of the above tests. Echidna also supports fuzzing for assertion violations, so an auditor might also modify existing functions to include extra assertions. Additionally, more API endpoints can be added to simulate the effects of the "real world" on the contract.

Empirical Evaluation

To test Echidna's effectiveness, we scraped the blockchain for popular contracts implementing the ERC20 interface for fungible tokens. This interface is a set of functions that must be included in a compliant contract's API, along with standards for how each function behaves. Because these functions have standard names and types, we can write a single set of tests for all ERC20 contracts. Then for each contract scraped, we inherit from it, add the new tests, and quickly run Echidna against these tests.

We discovered hundreds of vulnerabilities in deployed token code. These ranged from minor cases of specification noncompliance to the ability for any user to counterfeit tokens. The above chart shows the incidence of several bug classes in the sample of 1000 contracts we used. Trail of Bits has also used these tests in commercial auditing work, where they have found more bugs.

Conclusion

The unique design constraints of smart contracts provide a runtime uniquely amenable to fuzzing. While significant work is required to actually generate realistic calls against a given API and emulate a blockchain environment with sufficient fidelity to run our tests, once a fuzzer is operational, it is very easy to add new tests or test new contracts. This allows for easy scaling to test all deployed contracts, and has found numerous significant issues.

Further work is needed to refine the fuzzing strategy employed and extend Echidna to support ever more complex smart contracts. Echidna only recently gained support for testing multiple-contract systems, and understanding the complexities of "upgradeable" contracts is still a work in progress. Most Echidna users also only write Solidity tests, and a more powerful programmatic API might enable powerful new analyses.

Getting Echidna

Echidna is available on the Trail of Bits Github at `github.com/crytic/echidna`. Docker images are also available for easier setup via Docker Hub, and can be found at `hub.docker.com/r/trailofbits/echidna`.

Echidna is open-source software, and we welcome external contributions, bug reports, and feature requests. See the Github repository for instructions on using and contributing to Echidna.

Acknowledgements

The authors of this work would like to thank Ben Perez, Will Song, Dan Guido, Artur Cygan, Dan Bolognino, and all other open-source contributors to Echidna.