2017 IEEE/ACM 39th IEEE International Conference on Software Engineering Companion

Obsidian: A Safer Blockchain Programming Language

Michael Coblenz
Computer Science Department, Carnegie Mellon University
Pittsburgh, PA USA
mcoblenz@cs.cmu.edu

***Abstract*—Blockchain platforms, such as Ethereum, promise to facilitate transactions on a decentralized computing platform among parties that have not established trust. Recognition of the unique challenges of blockchain programming has inspired developers to create domain-specific languages, such as Solidity, for programming blockchain systems. Unfortunately, bugs in Solidity programs have recently been exploited to steal money. We propose a new programming language, Obsidian, to make it easier for programmers to write correct programs.**

***Keywords*-blockchain programming, blockchain security, programming language usability**

I. Introduction

Ethereum [1] and HyperLedger [4] are blockchain-based programming platforms that share the goal of enabling parties that have not established trust to conduct transactions in a distributed computing environment. Blockchain platforms maintain consistent distributed global state and enable participants to run programs that update the shared state. Unlike some other distributed computing platforms, blockchains assure correct state even when some of the servers execute the code maliciously. For example, a distributed autonomous organization could sell shares to parties in exchange for virtual currency. The shareholders may then vote on proposals to pursue with the organization's resources. The organization exists as a *contract*, which is a program that can maintain its own state and conduct *transactions* that transform that state according to messages sent by participants. Blockchains allow shareholders to host the organization in a distributed fashion and still trust the platform to execute the agreed-upon contract faithfully. However, if there is a security vulnerability in the contract, then it may be exploited. Such a vulnerability recently resulted in the loss of over $40M [3].

Luu et al. identified common sources of security bugs in Ethereum contracts and designed an analysis that identified 8,833 of 19,366 existing contracts as being vulnerable to one of these bugs [5]. Atzei et al. presented a taxonomy of vulnerabilities that are common among Ethereum programs [6]. We aim to make blockchain programs less bug-prone by designing a new programming language that encourages writing programs that avoid classes of those known vulnerabilities. Though some safety properties will be guaranteed by the compiler, we also focus on *usability*: can real programmers write correct code in our language, and is their code less likely to be buggy than the code they would have written in Solidity?

Several characteristics of blockchain programs motivate a new language design. First, correctness is critical: many of the proposed applications of blockchains involve financial transactions, so any bugs may result in lost or stolen money or virtual currency. Second, bugs in programs cannot be fixed because the programs are immutable. Once money is committed to an agreement (as implemented in a contract), that money can only be removed according to the existing contract implementation. Third, blockchain programs are commonly *state-based* [7]: finite-state machines provide a simple abstraction for high-level aspects of program behavior. We will exploit this in a novel language design, based on evidence that state-based reasoning facilitates faster, less buggy development [8].

II. Bugs and solutions

We focus here on two serious sources of bugs that we will mitigate or prevent with our language design.

1) **Re-entrancy attacks** occur when a function f calls a function g in an external contract, which makes a reentrant call to f. If f's call to g occurred while the contract was in an inconsistent state, then the reentrant call may make invalid assumptions about the starting state of the contract. A re-entrancy vulnerability was recently exploited to steal over $40M [3]. Avoiding this problem in Solidity is difficult and error-prone, since one must reason about reentrant behavior anytime a function makes an external call. In particular, in Solidity, sending money always results in an external call, so this is an especially frequent vulnerability.
   In general, external calls from function f in contract C may invoke additional calls on C to any function, not just f, via an intermediate external invocation. This is more problematic than internal-only calls because the external contract is likely to assume C is in a consistent state. By providing *named states* we will enable consistency checking on state transitions and verification that external calls only occur in safe states.
2) **Monetary** is owned by contracts, but contracts typically need to store finer-grained information, such as which money is owed to which client. Any bugs in this code could result in loss or misappropriation of resources. Obsidian will use a *linear type system* for quantities of money so that the compiler can guarantee lossless tracking of financial information. The type system can

978-1-5386-1589-8/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE-C.2017.150

ensure that variables with money type actually store real money and that money is conserved. Delmollino et al. showed that it is common for blockchain contracts written by beginners to accidentally leak money [9]. We will also use a dataflow analysis to show when it is possible for money to become trapped inside a contract.

III. Our Approach

Obsidian (now in a prototyping phase) is an object-oriented language that makes state first-class [10]. The methods that can be invoked on an object depend on the object's current state. As an example, consider a naïve, abbreviated Solidity implementation of a Rock, Paper, Scissors application, shown in Figure 1 (inspired by Delmollino et al. [9]). The creator of the contract offers to place bets with each of `capacity` players. After placing bets, players make their choices. Eventually, someone calls `finalize()`, which distributes payouts.

```
1  contract RockPaperScissors {
2    bool payoutHappened;
3
4    function bet() payable {
5      // Store the money if it is the right amount, otherwise reject.
6    }
7
8    function finalize() {
9      // Can only pay out once.
10     if (payoutHappened) {
11       throw; // abort the transaction
12     }
13
14     // Make sure all players have made their choices.
15
16     for (i = 0; i < otherPlayers.length; i++) {
17       address winningAddress = ...
18       // Bug 1: send executes external code, which may cause a re-entrant call.
19       // Bug 2: failed to check return value of send().
20       winningAddress.send(requiredBet * 2));
21     }
22     payoutHappened = true;
23   }
```

Fig. 1. Naïve Rock-Paper-Scissors in Solidity, showing bugs

In line 20, the winner is sent the reward. However, when the send() function is invoked, the winner may execute arbitrary code. This code may call back into finalize(), which does not detect the reentrant call, and attempts to pay the winners again. This results in some winners being paid too much. Eventually, some send() calls will fail when the contract runs out of money to pay them. The code also neglects to check the return value of send(). If it fails, then a winner will never receive a payout and their money will be trapped in the contract forever.

Figure 2 shows how the same application might look in Obsidian. By lifting high-level contract state into first-class states, we encourage programmers to write programs safely. This version is immune to re-entrant calls because the external code is invoked in a safe state: the payouts only occur on entering the state, not on calling a named function, and no unsafe functions are available in that state. By restricting available behavior according to the state and requiring that external calls be made from states in which inconsistent state is inaccessible, we prevent these reentrancy vulnerabilities.

In Solidity, accounting for money occurs through potentially inconsistent mechanisms. Each contract holds virtual currency. In addition, contracts may need to track finer-grained information. For example, a bank needs to track the balance of each account. Though inter-contract transfers are verified by the platform, bugs may result in the bank losing track of money. By treating money ("ether") as a *linear* resource, we can statically detect a class of errors involving money and other linear resources. For example, the implementation of outcomeFunction will be required to store the money it is passed somewhere. If an `if` statement sends the money somewhere but the `else` case does not, the compiler will report an error. By sending money in parameters of methods, rather than in a generic "send" method, the contract can appropriately handle received money.

```
1  contract RockPaperScissors {
2    type OutcomeFunction = ether -> unit;
3
4    state AcceptingBets {
5      transition bet(ether e, OutcomeFunction of) {
6        ... // store outcome function for later use
7        if (otherPlayers.length == capacity) {
8          ->MakingChoices(); // transition to MakingChoices state
9        }
10   }
11
12   state MakingChoices {
13     ... // after all players make their choices, distribute payouts
14   -> ChoicesCompleted();
15   }
16
17   state ChoicesCompleted {
18     ChoicesCompleted() { // constructor
19       for (i = 0; i < otherPlayers.length; i++) {
20         Player otherPlayer = otherPlayers[i];
21         if (computeWinner(i) == Winner.Creator) {
22           creator.outcomeFunction(requiredBet * 2);
23         }
24         else {
25           otherPlayers[i].outcomeFunction(requiredBet * 2);
26         }
27       }
28     }
29   }
```

Fig. 2. Rock-Paper-Scissors in Obsidian

IV. Evaluation

We will prove appropriate safety properties, such as conservation of money. However, we will also conduct user studies to evaluate whether (a) programmers can use Obsidian effectively to write programs with little training; (b) programmers are more likely to write correct, safe code with Obsidian than they are with Solidity. In our user studies, we will ask users to complete programming tasks, assigning some users to Obsidian and some to Solidity, and observe completion rates, completion times, and types and rates of bugs created. This approach leverages our past experience evaluating and improving usability of programming languages [11].

V. Conclusion

Obsidian is a promising direction in the design of languages for blockchain platforms. We expect to show formally that it guarantees the absence of some common bugs and show by user studies that programmers are more likely to write correct programs with Obsidian than with competing approaches.

Acknowledgment

The author thanks the reviewers as well as Jonathan Aldrich, Joshua Sunshine, Brad Myers, and Tyler Etzel. This material is based upon work supported by the NSF under Grant No. NSF CNS-1423054 and by NSA lablet contract H98230-14-C-0140.

REFERENCES

[1] Ethereum Foundation, "Ethereum project," http://www.ethereum.org. Accessed Jan. 3, 2017.

[2] ——, "Solidity," https://solidity.readthedocs.io/en/develop/. Accessed Jan. 3, 2017.

[3] E. Gün Sirer, "Thoughts on the DAO hack," 2016. [Online]. Available: http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/

[4] The Linux Foundation, "Hyperledger," https://www.hyperledger.org. Accessed Jan. 3, 2017.

[5] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, 2016.

[6] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts," Cryptology ePrint Archive: Report 2016/1007, https://eprint. iacr. org/2016/1007, Tech. Rep., 2016.

[7] Ethereum Foundation, "Common patterns," http://solidity.readthedocs.io/en/develop/common-patterns.html. Accessed Jan. 4, 2017.

[8] J. Sunshine, J. D. Herbsleb, and J. Aldrich, "Structuring documentation to support state search: A laboratory experiment about protocol programming," in *European Conference on Object-Oriented Programming (ECOOP)*, 2014.

[9] K. Delmolino, M. Arnett, A. E. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab." *IACR Cryptology ePrint Archive*, vol. 2015, p. 460, 2015.

[10] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and É. Tanter, "First-class state change in Plaid," in *ACM SIGPLAN Notices*, vol. 46, no. 10. ACM, 2011, pp. 713–732.

[11] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine, "Glacier: Transitive class immutability for Java," in *Proceedings of the 39th International Conference on Software Engineering - ICSE '17*, 2017.