A User Study of Obsidian

Anonymous Author(s)

Abstract

Blockchains are useful for this. There are blockchain languages that have security problems. We wanna design a new language in a human-centered way. We did that with a natural programming approach. In this paper, we discuss our approach to the design of user studies, and our preliminary findings

We are currently developing a blockchain-based programming language called Obsidian. Our language offers first-class typestate and linear resources in order to minimize the risk of common security vulnerabilities. It is important that users are able to understand the salient features of the language and write correct, safe code. The purpose of this user study is to explore how people naturally go about expressing and implementing domain-specific problems, and how effectively users are able to understand states and linear resources. The findings of this study will inform our design choices with the goal of increasing the usability of the language.

Keywords keyword1, keyword2, keyword3

ACM Reference Format:
Anonymous Author(s). 2017. A User Study of Obsidian. In *Proceedings of ACM SIGPLAN Conference on Programming Languages, New York, NY, USA, January 01–03, 2017 (PL'17),* 4 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Blockchain platforms such as Ethereum and Hyperledger are decentralized databases that facilitate transactions between parties that have not established trust. Increased interest in these platforms has motivated the design of blockchain-based programming languages like Solidity, which allow users to create blockchain programs. However, there have been several recent instances where Solidity programs have contained bugs that have been exploited. In a particularly calamatious example, $50 million was stolen from a contract on the Etherium blockchain called the DAO [2]. As this example illustrates, the security of blockchain programs is imperative since they often involve the exchange of large amounts of money.

In light of this, we are designing a blockchain-based program called Obsidian [1] with the goal of minimizing the risk of common security vulnerabilities in blockchain programs. Obsidian contains several core features that will allow users to write programs safely and effectively. The target userbase of Obsidian is business professionals who will use the language to write smart contracts, and its syntax is thus oriented towards this domain: Obsidian programs consist of contracts containing fields, states, and transactions.

1.1 Language Features

Typestate

The DAO hack was accomplished using a reentrant function call. States have the ability to prohibit malicious reentrant calls by ensuring that external code is only invoked in a safe state.

Obsidian makes state first class: an object in Obsidian has a mutable state that restricts which transactions can be invoked on it. The states of a contract are defined as explicit blocks containing transactions and fields, which can only be accessed if the contract is in the corresponding state.

```
contract LibraryCard {
   state NoCard {
       transaction getCard() {
           ...
           ->HasCard
       }
   }

   state HasCard {
       transaction checkOutBook() {
           ...
       }
   }
}
```

Fig. 1. An example of states in Obsidian.

In the above example, a `LibraryCard` is always in one of two states, `NoCard` or `HasCard`. The transaction `checkOutBook` can only be called on an instance of `LibraryCard` if it is in the `HasCard` state; otherwise, it will throw an exception. The -> operator indicates a state transition.

Linear Resources

Blockchain programs may manage some kind of resource, like a cryptocurrency, or a token indicating a more complicated right (e.g. ownership of a financial option). The static semantics for linear resources enforce a safe, clean programming model: resources cannot be used more than once, but must be used before leaving the current scope (thus ensuring that a resource is not lost accidentally).

```
contract Treasury {
 // Money is a resource of Treasury
   contract Money { ... }

transaction t1(Money m) {
       spendMoney(m);
       spendMoney(m);
       // compiler error: m is used twice
       Bond b = exchangeForBonds(m);
```

A note.
PL'17, January 01–03, 2017, New York, NY, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
       ...
   }

   transaction t2(Money m) {
     // compiler error: m is never used
     return;
   }
}
```

Fig. 2. Linear resources in Obsidian

Obsidian also supports path-dependent types, wherein linear values can have type members. In the above example, the type of Money is dependent upon a specific value of Treasury; put another way, each Treasury t has its own type t.Money.

Path-dependent type assist in writing secure contracts: without such types, every instance of Treasury would share the same Money type, and the programmer would have to manually check that all money deposited into a treasury - for example - is of the correct type. Incorrect or insufficient checks could leave contracts vulnerable to exploitation. With path-dependent types, all such checks are automatic: we can ensure that only money that comes from an instance of a treasury is used in transactions within that treasury.

1.2 Usability

It is also crucial that our language is usable - that is, real programmers are able to write correct code easily and efficiently. However, even with seemingly intuitive features, it is not always clear which approach to implementation is most user-friendly. For instance, consider the following simple Obsidian contract:

```
contract C {
   state Start {
       int x;

       transaction a() returns int {
           ->S1{x1 = x};
           return b(x1);
       }
   }

   state S1 {
       int x1;

       transaction b(int y) returns int {
           return y;
       }
   }
}
```

In transaction a, we transition to state S1 and then call transaction b. However, does it make sense to call b(x1) after the transition? While we have just transitioned to S1, lexically we are still in the Start state, and it is not immediately apparent which variables and transactions are in scope. Obviously this example is extremely simple, but in a program with many different contracts, transactions, and states, a situation like this could cause serious confusion. Failure to encode state machines properly has been shown to be a significant source of errors in smart-contract programming [3]. It is critical that users are able to understand and use states easily and effectively - if not, there is potential to create the same security weaknesses as you could find in a contract without states.

Similar questions of usability arise with path dependent types. An initial approach we have taken is to use nested contracts to indicate a dependency relationship; however, nesting has different implications in different languages, and we are not certain that users will be able to recognize the presence and utility of path dependency.

A related issue is whether users will want to use these features if they are made available. Are these logical, sensible solutions to real problems that programmers face? If people do not understand or choose to use the language's core features, then any of their potential benefits are lost. We are conducting a user study to investigate these questions of usability in Obsidian [4]. The key research questions we address in the study are as follows:

- Are states a natural way of solving a domain-specific problem?
- How do people naturally express states and state transitions?
- Are our participants able to use and understand states and state transitions?
- Are path dependent types a natural way of solving a domain-specific problem?
- Are people able to effectively use and understand path dependent types as they are implemented in Obsidian?

2 User Study Design

The purpose of this study is to gain information about the usability of state transitions and path dependent types. We chose these two features as the focus of our study because they are both important safety features that we expect users will use frequently. In addition, they both have several distinct options for how to be represented syntactically, and the results of the study will factor into which option we choose to implement.

Participants are asked to complete two similarly structured programming exercises, one pertaining to state transitions and one pertaining to path dependent types. They are instructed to think aloud throughout the exercise, and are permitted to ask questions.

2.1 Voter Registration Exercise

Participants are offered a description of a voter registration system for a hypothetical democratic nation. The system has certain stipulations that make a state machine a logical means of implementation. For instance, the system has specific conditions under which a citizen either becomes registered to vote, or remains unregistered.

The exercise is divided into five parts. In part one, participants are asked to implement the system using pseudocode. They are encouraged to invent any language features that they may want in order to solve this problem. We want to see how people naturally go about solving a problem in this domain: what ideas do they have, and what assumptions do they make?

In part two, participants are offered a state diagram that models the voter registration system, and are asked to modify their pseudocode to include the states and state transitions shown in the diagram. Again, we want to observe people's natural ideas about how to represent states and state transitions in a program.

In part three, participants are offered a two-page Obsidian tutorial detailing the key components of the language. The tutorial explains how state blocks work, but does not give any information about how transitions should be written. Participants are then given an Obsidian program that implements the voter registration system, but is missing state transitions. Participants are asked to add state transitions to the code, inventing the syntax themselves.

In part four, participants are offered three unique options for the syntax and functionality of state transitions, each accompanied by a short code example. Participants are presented the options in a random order; the order given here is arbitrary.

In option 1, users are allowed to use any transaction in any state, given they have already transitioned to that state. For instance, in fig. 3, it is legal to use the `toS2` transaction inside the `Start` state, even though that transaction is defined with `S1`. This is because there is a transition to `S1` in the previous line.

```
contract C {
    state Start {
        transaction t(int x) {
            ->S1{x1 = x};
            toS2();
        }
    }

    state S1 {
        int x1;

        transaction toS2() {
            ->S2{x2 = x1};
        }
    }

    state S2 {
        int x2;
    }
}
```

Fig. 3. One option for state transitions

In option 2, each state has a constructor that is invoked when the user transitions to that state. With this option, there cannot be any code following a state transition, so a transition must thus be the final line of a transaction. In option 3, there are conditional `if in {state}` blocks which allow the user to use the transactions and fields of that state while lexically in a different state.

Participants are asked to complete a short Obsidian contract once for each option. The contract is designed to be a simple yet non-trivial use of state transitions that illustrates the benefits and drawbacks of each option. The goal of this part is to assess participants' reactions and see whether the are able to implement the contract successfully with each option.

In part five, participants are asked to modify the Obsidian program from part three to use one of the three given options for state transitions. They are then asked to explain their reasoning and elaborate on if there was anything confusing about any of the options.

2.2 Lottery Ticket Exercise

Participants are offered a description of a program that allows users to create and participate in lotteries. Every lottery sells lottery tickets, but a user should only be able to redeem a winning ticket from the lottery where they bought it - thus motivating the use of path dependent types.

The exercise is divided into three parts. Part one mirrors the voter registration exercises in that participants are asked to implement this program using pseudocode. Again, we want to see how people naturally go about solving this problem. Will using path dependent types - or some feature similar to that - occur to anyone?

In part two, participants are given an explanation of path dependent types and offered an Obsidian contract that implements the lottery program, but has two transactions left unwritten. Participants are asked to write those transactions. We want to observe whether people are able to understand path dependent types and write correct code using them after only a brief introduction.

3 Discussion of Design

A crucial element of the user study was ensuring that the programming exercises had an appropriate level of difficulty. The scenarios had to be simple enough that participants could comprehend and implement them in the little time they had; however, they also had to be complex enough that implementing them was a non-trivial problem that actually motivated the use of Obsidian?s features. Our first several pilot studies revealed that the initial design of our exercises was too complicated, and participants took much longer to just read and understand the instructions than we had anticipated. Additionally, there were parts of the exercises that people were continually confused about, which made it difficult to assess their ability to use the language. We could not tell if people were confused about aspects of Obsidian, or about the instructions.

As we revised the programming exercises, we trended towards simplifying and condensing. For instance, the state diagram in the voter registration exercise originally had six states and five transitions, but was modified to have three states and three transitions; the tutorial was cut from three pages to one and a half; and two parts were removed from the lottery ticket exercise. We also made sure that each exercise targetted exactly one feature: the voter registration exercise is focused only on state transitions, the lottery ticket exercise on path dependent types.

We found that simplifying the exercises allowed us to collect better data. Participants who completed the simplified exercises spoke their thoughts aloud more consistently and stated their preferences and opinions more confidently. Still, simplifying our programming exercises also created certain limitations. Since the exercises were short and not very complex, participants? opinions may have been based only on a cursory understanding of the language. There may be an option for state transitions, for instance, whose utility only becomes apparent in a big, complicated contract. We are currently missing out on that insight.

4 Preliminary Findings

…

5 Future Work

In the future, we will conduct more user studies with our current exercises and introduce new programming exercises.

- We will target people in the business domain (e.g. business students and business analysts) with limited programming experience, in order to collect data from the intended user base of Obsidian.
- We will design programming exercises that further address the usability of linear resources. One question of interest, for example, is how to enforce linearity for field accesses and state transitions: what should happen when you attempt to access an owned field that has already been read once, and
- We will design programming exercises that require participants to read and write longer, more complicated Obsidian contracts that actually compile. This will offer us more evidence about whether people are able to write correct Obsidian code. It will also allow participants to gain a deeper, less superficial understand of Obsidian's features and thus offer more constructive feedback about Obsidian's usability.
- We will conduct a user study to evaluate whether people are more likely to write correct, safe code with Obsidian than with Solidity.

6 Conclusion

…

References

[1] M. Coblenz, "Obsidian: A Safer Blockchain Programming Language," in *Proceedings of the 39th International Conference on Software Engineering - ICSE '17,* 2017.

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

[3] 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.

[4] B. Myers, A. Ko, T. LaToza, and Y. Yoon, "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools," IEEE Computer, Special issue on UI Design, 49, issue 7, July, 2016, pp. 44-52.