A User Study to Inform the Design of the Obsidian Blockchain DSL

Celeste Barnaby
Wesleyan University
cbarnaby@wesleyan.edu

Michael Coblenz
Carnegie Mellon University
mcoblenz@cs.cmu.edu

Tyler Etzel
Cornell University
tje44@cornell.edu

Eliezer Kanal
Carnegie Mellon University
ekanal@cert.org

Joshua Sunshine
Carnegie Mellon University
sunshine@cs.cmu.edu

Brad Myers
Carnegie Mellon University
bam@cs.cmu.edu

Jonathan Aldrich
Carnegie Mellon University
jonathan.aldrich@cs.cmu.edu

Abstract

Blockchain platforms such as Ethereum and Hyperledger facilitate transactions between parties that have not established trust. Increased interest in these platforms has motivated the design of programming languages such as 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. The security of blockchain programs is especially important given that they commonly involve the exchange of money or other objects with real-world value. We are currently developing a blockchain-based programming language called Obsidian with the goal of minimizing the risk of common security vulnerabilities. We are designing this language in a human-centered way, conducting exploratory user studies with a natural programming approach to inform our design choices. In this paper, we discuss our approach to the design of a user study, as well as our preliminary findings.

Keywords blockchain programming, blockchain security, programming language usability, user studies of programmers

1 Introduction

We are designing a blockchain-based programming language called Obsidian [1] with the goal of minimizing the risk of common security vulnerabilities in blockchain programs. Blockchain programs written with current domain-specific languages such as Solidity [2] often contain exploitable bugs. In a particularly calamitous example, $50 million was stolen from a contract called *The DAO* on the Ethereum blockchain [3]. Obsidian contains core features – namely, first-class typestate, linear resources, and path-dependent types – that will allow users to write safe, effective programs. The target user base of Obsidian is business professionals who will use the language to write smart contracts, and its design is thus oriented towards this domain. Obsidian programs consist of contracts – similar to classes in Java – which contain fields, states, and transactions – similar to methods.

PLATEAU'17 Workshop on Evaluation and Usability of Programming Languages and Tools, October 23, 2017, Vancouver, CA

1.1 Typestate

We (and others) have observed that programs in blockchain-related domains are typically state-oriented [9]. Furthermore, the DAO exploit stemmed from invoking a function in an external contract while the calling contract was in an inconsistent state. In light of this, Obsidian makes state first-class: an object in Obsidian has a mutable state that restricts which transactions can be invoked on it [4].

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

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

Fig. 1. An example of states in Obsidian.

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. In the example in Fig. 1, 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.

1.2 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). Linear types [5] are an existing approach that allow the compiler to 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
   resource contract Money { ... }

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

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

Fig. 2. Linear resources in Obsidian

1.3 Path-Dependent Types

There are certain cases in blockchain programs in which a programmer may want a resource to be dependent upon a specific contract. For example, we may want each instance of the `Treasury` contract in Fig. 2 to mint its own kind of money. If money `m1` from treasury `t1` is used with a distinct treasury `t2`, we would like the compiler to give an error message. Obsidian supports this via the inclusion of path-dependent types [6], wherein 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 types assist in writing secure contracts: without such types, every 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 a specific treasury is used in transactions within that treasury.

1.4 Usability

It is crucial that real programmers are able to write *correct* Obsidian code easily and efficiently. But even with seemingly intuitive features, it is not always clear which design is most effective for programmers. 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`, does it make sense to call `b(x1)` after transitioning to state `S1`? Lexically the contract is still in the `Start` state, so it is not immediately apparent which variables and transactions are in scope. In a nontrivial 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 [7]. It is critical that users are able to understand and use states easily and effectively – if not, there is potential to create the same or worse bugs, including security weaknesses, as in a language without states.

Similar questions of usability arise with linear resources and path-dependent types. Despite their apparent utility, linear types have seen limited adoption in popular programming languages (though Rust uses a form of linearity for alias control [10]), so it is not clear what is the best user-facing approach to integrating these types into our design. As for 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 the potential benefits of those features are lost.

We are conducting a user study to investigate the usability of state transitions and path-dependent types in Obsidian (a study of the usability of linear resources will occur in future work). This study is consistent with prior work that showed

the applicability of human-computer interaction techniques to programming tools [8]. One technique we use is the *natural programming* approach, in which we ask participants how they would like to express their solutions to programming problems [11]. The key research questions we address in the study are as follows:

- **RQ1**: Are states and path-dependent types a natural way of approaching the challenges that arise in blockchain programming?
- **RQ2**: Which (if any) of our proposed ways of presenting states, state transitions, and path-dependent types is most understandable and usable by programmers?
- **RQ3**: Are people able to effectively use and understand path-dependent types as they are implemented in Obsidian?

2 User Study Design

This study was exploratory rather than evaluative: its purpose was to give us information about the usability of state transitions and path-dependent types so we can make informed choices about the design of our language. We chose these features as the focus of our study because they are both key safety features that we expect users will use frequently. In addition, they both have several distinct options for their syntactic representation, and the results of the study will factor into which design we choose to implement.

Participants were asked to complete two programming exercises, both of which were divided into multiple parts that gradually introduced the participant to Obsidian and its main features. Participants were instructed to think aloud throughout the exercise, and were permitted to ask questions. We designed the study so that both exercises could be completed in approximately an hour and a half, in order to make it easier for us to find willing participants as well as to limit participant fatigue. We obtained IRB approval for the study; participants gave informed consent and were paid $10/hour for their time.

2.1 Voter Registration Exercise

Participants were given a description of a voter registration system for a hypothetical democratic nation. The system had certain stipulations that made a state machine a logical means of representing its required behaviors. For instance, the system had specific conditions under which a citizen either became registered to vote or remained unregistered.

The exercise was divided into five parts. In part **one**, participants were asked to implement the system using pseudocode. They were encouraged to invent any language features that they wanted in order to solve this problem. Our goal was to see how people naturally solve a problem in this domain: what ideas do they have, and what assumptions do they make?

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

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

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

In option 1, shown in Fig. 3, users were allowed to use any transaction available in the current *dynamic* state regardless of the lexical context. For instance, it is legal to use the `toS2` transaction (on line 5) inside the `Start` state, even though that transaction is defined within `S1`. This is because there is a transition to `S1` in the previous line.

```
1  contract C {
2      state Start {
3          transaction t(int x) {
4              ->S1{x1 = x};
5              toS2();
6          }
7      }
8
9      state S1 {
10         int x1;
11
12         transaction toS2() {
13             ->S2{x2 = x1};
14         }
15     }
16
17     state S2 {
18         int x2;
19     }
20 }
```

Fig. 3. Option 1 for state transitions

In option 2, shown in Fig. 4, each state had a constructor that was invoked when the contract transitioned to that state. With this option, there could not be any code following a state transition; thus, a transition had to be the final line of a transaction.

In option 3, shown in Fig. 5, there were conditional `if in {state}` blocks, which allowed the user to lexically nest states so that another state's transactions and fields could be used directly.

```
1  contract C {
2      state Start {
3          transaction t(int x) {
4              ->S1(x)
5          }
6      }
7
8      state S1 {
9          int x1;
10         S1(int x) { // State constructor
11             x1 = x;
12             ->S2(x1);
13         }
14     }
15
16     state S2 {
17         int x2;
18         S2(int x) { // State constructor
19             x2 = x;
20         }
21     }
22 }
```

Fig. 4. Option 2 for state transitions

Participants were asked to complete a short Obsidian contract once for each option. The contract was designed to be a simple yet non-trivial use of state transitions that illustrated the benefits and drawbacks of each option. The goal of this part was to see whether participants would be able to implement the contract successfully with each option, as well as to gather feedback about which option they preferred and why.

```
1  contract C {
2      state Start {
3          transaction t(int x) {
4              ->S1({x1 = x})
5              if in S1 {
6                  ->S2({x2 = x1})
7              }
8              if in S2 {
9                  ...
10             }
11         }
12     }
13
14     state S1 {
15         int x1;
16     }
17
18     state S2 {
19         int x2;
20     }
21 }
```

Fig. 5. Option 3 for state transitions

In part **five**, participants were asked to pick one of the three options for state transitions and use it to complete the Obsidian program from part three. They were 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 were offered a description of a program that allowed users to create and participate in lotteries. The criteria for these lotteries was listed as follows:

- A lottery has a secret *winning number* between 0 and 100.
- Anyone can purchase a lottery ticket for a set amount of money. A lottery ticket also has a number (picked by the buyer) between 0 and 100. If the lottery ticket's number is equal to the lottery's number, that is a winning ticket.
- The buyer of a ticket can check whether a ticket is the winning ticket and redeem a winning lottery ticket for a set amount of money. The buyer of a ticket can redeem the ticket at any point after they buy it.
- It is only possible to redeem a lottery ticket from the lottery where the ticket was putchased. For instance, if you buy a lottery ticket from lottery1, you cannot redeem your ticket from lottery2, even if that ticket's number matches the winning number of lottery2.

The exercise was divided into two parts. Part **one** mirrored the voter registration exercise in that participants were asked to implement the program using pseudocode. Again, we wanted to see how people naturally go about solving this problem. Would using path-dependent types – or some feature similar to that – occur to anyone? In the most-recently revised version of this exercise, participants were given the following instructions:

> Design, using pseudocode, a program to handle this lottery system. Do not worry about writing code that resembles any particular language, and feel free to make up any language features you may want. We want to see the kind of code you would want to be able to write. The goal here is to see how people naturally go about solving a problem like this.
>
> It is critical to the system that a lottery ticket can be redeemed only from the lottery where it was purchased. You can assume that creator and players of the lottery have accounts of some sort from which money can be withdrawn and deposited.

The instructions were worded carefully to motivate the use of path-dependent types, without fully expounding this feature or positioning it as the only way to implement the program.

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

3 Discussion of Study Design

One challenge in designing 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, but complex enough that implementing them was a non-trivial problem that actually motivated the use of Obsidian's features. Several initial studies revealed that our exercises were too complicated, and participants took much longer to 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.

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 targeted exactly one feature: the voter registration exercise was 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. They were able come up with better and more interesting solutions to the problems and write Obsidian code more effectively. But 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 large, complicated contract. Testing these issues is left for future work.

4 Preliminary Results

We recruited a convenience sample of 12 participants. They had varying levels of programming experience: some were beginners, some experts. None had any knowledge about Obsidian prior to completing the study. Nine of the participants were undergraduates studying computer science; one was a computer science Ph.D student, and two were working in a business-related fields. Since this was an exploratory study, we revised the study materials after each participant according to what we learned about the materials or the language design choices, and we asked participants to complete either one or both exercises according to our experimental design needs and the participants' time constraints. Some participants only completed several parts of one exercise due to time constraints.

4.1 Voter Registration Exercise

Seven participants were given the voter registration exercise. When asked to write pseudocode for the voter registration exercise, the general approach every participant took was to create a globally accessible list that stored the registration status (either registered or unregistered) of each citizen. While this is a logical implementation of the problem, it was not completely secure. For example, some participants created separate lists for registered and unregistered citizens, meaning that it would be possible for a citizen to erroneously appear on both lists.

Six out of these seven participants were shown a state diagram and asked to modify their pseudocode to use states. Of these six participants, two created explicit state blocks with functions and variables inside, similar to the design of Obsidian. The rest either maintained a global state variable that changed based on the status of a citizen, gave each citizen a state field that changed based on the citizen's status (e.g. with syntax such as "`Citizen.state = CANVOTE()`"), or created empty, immutable states at the top of the program. Several participants did not check whether a citizen was unregistered before processing their application, meaning it would be possible for an already registered citizen to register again – something we expressly prohibited in the instructions.

When looking at and writing Obsidian code with states, participants asked a lot of questions about what should be allowed to happen during and after a state transition – that is, what variables are and are not in scope, what the keyword "`this`" refers to, and what transactions can be used. Several participants asked if there was any way to check which state the contract was in. One participant noted that he felt it should never be allowed to call transitions from one state while lexically in another, saying "I'm calling S1's transaction from code for Start." Another participant said that she felt that state transitions were like return statements, and after completing a transition there should not be any more code in that transaction.

Three participants preferred the option that included state constructors, maintaining that this option was easier to understand. One preferred the option with `if in {state}` blocks because it made it immediately apparent which state a contract was in. The remaining three participants either did not express a preference or did not complete this part of the exercise.

4.2 Lottery Ticket Exercise

Six participants were given the lottery ticket exercise. When asked to implement the program using pseudocode, four out of the six defined a class for Lottery. The instructions specified that users of the program should be able to buy a ticket from any lottery, but must only be able to redeem a winning ticket from the lottery where they bought the ticket. Four out of the six implemented a program without immediately recognizing or forming a solution to this problem. When the study facilitator pointed out the issue, the approach all four participants took to resolve it was to give every lottery a fixed ID. They then made sure that the function that redeems a ticket must check that the ticket's ID is equal to the lottery ID.

This implementation left some room for exploitation. Two participants made the lottery's ID a randomly generated number, meaning it would be possible for two lotteries to have the same ID. One participant had ticket owners input the lottery ID themselves upon redeeming the ticket, meaning that if a ticket owner somehow found the ID of a different lottery, they could redeem their ticket from there. In each case, the participant was able to understand the need to have lottery tickets be tied to lotteries in some way, but four out of the six participants had trouble executing this easily and effectively.

When offered an explanation of path-dependent types and given an Obsidian program to complete, all participants were able to write correct (albeit very simple) code. Five participants were asked to identify the types of two lottery tickets that had been purchased from different lotteries. Since lotteries and lottery tickets had an established dependency relationship, the correct answer was that they had different types, even though they were both lottery tickets. Of these five participants, two were able to offer this answer with accurate reasoning. One vaguely said that it "seems" like they should have different types, but was not sure since it was not indicated explicitly in the code.

Three participants noted that the use of nested classes was confusing or unclear – one said, "it's usually bad practice to use nested classes in Java."

5 Discussion

The results of the study addressed each of the research questions presented in the introduction.

- **RQ1**: Are states and path-dependent types a natural way of approaching the challenges that arise in blockchain programming?

The results of the pseudocode portions of both exercises indicate that states and path-dependent types are not necessarily the most obvious or natural ways of solving the problems we presented. This makes sense given the backgrounds of our participants: most of them noted that they were writing pseudocode resembling the language they were most comfortable with, and thus may not have thought about inventing new, unfamiliar language features. Moreover, the simplicity of both exercises – the voter registration exercise in particular – may have made the need for these features somewhat opaque. Still, we found it encouraging that two participants independently invented special syntax denoting states, with appropriate scoping for fields and transactions. Further, we found in many cases that the approaches participants did take to implementing the programs (reflecting approaches representative of commonly-used languages) were insufficient or unsafe, and that the weaknesses in their programs could have been solved by using states or path-dependent types. This result, coupled with our strong background evidence of the utility of these features, motivates us to continue developing these features in Obsidian, while further investigating the best ways to design, present, and evaluate them.

- **RQ2**: Which (if any) of our proposed ways of presenting states, state transitions, and path-dependent types is most understandable and usable by programmers?

The results of the voter registration exercise indicate that a majority of participants prefer to specify code that executes after state transitions using state constructors. The fact that participants preferred this option after a short coding exercise is not conclusive proof that this is the best option or the one we should implement; however, it does indicate that Obsidian users want it to be simple and easy to tell which variables and transactions are in scope and to lexically determine the current state of an object. Our results offer evidence that encapsulating all the actions of a state within that state may allow users to understand more easily which state an object is currently in and which transactions and fields they are allowed to use – thus enabling them to write better code.

We have already revised the language to move transactions outside of states as a result of this study, and plan to offer IDE-based information to users about which transactions and fields are available in each state. For example, the editor will show each field in the scope of every state in which that field is available.

- **RQ3**: Are people able to effectively use and understand path-dependent types as they are implemented in Obsidian?

The responses we received from participants in the lottery ticket exercise reveal that nesting contracts is likely not the most understandable way to express a dependency relationship. Three participants made comments about this, and those who did not were not able to identify path-dependent types correctly. An alternative to this approach would be to prohibit nesting and instead use a keyword to denote this relationship (e.g. `"resource contract LotteryTicket depends on Lottery"`).

6 Limitations

This study has a number of limitations:

- Because this was a pilot study, we recruited mostly computer science students rather than people in the business domain. Our convenience sample does not reflect the intended user base of Obsidian, and we may have thus missed out on the insights that business professionals could offer, as well as the challenges in encouraging such users to write safe programs.
- Participants were likely influenced by their prior programming experience: people may have had a tendency to prefer syntax that was familiar to them, even if it was not the most effective means of implementing a program.
- Participants were likely influenced by the descriptions of the tasks as well. Because the tasks were designed by the same people who are designing Obsidian, they they may have been described in a way that aligns with the DSL's design more closely than realistic domain-appropriate tasks would be.

7 Future Work

We are continuing to refine the language features and test them with further user studies.

- We will target 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 attempting to access an owned field that has already been consumed, and how can one transition between states with different owned fields?
- 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 understanding of Obsidian's features and thus offer more constructive feedback about Obsidian's usability.
- Finally, we plan to modify the Obsidian language implementation using the results of these studies, and evaluate the final design in a formal study testing its effectiveness.

8 Conclusion

We designed and conducted an exploratory pilot study of the usability of two of Obsidian's major safety features. Preliminary results from this study will inform both the design choices we will make in the language as well as the direction that future user studies will take. The natural programming approach we took in the design of the user study allowed us to obtain insightful, specific, and inventive responses from participants, which may not have been possible with more structured testing methods such as A/B testing. By using a human-centered approach in the design of Obsidian, we aim to offer a blockchain-based programming language that allows users to write smart contracts more safely and easily than currently available blockchain languages.

References

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

[2] Ethereum Foundation, "Solidity," https://solidity.readthedocs.io/en/develop/. Accessed Aug. 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] J. Aldrich, J. Sunshine, D. Saini, Z. Sparks, "Typestate-Oriented Programming," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications,* 2009, pp. 1015-1022.

[5] P. Wadler, "Linear Types Can Change the World," IFIP TC, vol. 2, pp. 347 - 359, 1990.

[6] N. Amin, T. Rompf, and M. Odersky. "Foundations of Path-Dependent Types." in OOPSLA, 2014.

[7] 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.

[8] 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.

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

[10] N. D. Matsakis and F. S. Klock, II. 2014. "The Rust language." Ada Lett. 34, 3 (October 2014), 103-104.

[11] B.A. Myers, J.F. Pane, A. Ko, "Natural Programming Languages and Environments", Comm. ACM, vol. 47, no. 9, pp. 47-52, 2004.