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 are taking a user-centered approach in the design of a new programming language, Obsidian, to make it easier for programmers to write correct programs while leveraging a type system to provide strong safety guarantees. This paper describes how experiments with programmers have informed the design of the language.**

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

I. Introduction

Ethereum [1] and Hyperledger Fabric [2] 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 (DAO) could sell shares to parties in exchange for virtual currency. The DAO's 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 execute *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, contracts with bugs are vulnerable to attack. Such vulnerabilities resulted in the loss of over $70M in the DAO and Parity attacks [3], [4].

Existing approaches aim to make blockchain development safer by layering checks on top of existing languages, such as Solidity [5]. For example, 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 [6]. Atzei et al. presented a taxonomy of vulnerabilities that are common among Ethereum programs [7]. In contrast, 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. Some safety properties will be guaranteed by the compiler, but it would not suffice to offer guarantees in a language that programmers cannot use effectively. Instead, we emphasize *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 or Java?

Several characteristics of blockchains 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. Second, bugs in programs cannot be fixed easily because the programs are immutable. Once money is committed to an agreement (as implemented in a contract), the money can only be removed according to the existing contract implementation. Third, blockchain programs are commonly *state-based* [8]: finite-state machines provide a simple abstraction for high-level aspects of program behavior. We exploit this in a novel language design using *typestate*, based on evidence that state-based reasoning facilitates faster, less buggy development [9].

Typestate [10] is the practice of making state information a first-class part of types [15] to facilitate static reasoning about states of objects. For example, the Obsidian type `RockPaperScissors@AcceptingBets` is a reference to a `RockPaperScissors` object that is statically known to be in state `AcceptingBets`. When typechecking a method invocation, the compiler ensures that the invoked method is statically known to be available all of the possible current states of the referenced object. This introduces a problem of alias analysis: if the referenced object is mutable, then the compiler must be able to prove that there are no other aliases of the same object through which the object's state may be mutated. Otherwise, the state specification in one alias may be violated by a mutation via another alias. The usual solution involves complicated systems of *permissions*, which annotate references with specifications regarding which operations can be performed via those references. Our design challenge in Obsidian is to create an approach that is expressive, safe, and also easy to use effectively.

In addition to addressing serious problems that blockchain software developers face, Obsidian also serves as a testbed for research on *language design methodology*. Language designers create languages in order to obtain particular guarantees,

assuming that programmers will learn to use their tools if they have been shown to have useful properties. However, we and others argue that it is better to take users into account directly during the design phase for programming languages [11], [12]. In particular, Obsidian integrates several sophisticated type system-based approaches to improve safety, but the usability of these techniques has never been studied formally. The evidence that users will be able to use language designs effectively *when writing code* is typically based on the experiences of researchers using their own systems, which may not generalize to users who are not programming language researchers.

Our view of programming languages as *user interfaces* motivates us to make Obsidian as usable as possible by programmers. We conducted *formative user studies* to inform language design changes that improve usability. When Obsidian implementation is complete, we will conduct *summative* studies to compare Obsidian with Solidity directly. In this paper, we describe how qualitative user studies to date have driven significant changes in the design of Obsidian. Due to space constraints, we will limit our discussion to a study of typestate, permissions, and ownership, though we have investigated other aspects as well.

We will next introduce some sources of bugs in existing blockchain applications that we address in Obsidian. Then, we show by example how Obsidian addresses those vulnerabilities. Finally, we explain how we have used user studies to improve the Obsidian language design.

II. Bugs and solutions

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

1) Our analysis of proposed blockchain applications showed many applications are structured around high-level **states**, on which the available operations depend. For example, a `Bond` contract might initially be in `Offered` state, but once purchased, it is in `Purchased` state and no longer available for sale. Traditional languages require dynamic checks to ensure that operations are only executed in appropriate states, but this only allows bugs to be detected an runtime, not compile time. Obsidian uses typestate to lift dynamic state information into types to facilitate static reasoning about state.
2) **Money** is owned by contracts. Any bugs in code manipulating money or other resources could result in loss or misappropriation of resources. Delmollino et al. showed that it is common for blockchain contracts written by beginners to accidentally leak money [14]. Obsidian uses a *linear type system* [13] for quantities of money so that the compiler can guarantee lossless tracking of financial information. Unlike traditional types, *linear* types represent valuable resources that must not be accidentally lost.
3) **Re-entrancy attacks** occur when a contract allows inconsistent state to be exposed to an external caller. Obsidian detects reentrancy dynamically to avoid a large class of security vulnerabilities. Due to space constraints, this paper does not focus on re-entrancy attacks.

III. Obsidian language example

As is traditional in typestate-oriented languages, 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 1a (inspired by Delmollino et al. [14], who studied errors that beginners make implementing a similar application in Solidity). The creator of the contract offers to place bets with each of `capacity` players. After placing bets, players make their choices. Eventually, `finalize()` is called to distribute payouts. In line 20 of Fig. 1a, the winner is sent the reward. However, when the `call()` function is invoked to send the money, 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.

In Solidity, accounting for money occurs through potentially-inconsistent mechanisms. Each contract holds virtual currency, but in addition, contracts may need to track finer-grained information. For example, a bank records the balance of each account rather than only tracking the total amount of deposits. Thus, bugs may result in the bank losing track of who owns money.

Fig. 1b 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. By treating money as a *linear* resource, we can statically detect a class of errors involving money and other linear resources. For example, the implementation of `payCreator` (Fig. 1b, line 4) acquires ownership of the input money, and the compiler will give an error if the money reference goes out of scope. By sending money in parameters of methods, rather than in generic `call` or `transfer` methods, the contract can handle received money and other resources in a disciplined, application-appropriate way.

IV. Support for Typestate, Permissions, and Ownership in Obsidian

A. Design and initial user experiments

We initially took a traditional *type declarations* approach to the language design, similar to that used in Plaid [15] and Plural [17], as well as in session type approaches [18]. For example, suppose a `LightSwitch` class is defined with states `On` and `Off`. Consider a method body that includes the following:

```
LightSwitch@Off s = new LightSwitch();
s.turnOn();
```

`LightSwitch@Off` is the type of a reference to a `LightSwitch` object that is in `Off` state. After the above code executes, the type of `s` is inconsistent with its declaration: the new type is `LightSwitch@On`. Typestate specifications can also occur in method parameters, in which case the typestate at the beginning and end of the method can be specified:

```
contract RockPaperScissors {
  enum State {AcceptingBets, MakingChoices, PayingOut, Complete}
  State state;
  address[] otherPlayers;

  function RockPaperScissors() {
    state = State.AcceptingBets;
  }

  function bet() payable {
    if (state != State.AcceptingBets) {
      revert;
    }
    ... // Record the money if it is the right amount, otherwise abort.
    if (allBetsPlaced()) {
      state = State.MakingChoices;
    }
  }

  function makeChoice(int c) {
    ... // Body omitted; similar in structure to bet()
  }

  function finalize() {
    if (state != State.PayingOut) {
      revert; // abort the transaction
    }

    for (int i = 0; i < otherPlayers.length; i++) {
      address winningAddress = computeWinner(i);
      // Bug 1: call executes external code, which may cause a re-entrant call.
      // Bug 2: failed to check return value of call().
      winningAddress.call.value(wonAmount()); // transfer money
    }
    state = State.Complete;
  }
}
```

(a) Naïve Rock-Paper-Scissors in Solidity, showing bugs

```
contract RockPaperScissors {
  type PayoutFunction = Money -> unit;

  PayoutFunction payCreator;
  Money@Owned pool; // stores funds from bets
  PayoutFunction otherPlayerPayouts[];

  state AcceptingBets;
  state MakingChoices;
  state PayingOut;

  transaction bet(Money m, PayoutFunction payout) available in AcceptingBets {
      ... // store payout function for later use
      if (allBetsPlaced()) {
        ->MakingChoices; // transition to MakingChoices state
      }
  }

  transaction makeChoice(Choice c) available in MakingChoices {
    ... // Body omitted; similar in structure to bet()
  }

  transaction finalize() available in PayingOut {
      // This shows how resource ownership is transferred in transaction calls.
      for (i = 0; i < otherPlayers.length; i++) {
        Money@Owned bet = pool.split(wonAmount());
        if (creatorWon(i)) {
          payCreator(bet);
        }
        else {
          otherPlayers[i].payout(bet);
        }
      }
    }
  }
```

(b) Rock-Paper-Scissors in the final version of Obsidian

Fig. 1. A comparison of Rock Paper Scissors implementations (abbreviated)

```
transaction turnSwitchOff(LightSwitch @ On >> Off s)
{
  s.turnOff();
}
```

The body of `turnSwitchOff` can assume that the object referenced by `s` is in state `On` at the beginning and must ensure that it is in state `Off` at the end. Obsidian also supports the state specification `?`, which means that the typestate is statically unknown. For example:

```
\texttt{randomizeSwitch(LightSwitch @ On >> ? s)}
```

takes a `LightSwitch` in `On` state and provides no postcondition regarding its state. When typestates are specified on *field* declarations, every method must end with the field referencing an object in corresponding state so that methods can assume that fields are in appropriate state on entry.

In typical typestate-oriented languages, each reference can include a *permission*, which specifies what operations that reference supports. In the model by Garcia et al. [16], there are three possible permissions: `full`, `shared`, and `pure`. `full` provides exclusive write access (ownership); `shared` provides non-exclusive write access; `pure` provides read-only access.

In Obsidian, the user may define certain contracts to be *resources*. Obsidian facilitates reasoning about resources: although there may be many references to an instance of a resource, one of them is treated *linearly*: it represents the *owner* of the resource. The compiler ensures that the owning reference cannot go out of scope, which is the only way to lose an object; instead, ownership must be transferred to a new owner, such as by assigning it to a field of appropriate type. Note that this notion of resource ownership coincides with the concept of a `full` permission in that they both designate a "unique" reference to an object. Does this necessitate replicating the entire permissions system, despite its complexity? Integrating the overlapping concepts of typestate and resources into the same language, then, presents a novel design problem, especially given the design goal of making the language easy to use.

Any reference that bears a typestate specification must also be an owning reference — otherwise there could be *two* typestate-bearing aliases to a potentially-mutable object, which is unsound because using one to mutate the object's state could make the object inconsistent with the specification on the other reference. We used the `owned` keyword only for non-typestate-bearing references, leaving ownership of typestate-bearing references implicit. This notation is concise but inconsistent because not every owned object is explicitly annotated `owned`.

After a sequence of operations involving an object with typestate, it may be unclear to a reader of the code what the compiler knew about the state, since the state may be inconsistent with the one in the variable's declaration. A reader would have to trace an object through each usage to see what its typestate was at any given program point. The fact that the compiler's knowledge of types is not fixed at type declaration is a significant departure from traditional languages. We are interested in designing a language whose semantics and syntax would *help* users reason both about their programs and about the compiler's knowledge of object state. If users have difficulty reasoning about the compiler's knowledge of object state, they will receive compiler error messages that they may find surprising and difficult to fix when they do not understand why the compiler cannot reason about states and mutation as effectively as they can.

We conducted *User Study 1* to evaluate our approach to ownership transfer [19]. Among other findings, we observed that participants had difficulty understanding when ownership was transferred in assignment and in method invocation. We

had specified that ownership transfer occurred when an owned reference was passed as a parameter to a method that took an owned reference. For example:

```
owned Money m = ...
spend(m); // transfer ownership to parameter of spend()
spend(m); // COMPILE ERROR: n is no longer owned
```

Unfortunately, participants found this approach confusing, in part because the semantics of invocation depended on the signature of the method.

B. User Study 2

We conducted user experiments to gather empirical data on the usability of these different approaches to typestate, permissions, and ownership. In the experiments, we asked participants to do programming tasks. However, there is a difficult practical problem running lab studies on a new programming language: if the language is very different from those with which a participant is familiar, there is a learning stage where the participant must learn the new language. A corresponding problem is one of conflation. If one teaches participants a new language, the participants are likely to have difficulty with many different aspects of the language, not just the one of interest. Furthermore, when a participant is confused, it may not be clear whether the cause is the design decision itself or some unrelated point of confusion or aspect of the language. The effects of the design decisions become overwhelmed by the noise from unrelated parts of the language.

We observe, however, that languages are designed so that features are as orthogonal as possible [20]. Therefore, our approach is to study the design decisions in isolation by *back-porting them to a language with which participants are already familiar*. This approach introduces its own kind of noise: perhaps the aspects do not behave in the second language as they would in the first, and the outcome might not be applicable to the first language. We plan to address this limitation of the external validity in the future by conducting a summative usability study of the complete Obsidian system. However, we limit the impact of any differences by choosing Java, which is structurally similar to Obsidian. For example, both Java and Obsidian are statically-typed object-oriented languages, and both languages share similar difficulties with aliases to mutable state. Rather than implementing a permission system in Java, we conducted a Wizard-of-Oz study [21] where participants received documentation on an extension to Java and the experimenter provided simulated compiler error messages. Importantly, in usability studies such as this one, the goal is to efficiently find as many usability problems as possible; the assumption is that problems faced by one participant would likely be faced by many others.

User Study 2 began by giving participants a prescription-tracking system implemented in Java. The system was seeded with a bug: it was possible for a patient to fill a prescription more times than allowed by depositing it in multiple pharmacies. To justify that this bug is worth detecting with a type system, we first confirmed that detecting this bug was difficult for at least some participants. We gave the first two participants 30 minutes to find the bug; one participant found it just as the 30 minutes elapsed, and the other did not find it at all. Then we asked all five participants (all experienced Java programmers) to fix the bug with an annotation we added to Java, `@Owned`. We gave a tutorial explaining `@Owned` but we were interested in seeing what assumptions participants made about the behavior, so we did not give exhaustive explanations.

Participants found this task very difficult in our initial, traditional ownership design. Some participants had difficulty reasoning about ownership in a *static* way rather than a dynamic way. For example, one participant wrote `if (@Owned prescription)`, attempting to indicate a dynamic check of ownership. Several participants had difficulty figuring out which variables should be owned, perhaps related to their confusion about the static nature of ownership.

In a second part of User Study 2, we provided a tutorial on a proposed Java extension that supports resources and typestate, and asked participants to add typestate specifications where possible in a small program that manipulated bonds. But because using typestate requires using ownership effectively, many participants had difficulty with these tasks. However, we observed a common expectation that the compiler would do typestate inference. For example:

```
1  LightSwitch s = new LightSwitch();
2  s.turnOn(); // ERROR: turnOn() may not be available
```

In this initial language, typestate must be specified in declarations to cause the compiler to reason about typestate. In contrast, participants expected code akin to the above code to not give an error because they expected the compiler to infer on line 1 that `s` refers to an object in state `Off`.

One participant was confused about the relationship between ownership and typestate, expecting that all owned references had to include a state specification. Another felt that state postconditions were redundant with a body that ended in a state transition, suggesting a dynamic view of the semantics rather than a static view.

C. Language revisions

To address the difficulties we observed understanding when ownership transfer occurred and resolving the inconsistency between type declarations and static typestate knowledge, we designed an alternative notation that uses *static assertions* to denote state knowledge. For example:

```
LightSwitch s = new LightSwitch(); [s @ Off]
s.turnOn(); [s @ On]
```

The static assertions, shown in square brackets, indicate assertions that the compiler checks. Note that the declaration does not include a typestate specification; this way, typestate of local variables is consistently only shown with assertions. This approach gives the expected property that the declared type of a variable never changes.

Since every typestate-bearing reference also carries ownership, the revised version of Obsidian considers ownership to be a special case of typestate: one in which the state is not specified. This allows use of the same notation for

both ownership and typestate. We added keywords `Owned` and `Unowned` to denote these types, as in `Money@Owned`. This notation allows us to avoid exposing permissions as a separate concept; operations on objects are permitted or refused according to a consistent way of specifying properties of references.

D. User Study 3

User Study 3 is an experiment that compares the original approach (which had *separate* notions of ownership and typestate; lacked static typestate assertions; and required typestate specifications in local variable declarations) with the new approach. In contrast with Experiment 2, this experiment consists of both qualitative and quantitative phase. We developed a Web-based experiment, which asks participants to answer questions about programs and also write some code. As with User Study 2, we adapted the concepts of Obsidian to a Java context so that we need not demand that they learn a fresh language. The experiment first introduces participants to typestate, and then describes ownership and its implications on typestate. In Part 1, we compare typestate inference to declarations, and we evaluate the effect of static assertions. In Part 2, we compare explicit, separate ownership and typestate to implicit ownership merged with typestate.

The qualitative phase took place in a laboratory setting, allowing us to use a think-aloud protocol so that we could understand in detail which aspects of the design were confusing. After each participant, we refined the questions and instructions and changes to the language itself. For example, the language originally only required type specifications on some references to objects that maintain state; we found that people found it confusing to keep track of which behaviors were the default, so we now include typestate specifications except when the typestate is read-only. Participants in the final version of User Study 3 were generally able to reason successfully about typestate and ownership.

The quantitative phase is planned to begin soon, after we complete a formal analysis of the revised type system. We plan to recruit Java programmers via Facebook ads in order to get a broader sample of programmers than we would likely find at a university. In this phase, programmers will be randomly assigned to one of the experimental conditions, and we will compare completion times for the tasks and correctness of the participants' answers to the programming questions across the two experimental conditions.

V. Future Work

We will formally prove appropriate safety properties, such as conservation of money. We will also conduct case studies to evaluate how suitable the language is for expressing real-world applications that are in use on current blockchains. Finally, we will conduct summative 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. This approach leverages our past experience evaluating and improving usability of programming languages [22].

VI. 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. Our design methodology demonstrates how language designers can take users into account directly when designing programming languages in order to make them more effective for programmers.

Acknowledgment

The author thanks the reviewers as well as Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 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.
[2] The Linux Foundation, "Hyperledger," https://www.hyperledger.org.
[3] E. Gün Sirer, "Thoughts on the DAO hack," 2016. [Online]. Available: http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/
[4] L. Graham. (2017) $32 million worth of digital currency ether stolen by hackers. [Online]. Available: https://www.cnbc.com/2017/07/20/32-million-worth-of-digital-currency-ether-stolen-by-hackers.html
[5] Ethereum Foundation, "Solidity," https://solidity.readthedocs.io/en/develop/.
[6] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *ACM CCS*, 2016.
[7] 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.
[8] Ethereum Foundation, "Common patterns," http://solidity.readthedocs.io/en/develop/common-patterns.html.
[9] J. Sunshine, J. D. Herbsleb, and J. Aldrich, "Structuring documentation to support state search: A laboratory experiment about protocol programming," in *ECOOP*, 2014.
[10] K. Bierhoff and J. Aldrich, "Modular typestate checking of aliased objects," in *OOPSLA*, 2007.
[11] A. Stefik and S. Hanenberg, "The programming language wars: Questions and responsibilities for the programming language community," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2014.
[12] M. Coblenz, J. Aldrich, J. Sunshine, and B. Myers, "Interdisciplinary programming language design," *Dagstuhl Conference on Evidence About Programmers for Programming Language Design*, Feb. 2018.
[13] P. Wadler, "Linear types can change the world," in *IFIP TC*, vol. 2, 1990, pp. 347–359.
[14] 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*, 2015.
[15] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and É. Tanter, "First-class state change in Plaid," in *ACM SIGPLAN Notices*. ACM, 2011.
[16] R. Garcia, E. Tanter, R. Wolff, and J. Aldrich, "Foundations of typestate-oriented programming," *ACM Trans. Program. Lang. Syst.*, 2014.
[17] K. Bierhoff, N. E. Beckman, and J. Aldrich, "Checking concurrent typestate with access permissions in plural: A retrospective," *Engineering of Software*, pp. 35–48, 2011.
[18] F. Pfenning and D. Griffith, "Polarized substructural session types," in *Foundations of Software Science and Computation Structures*, 2015.
[19] C. Barnaby, M. Coblenz, T. Etzel, E. Kanal, J. Sunshine, B. Myers, and J. Aldrich, "A user study to inform the design of the Obsidian blockchain DSL," in *PLATEAU*, 2017.
[20] R. W. Sebesta, *Concepts of Programming Languages, Seventh Ed.*, 2006.
[21] P. Green and L. Wei-Haas, "The wizard of oz: a tool for rapid development of user interfaces," no. UMTRI-85-27, June 1985.
[22] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine, "Glacier: Transitive class immutability for Java," in *ICSE*, 2017.