Can Advanced Type Systems Be Usable? An Empirical Study of Ownership, Assets, and Typestate in Obsidian

MICHAEL COBLENZ, Carnegie Mellon University, USA
JONATHAN ALDRICH, Carnegie Mellon University, USA
BRAD A. MYERS, Carnegie Mellon University, USA
JOSHUA SUNSHINE, Carnegie Mellon University, USA

Some blockchain programs (smart contracts) have included serious security vulnerabilities. Obsidian is a new typestate-oriented programming language that uses a strong type system to rule out some of these vulnerabilities. Although Obsidian was designed to promote *usability* to make it as easy as possible to write programs, strong type systems can cause a language to be difficult to use. In particular, ownership, typestate, and assets, which Obsidian uses to provide safety guarantees, have not seen broad adoption together in popular languages and result in significant usability challenges. We performed an empirical study with 20 participants comparing Obsidian to Solidity, which is the language most commonly used for writing smart contracts today. We observed that Obsidian participants were able to successfully complete more of the programming tasks than the Solidity participants. We also found that the Solidity participants commonly inserted asset-related bugs, which Obsidian detects at compile time.

CCS Concepts: • **Software and its engineering** → **Language features**; **Domain specific languages**; • **Security and privacy** → *Software and application security*; • **Human-centered computing** → *Empirical studies in HCI*.

Additional Key Words and Phrases: typestate, linear types, ownership, assets, permissions, blockchain, smart contracts, empirical studies of programming languages

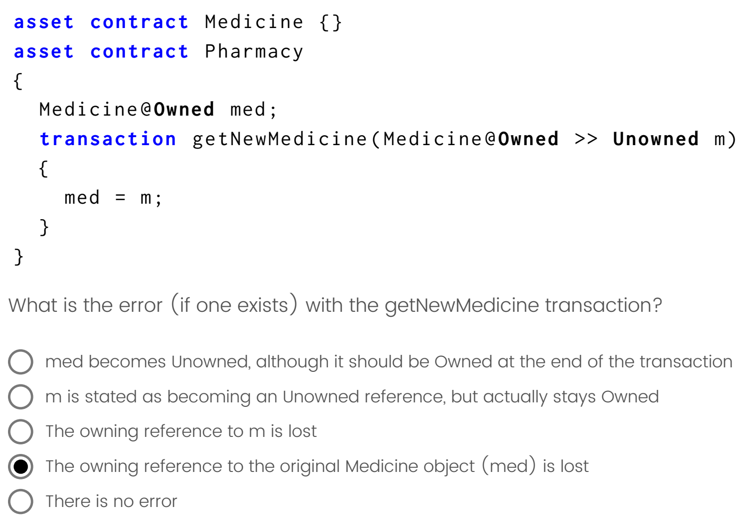
ACM Reference Format:
Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2020. Can Advanced Type Systems Be Usable? An Empirical Study of Ownership, Assets, and Typestate in Obsidian. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 132 (November 2020), 28 pages. https://doi.org/10.1145/3428200

1 INTRODUCTION

Obsidian [Coblenz et al. 2020b] is a new programming language for writing *smart contracts* [Szabo 1997], which are programs for blockchain platforms. Blockchains aim to provide a trusted computing medium among users who have not necessarily established trust. By decentralizing computation, system designers can obtain strong security properties. Unfortunately, these properties can only be obtained when the smart contracts themselves implement the intended behavior. Through

Authors' addresses: Michael Coblenz, Computer Science Department, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA, mcoblenz@cs.cmu.edu; Jonathan Aldrich, Institute for Software Research, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA, jonathan.aldrich@cs.cmu.edu; Brad A. Myers, Human-Computer Interaction Institute, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA, bam@cs.cmu.edu; Joshua Sunshine, Institute for Software Research, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA, joshua.sunshine@cs.cmu.edu.

2475-1421/2020/11-ART132
https://doi.org/10.1145/3428200



```
asset contract Medicine {}
asset contract Pharmacy
{
  Medicine@Owned med;
  transaction getNewMedicine(Medicine@Owned >> Unowned m)
  {
    med = m;
  }
}
```

What is the error (if one exists) with the getNewMedicine transaction?

- ○ med becomes Unowned, although it should be Owned at the end of the transaction
- ○ m is stated as becoming an Unowned reference, but actually stays Owned
- ○ The owning reference to m is lost
- [✓ The owning reference to the original Medicine object (med) is lost]
- ○ There is no error

Fig. 1. An example practice question on assets from the Obsidian tutorial (showing the correct answer).

bugs and security vulnerabilities in blockchains, attackers have stolen millions of dollars worth of cryptocurrencies [Graham 2017; Sirer 2016].

To improve safety and prevent vulnerabilities, researchers have developed new languages with stronger safety properties. In this work, we focus on Obsidian, which stands for *Overhauling Blockchains with States to Improve Development of Interactive Application Notation*. Obsidian was based on a requirements analysis for smart contract tools [Coblenz et al. 2019b] to provide safety properties that would be relevant to many blockchain applications. In addition, Obsidian was designed to be *usable* through iterative user testing with 44 participants [Coblenz et al. 2019a] so that programmers would be able to use it effectively. In prior work, we showed that most of the participants in a six-participant qualitative study were able to complete relevant programming tasks [Coblenz et al. 2019a]. However, that small study did not compare Obsidian to any other languages.

Ours is the first quantitative user study (of which we are aware) of a type system that supports linear types, ownership, or typestate, or any combination of these together. Earlier formative studies of Obsidian isolated particular language features, sometimes by adapting those features in the context of a language with which participants were already familiar. In contrast, in this summative study we studied the full Obsidian language, including the compiler and editor. Our study compares Obsidian to the most widely-used language for writing smart contracts, *Solidity* [Ethereum Foundation 2020b].

Although Solidity and Obsidian target different blockchain platforms (Ethereum [Ethereum Foundation 2020a] and Hyperledger Fabric [The Linux Foundation 2020b], respectively), we chose Solidity for comparison because it is the dominant programming language used in public smart contract development and because it is the language that was used to develop popular smart contracts that had serious security vulnerabilities [Graham 2017; Sirer 2016].

Considering Obsidian's design goal to improve safety with a novel type system, we identified three high-level research questions for our study:

RQ1: Could we obtain actionable data about the usability of a novel programming language (that uses a type system that would be unfamiliar to our participants) in a short-duration user study (less than one day), which would be representative of real-world smart contract development?

RQ2: Do programmers using Solidity insert more of the kinds of bugs that Obsidian is designed to catch?

RQ3: Strong type systems can impose a usability burden on programmers because the compiler forces programmers to write code so that the compiler can verify various safety properties. Can programmers who were previously familiar with object-oriented programming (but not with Obsidian, typestate, ownership, or linear type systems in general) successfully use Obsidian to complete relevant smart contract programming tasks? If so, is there a significant impact on task completion times?

In RQ1, *actionable data* refers to data that inform programming language designers about their designs in ways that have implications on language design or language training. We used these three research questions to develop task criteria (§ 7). Then, we developed three tasks according to the criteria: Auction (§ 8), Prescription (§ 9), and Casino (§ 10). In each task, we refined these high-level research questions into task-relevant ones.

In summary, after the training period (about 90 minutes), seven of the ten Obsidian participants were able to successfully use Obsidian to finish implementing the required small program in our Auction task. In contrast, only two of the ten Solidity participants finished the task correctly. In the Prescription task, seven of the ten Obsidian participants were able to fix a security vulnerability in the program, whereas only two of the ten Solidity participants were able to do so. Six of the Obsidian participants used ownership to do the task, suggesting that ownership is teachable in a relatively short training period.

Five of the Obsidian participants had enough time remaining to do the Casino task (meaning that the *other* participants felt that their solutions were incomplete when their four-hour time expired). Although four of these wrote solutions that compiled, all four abused the `disown` operator, resulting in asset loss. Among the eight Solidity participants who had enough time for this task, half inserted bugs that resulted in improper asset fabrication or loss, showing that it is still important to pursue approaches to prevent asset abuse. Only one participant, who was in the Solidity condition, finished the task correctly.

After the tasks had ended, we gave participants a post-study survey asking for their opinions about the language they had used. We found that Obsidian participants felt that ownership was useful. They also observed that the tutorial and exercises were effective tools for learning the language.

We come to three conclusions in this paper. First, the methodology we used, in which we include practice-based training in a traditional quantitative study, is effective for evaluating a novel programming language. Second, despite the strong, unfamiliar restrictions it imposes, Obsidian's ownership and permissions system can be taught to some kinds of programmers in a short period of time in a way that results in many of them being able to use it effectively. That is, for some tasks, Obsidian enables people to be more successful at writing code that is free of the serious bugs that Obsidian detects. Third, without using techniques to detect or prevent asset-loss bugs, when doing some kinds of smart contract programming tasks, programmers will accidentally insert these kinds of bugs.

2 THE OBSIDIAN LANGUAGE

Obsidian uses *ownership* to express in the type system that each *asset* (an object that has value and therefore should not be lost) has exactly one owning reference. Ownership represents a form of *linear types* [Wadler 1990]. Ownership may be transferred between references, but if the owning reference goes out of scope, then the compiler reports an error. This ensures that owning references to assets cannot be lost unless they are explicitly `disowned` by the programmer. This design was motivated by Delmolino et al. [2016], who found in a user study that programmers of smart contracts inserted bugs that lost assets.

Obsidian extends ownership with *typestate* [Strom and Yemini 1986] because blockchain programs are typically stateful with objects supporting different operations depending on their state [Ethereum Foundation 2020c]. Thus, owning references can also specify what *state* the referenced object is in. For example, `Auction@Open` is the type of a reference to an `Auction` object that is in state `Open`. Because only owning references can specify typestate in Obsidian, ownership is implied by the presence of a typestate specification. Alternatively, `Auction@Owned` is the type of an owned reference to an `Auction` object, which may be in any state. Ownership is one example of *linear* types [Wadler 1990]: types that are consumed when used rather than being duplicated, and which *must* be consumed rather than merely dropped.

To improve flexibility, Obsidian also allows `Shared` references, which can refer to an object that has no owner, and `Unowned` references, which refer to an object that may have an owner. These additional options allow users to avoid establishing ownership structures when none are needed. Unlike `Owned` and `Shared` references, `Unowned` references cannot be used to change which of the various named states the object is in. This is required for soundness, since a change in state through an `Unowned` reference might violate a typestate specification on the owning reference. These annotations (`Owned`, `Shared`, and `Unowned`) are referred to as *permissions* because they denote what operations can be done via references with those annotations.

Obsidian is a class-based object-oriented language that uses a syntax similar to Java. Obsidian uses the keyword `contract` instead of `class` due to conventions of blockchain platforms, and calls methods `transactions` because invocations of blockchain code from outside the blockchain have transactional semantics: either the execution finishes successfully and new state is stored in the blockchain, or the transaction is reverted and any state changes are not preserved. Currently, Obsidian supports the Hyperledger Fabric blockchain platform [The Linux Foundation 2020a].

We provided a full description and formal treatment of Obsidian [Coblenz et al. 2020b]; here, we explain Obsidian by example. The left column of Figure 2 shows part of an `Auction` contract. An `Auction` instance is always in one of three states: `Open`, `BidsMade`, or `Closed`. A field, `seller`, is always in scope, but when the object is in state `BidsMade`, fields `maxBidder` and `maxBid` are also in scope. Transactions specify types for their parameters, but in addition to the normal parameters, when `this` is specified as the first parameter, the program can specify a type for the `this` reference. The `bid` transaction, for example, can only be invoked via a `Shared` reference to the receiver. Formal parameters use » to denote changes in permission or state. For example, the `money` parameter of `bid` is specified with permission `Owned » Unowned`, meaning that the caller must pass an `Owned` reference, and when `bid` returns, from the perspective of the caller, the `money` is now `Unowned`. In other words, ownership was passed from the caller to the transaction.

Dynamic state tests, via the `in` operator, check the dynamic state of a referenced object. For example, line 20 checks to see if `this` is in state `Open`. If the test passes, lines 23-24 initialize the `maxBidder` and `maxBid` fields of the `BidsMade` state, to which the object transitions on line 25. Line 24 transfers ownership of an object initially referenced by `money` to the `maxBid` field, leaving `money` with type `Money@Unowned`.

Fields can temporarily have types that differ from their declarations as long as by the end of the transaction, all fields have types consistent with their declarations. For example, line 31 returns the money from the previous `maxBidder` which causes field `maxBid` to temporarily have type `Money@Unowned`; this is corrected on line 33, which transfers ownership from `money` to `maxBid`.

```
1  main asset contract Auction {
2    Participant@Unowned seller;
3
4    state Open;
5    state BidsMade {
6      // the bidder who made the highest bid so far
7      Participant@Unowned maxBidder;
8      Money@Owned maxBid;
9    }
10   state Closed;
11
12   . . .
13
14
15
16
17   transaction bid(Auction@Shared this,
18                   Money@Owned >> Unowned money,
19                   Participant@Unowned bidder) {
20       if (this in Open) {
21         // Initialize destination state,
22         // and then transition to it.
23         BidsMade::maxBidder = bidder;
24         BidsMade::maxBid = money;
25         ->BidsMade;
26       }
27       else {
28         if (this in BidsMade) {
29           //if the new bid  > current Bid
30           if (money.getAmt() > maxBid.getAmt()) {
31             //1. TODO: fill this in.
32             //Can call other transactions.
33             maxBidder.receivePayment(maxBid);
34             maxBidder = bidder;
35             maxBid = money;
36           }
37           else {
38             //2. TODO: return money to the bidder,
39             //   since the new bid was too low.
40             //Can call other transactions.
41             bidder.receivePayment(money);
42           }
43         }
44         else {
45           revert("Can't bid on closed auctions.");
46         }
47       }
48     }
49 }
```

Obsidian condition.

```
contract Auction {
  // the bidder who made the highest bid so far
  address maxBidder;
  uint maxBidAmount;

  // 'payable' indicates we can transfer money
  // to this address
  address payable seller;

  // Allow withdrawing previous money
  // for bids that were outbid
  mapping(address => uint) pendingReturns;

  enum State { Open, BidsMade, Closed }
  State state;
  . . .
  function bid() public payable {


    if (state == State.Open) {
      maxBidder = msg.sender;
      maxBidAmount = msg.value;
      state = State.BidsMade;
    }


    else {
      if (state == State.BidsMade) {
        //if the new bid  > current Bid
        if (msg.value > maxBidAmount) {
          //1. TODO: fill this in.
          //Can call other functions as needed.
          pendingReturns[maxBidder] += maxBidAmount;
          maxBidder = msg.sender;
          maxBidAmount = msg.value;
        }
        else {
          //2. TODO: return money to the bidder,
          //   since the new bid was too low.
          //Can call other functions as needed.
          pendingReturns[msg.sender] += msg.value;
        }
      }
      else {
        revert("Can't bid on closed auctions.");
      }
    }
  }
}
```

Solidity condition.

Fig. 2. A side-by-side example comparing Obsidian code to Solidity code, taken from the two conditions of the Auction task (with minor changes to fit on this page). Code highlighted in yellow represents a correct solution; the rest was given to participants as starter code. In the Obsidian code, line 33 transfers ownership of the object referenced by `maxBid` to the `receivePayment` parameter. The new type of `maxBid` from then until line 35 is `Money@Unowned`. Line 35 re-establishes ownership in `maxBid` by transferring ownership from `money` to `maxBid`.

The `revert` statement (line 45) discards all changes that have been made to state in the transaction and reports an error.

3 THE SOLIDITY LANGUAGE

Solidity [Ethereum Foundation 2020b], like Obsidian, is a class-based object-oriented language. Solidity targets the Ethereum blockchain platform [Ethereum Foundation 2020a]. The `function` keyword denotes methods, though methods have transactional semantics. It has no built-in notion of states, but programs can declare `enum`s and use them to represent states. Solidity supports a built-in cryptocurrency called *ether*. Functions that are annotated `payable` can receive quantities of ether; the ether is conceptually sent with the invocation, and the amount is stored in the variable `msg.value`. Each contract instance can own a quantity of ether; this quantity is automatically updated by the runtime when a function receives a payment. Every contract instance is stored on the blockchain at a particular address. The language has a built-in type called `address` to represent these addresses.

Restricting only functions annotated `payable` to receive ether may prevent some kinds of lost-ether bugs, in which clients accidentally send ether to the wrong function. However, this approach does not apply to other kinds of resources, and only covers *whether* the function can receive ether, not whether the function body accounts for it correctly. Likewise, although Solidity includes support for permission modifiers, such as `private`, to protect sensitive functions from being called externally, these modifiers do not prevent the bodies of methods from abusing assets or from being invoked when the receiving contract is in an inappropriate state.

Programs typically implement their own fine-grained accounting mechanism. For example, the `pendingReturns` structure records how much money is owed to each of a number of addresses. Without this mapping, although the contract would still record how much ether it held, the implementation would not be able to track for whom it is being kept.

The `pendingReturns` mapping supports the *withdrawal pattern* [Ethereum Foundation 2020d], which is an Ethereum coding convention that protects against re-entrancy attacks. The possibility of attacks arises because sending ether to a contract can cause the recipient to execute arbitrary code: the recipient can invoke a function on the sender, to which there is already an invocation on the stack. This is dangerous if the funds were sent while the sender was in an inconsistent state. Instead, it is recommended that contracts merely record that they owe ether to the intended recipient and provide a `withdraw` function that recipients can call to retrieve their money. The withdrawal pattern is not used in Obsidian, since Obsidian targets Hyperledger Fabric, which does not have this vulnerability. Because this was a summative study of the programming environments that users would encounter when using each language, we expected participants to use the withdrawal pattern with Solidity and not with Obsidian. This introduced the risk that the use of the withdrawal pattern would cause additional complexity relative to Obsidian.

In Solidity, using the withdrawal pattern typically results in the programmer writing code with arithmetic operations to update balances. In contrast, an Obsidian implementation of the withdrawal pattern would be expected to use linear assets, which the compiler could check for abuse — making the withdrawal pattern safer on Obsidian than Solidity. Although relevant to this study, if Obsidian were used with Ethereum, we expect Obsidian's asset-based approach would guard against bugs in using the withdrawal pattern, regardless whether the opportunity for it arises from the platform design or from the particular API being used.

4 STUDY DESIGN

Our experimental protocol was approved by our university IRB. The experiment began with obtaining informed consent. Participants (§ 5), whom we recruited from the university community, were randomly assigned to use either Obsidian or Solidity (using a block size of two) and given a tutorial (§ 6) on their assigned language, during which an experimenter answered any questions they had. Then, the study proceeded with three tasks: Auction (§ 8), Prescription (§ 9), and Casino (§ 10). After the tasks had ended, we gave participants a post-study survey (§ 11) asking for their opinions about the language they had used. We compensated participants with a $75 Amazon gift card.

Our participants had a variety of levels of programming skill and were new at programming in Obsidian and mostly new at programming in Solidity. To make it more likely that they would complete at least some tasks and to leverage the skills that participants would acquire by doing the tasks, we gave the tasks to the participants in order of increasing difficulty (according to our experience with pilot studies). All participants worked on the tasks in the same order.

Prior empirical work in programming language evaluation found that testing and debugging programs in the context of empirical studies substantially increases variance. For example, we previously found that different participants have different levels of thoroughness in writing tests and different levels of debugging skill [Coblenz et al. 2017]. When allowed to test and debug, participants frequently spend large amounts of time debugging issues that are not relevant to the experiment. To focus our participants' time on work related to our research questions, we allowed them to edit their code until they were satisfied, but did not give them an opportunity to test their code. Then, instead of assessing programs on the basis of tests, we inspected their code. We looked both for specific bugs that corresponded to the research questions we were interested in as well as for unrelated bugs. To do this, we developed a rubric for each task, which listed particular bugs to look for; we iterated on this rubric to add bugs that were present in the programs. This approach was made feasible by the relatively small codebases (see Table 11). Although it is possible that we missed particular bugs in the code, the same would be true even if we provided unit tests. However, by evaluating by inspection rather than unit tests, we were able to assess code with significant functional problems, similar to how one might grade a student exam by using a rubric rather than by executing unit tests. By using a rubric, partial credit can be awarded and specific mistakes can be identified even when the solution contains significant flaws.

Our decision to allow compilation but not testing was also motivated by our desire to evaluate the ability of Obsidian's type system to detect bugs that might otherwise be introduced. In many cases, it is cheaper to find bugs earlier in the software development process than later, so we focused on whether the kinds of bugs that Obsidian can detect at compile time are ones that are introduced at all and how successfully Obsidian programmers can complete tasks. In Delmolino et al. [2016], the programmers had access to a blockchain and still finished their work with code that lost assets; nonetheless, this does reflect a limitation in the study design. Because Obsidian's design is focused on identifying more bugs at *compile time*, allowing compilation but not testing allowed us to focus on our research questions about the usability and effectiveness of the *type system* while using our participants' time as effectively as possible. However, this may have introduced a kind of bias, since the Solidity participants might have found some of their bugs through testing if that had been permitted.

Another approach that could improve external validity is adding code review to the process. We did not include an explicit code review step in which peers inspected the code; although this might more-realistically represent some settings, studies have found that code reviews only find bugs infrequently [Czerwonka et al. 2015]. Also, adding code reviews to the process would have substantially increased time requirements as well as variance, since the quality of the feedback would necessarily have varied.

5 PARTICIPANTS

We recruited participants to our four-hour study with posters at our university, with emails to lists of appropriate degree programs (such as the Master of Software Engineering program), and by advertising to students in relevant courses (e.g., a software engineering course).

Because we advertised the study broadly (the posters were visible to anyone on campus), and because the study assumed that participants already knew an object-oriented programming language, we wanted to make sure that all participants had appropriate preparation. Therefore, we pre-screened participants with an online survey that asked them basic Java questions; we invited respondents who answered five of six questions correctly to participate in the study. The complete pre-screening instrument is in the supplement as well as in the replication package [Coblenz et al. 2020a]. The six questions concerned: Java constructor syntax; the definition of encapsulation; whether changes to a list through a reference would be visible through another reference; whether methods in interfaces may include bodies; whether abstract classes may be instantiated; and whether concrete subclasses of abstract classes must implement methods that were abstract in the superclass.

The same pre-screening instrument was used for earlier Obsidian studies in addition to this experiment. Over the period during which the instrument was used, 53 people completed it. Scores on the "basic Java" portion were out of six. Of the 53 respondents, seven scored below five, 13 scored five, and 33 scored six. Of those who participated in the RCT described in this paper, six scored five and the remaining participants scored six. Of the respondents who were not recruited in earlier studies, we contacted those with scores of at least five in the order in which they filled out the instrument. Those who responded to our requests were scheduled according to their and the experimenters' schedule requirements. No participants left the study early.

Table 1 summarizes the previous experience of the experiment participants in each condition. We excluded an additional participant who took so long on the training phase that not enough time was available for more than one programming task[1]. This left 20 participants; 14 of them identified as male, and six as female. Only two participants, both in the Obsidian condition, had no professional experience. Blockchains are targeted at enabling general software engineers in a wide variety of industries to build applications in contexts that lack mutual trust. Since our participants had substantial programming experience and 18 of them had some professional experience, we argue that our participants were representative of at least some kinds of programmers in industry who might be interested in writing smart contracts.

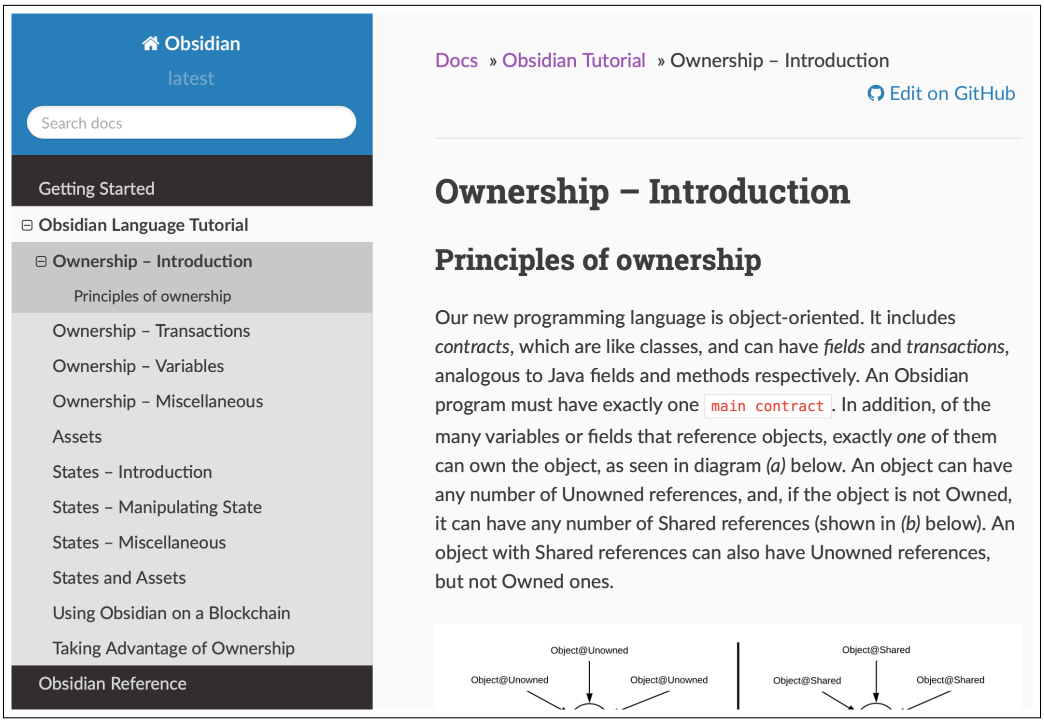
6 TRAINING

We provided a web-based tutorial (implemented with Qualtrics, and included in the supplement), which stepped participants through web-based documentation and exercises for their assigned language. Some of the exercises were to be completed in Visual Studio Code, which was configured with a compiler. Some of the questions were multiple-choice; for these, the tool automatically

Table 1. Participant experience (self-reported).

	Solidity ($N = 10$)	**Obsidian** ($N = 10$)
Median programming experience, years (range)	9.2 (3.3 – 13)	5.0 (2.3 – 8.5)
Median professional experience, years (range)	1 (0.5 – 9.0)	0.92 (0.0 – 5.0)
Median Java experience, years (range)	2.0 (0.67 – 4.2)	1.5 (0.25 – 6.0)

[1]That participant spent 3 hours and 11 minutes on the tutorial, which was three standard deviations above the mean.



Ownership – Introduction

Principles of ownership

Our new programming language is object-oriented. It includes *contracts*, which are like classes, and can have *fields* and *transactions*, analogous to Java fields and methods respectively. An Obsidian program must have exactly one `main contract`. In addition, of the many variables or fields that reference objects, exactly *one* of them can own the object, as seen in diagram *(a)* below. An object can have any number of Unowned references, and, if the object is not Owned, it can have any number of Shared references (shown in *(b)* below). An object with Shared references can also have Unowned references, but not Owned ones.

Fig. 3. An example from the web-based Obsidian tutorial.

showed participants if they had entered an incorrect answer. An experimenter was available to answer questions. Participants were told that they should try to get all of their questions addressed during the training phase, since no questions could be answered after training was completed.

We included practice questions in the tutorial to ensure that participants absorbed the material (prior studies had found that without practice questions, participants skimmed the material without mastering the concepts). Figure 1 shows an example practice question. Figure 3 shows a sample from the Obsidian documentation.

To make the two experimental conditions as similar as possible, even though Solidity includes no support for ownership, typestate, or assets, Solidity participants received a tutorial that explained these concepts and recommended using comment-based annotations. If participants asked about the utility of these annotations, we argued that this was similar to how one might write preconditions or postconditions in comments. Table 2 summarizes the distribution of times participants spent on the tutorial in the two conditions. If we had trained only the Obsidian participants in ownership, for example, any differences in behavior could have been attributed to the training and not to the language they were using.

Solidity participants received training on the *withdrawal pattern* and its necessity for security reasons, as well as language-specific details such as the `payable` keyword and the concept of *ether*.

7 TASK SELECTION

Based on our research questions, we sought a collection of tasks for our study that met several criteria. We selected criteria, shown in Table 3, to help us identify tasks that would address our research questions.

Table 2. Training times in Solidity and Obsidian conditions.

	Solidity ($N = 10$)	**Obsidian** ($N = 10$)
Average (standard deviation)	86 (28) min.	98 (31) min.
Range	39 to 138 min.	50 to 148 min.

Table 3. Alignment between task design criteria and the tasks we designed.

Name	**Criterion**	**RQs**	**Tasks**
C1	All tasks should reflect smart contract use cases that have received some attention in the blockchain community.	RQ1	Auction, Prescription, Casino
C2	At least one open-ended task that could be used to evaluate whether Obsidian participants could create their own typestate-oriented interfaces (rather than only implementing according to a well-defined specification).	RQ3	Casino
C3	At least one task should put Solidity participants in a position where they might accidentally lose an asset. For Obsidian participants, the corresponding question is whether they are able to complete the task in spite of the strictures of ownership.	RQ2, RQ3	Auction, Casino
C4	In a previous qualitative study of an early version of Obsidian, participants found it very challenging to use ownership ([Coblenz et al. 2019a]). Therefore, at least one task should assess whether the changes to Obsidian since its original design have addressed the difficulty of using ownership to restrict the use of assets.	RQ3	Prescription, Casino

The tasks and their results are described in detail in the following sections. Complete task materials are included in the supplement. Although we told participants that we might interrupt them eventually if they needed to move on to the next task, we reduced time pressure by not telling them specific per-task time limits.

8 AUCTION TASK

8.1 Auction Task Design

In the *Auction* task, we asked participants to fill in missing code in an implementation of an English auction, in which bids are made openly and the highest bidder wins. To increase external validity (criterion C1 in Table 3), we modeled the task after an example from a Solidity tutorial [Foundation 2020]. We required that all bids be accompanied by funds to ensure that the winning bidder will pay for the item. When a bid is exceeded, the original bidder should receive a refund of their bid. We gave the participants 30 minutes to complete the task.

Figure 2 shows the `bid` transaction that was provided to participants as well as a sample solution. In the first subtask, marked by `// 1. TODO`, participants needed to write code to refund the existing bid to the previous bidder, whose address was stored in `maxBidder`, and record the new bid (`money`). In the second subtask (`// 2. TODO`), participants needed to refund the bid to the bidder. The code in yellow shows a correct answer. In both cases, there was an opportunity for *asset loss*: if participants overwrote the old `Money` reference (stored in `maxBid`), then the old bid would be lost. In Obsidian, the compiler would report an error if this happened; in Solidity, there was no protection against that mistake. This opportunity for error reflected task criterion C3.

We refine the high-level research questions for this task (using numbering that reflects the original research questions in § 1):

RQ 2.1: How frequently do Solidity participants accidentally lose assets in the Auction task?
RQ 3.1: Overall, do completion times differ across conditions?
RQ 3.2: Overall, are participants more likely to finish Auction (and do so correctly) if they use Obsidian rather than Solidity?

8.2 Auction Results and Discussion

Table 4 summarizes the results of the Auction task; errors are shown in Table 5. Nine Solidity participants said they were done with the task before the 30 minutes expired; among these nine, the average time was 12 minutes (95% CI: [6.8, 17.4]). Eight Obsidian participants said they were done with the task before running out of time; among these eight, the average time was also 12 minutes (95% CI: [6.4, 18.3]). Thus, for **RQ 3.1**, the difference in times was not significant. Two participants completed the task correctly in the Solidity condition; seven completed the task correctly in the Obsidian condition. The difference in success rates (summarized in the first two rows of Table 4) is statistically significant, with $p \approx .015$ (Fisher's exact test; odds ratio 0.053). We conclude for **RQ 3.2** that participants who finished were more likely to finish correctly if they used Obsidian than if they used Solidity.

Of the two Obsidian participants who did *not* finish the Auction task in time, one (P45) was confused about the semantics of the `::` field initialization operator, attempting to use `BidsMade::maxBid` to refer to the current value of the `maxBid` field rather than the future value after a state transition. This misconception led to a compiler error message that the participant did not find helpful. The other participant also received a confusing error message: although the code invoked a transaction that did not exist, the error message pertained to ownership of the transaction's parameter. A more mature compiler with better error messages might have helped the participants finish the task.

In **subtask 1** (starting at line 31 in Figure 2), participants needed to record the new bid and refund the old bid. We found the following errors among the Solidity participants who said they were done:

(1) Loss of previous refunds: the correct implementation *added* the new refund to any prior refund. Four participants used = instead of +=, overwriting any old refund (in line 33).
(2) Omission of refund: three participants neglected to refund the previous bid (e.g., omitting line 33).

All eight of the Obsidian participants who said they were done did so without losing any assets, since otherwise the compiler would have given an error. However, one participant refunded the old money to the *new* bidder instead of to the *previous* bidder.

Table 4. Auction task results. N=10 in each condition.

	Solidity	**Obsidian**
Completed task correctly	2	7
Completed task with bugs	7	1
Time in min., completed tasks only; 95% CI	12; [6.8, 17.4]	12; [6.4, 18.3]
Did not complete the task	1	2

Table 5. Errors in Auction task. N=10 in each condition.

	Solidity	**Obsidian**
Ran out of time	1	2
Lost an asset in either subtask	7	0
Subtask 1		
omitted refund of old bid	3	0
overwrote old refund	4	0
refunded to wrong bidder	0	1
Subtask 2		
overwrote old refund	4	0
refunded via `transfer()` instead of `pendingReturns`	4	N/A

While doing the task, two of the Obsidian participants received a compiler error indicating that they had lost an asset. For example:

```
auction.obs 37.28: Variable 'maxBid' is an owning reference to an asset,
so it cannot be overwritten.
```

Both of these participants successfully fixed the error.

In **subtask 2** (starting at line 41 of Figure 2), participants needed to refund the new bid, since it was not larger than the previous bid. Among the nine Solidity participants who said they finished the task, two refunded the bid properly (using `pendingReturns`). Four refunded via `transfer`, which would not have resulted in asset loss but was inconsistent with the documentation we gave them. The documentation specified to use the withdrawal pattern to be consistent with the typical recommendation when using Solidity. Four attempted to refund via `pendingReturns` but, as in the first subtask, overwrote any previous refund, potentially losing money.

One might argue that the potential for asset loss due to improper use of `pendingReturns` was due to the need to use the *withdrawal pattern* [Ethereum Foundation 2020d], as discussed above. However, the particular bug we observed was due to participants overwriting an integer rather than adding to it, and we infer that arithmetic errors are likely common when manipulating assets manually. Obsidian protects against these bugs by encouraging programmers to design APIs that use assets to represent money rather than raw integers.

Of the seven Obsidian participants who completed the Auction task successfully, while they were working, two received compiler errors indicating that they had lost assets. One additional participant, P45, got an error about a lost asset, but did not finish the task because they were confused about the use of the `::` operator.

We conclude (**RQ 2.1**) that asset loss was frequent among Solidity users, and *more* frequent than among Obsidian users, who did not lose any assets ($p \approx .002$, Fisher's exact test). This difference may have been caused by a combination of differences in language design and differences in API design, but the different API designs were related to different language design choices.

9 PRESCRIPTION TASK

9.1 Prescription Task Design

To address task criterion C4, we gave participants a short `Pharmacy` contract (43 lines in Obsidian or 46 lines in Solidity including whitespace). The code included an example to show how the contract was vulnerable to attack. Although a `Prescription` was specified to only permit a fixed number of refills, a `Patient` could invoke `depositPrescription` on more than one `Pharmacy` object, resulting in the patient being able to refill the prescription the given number of times at *each* pharmacy. We asked participants to fix the bug, avoiding runtime checks if possible. In Solidity, for example, `depositPrescription` had the signature below:

```
function deposit(Prescription p) public returns (int);
```

In Obsidian, the starter code provided this signature:

```
transaction deposit(Prescription@Shared p) returns int;
```

In Obsidian, it sufficed for participants to change the signature so that the `Pharmacy` acquired ownership of the prescription object:

```
transaction deposit(Prescription@Owned >> Unowned p) returns int;
```

In Solidity, in contrast, since there is no static feature that would make the above safe, participants had to implement a global tracking mechanism across all `Pharmacy` objects.

The task was based on a task from our prior study, which found that users of an earlier version of Obsidian had great difficulty using ownership to fix this problem [Coblenz et al. 2019a]. For example, some participants in that study thought about ownership in a dynamic way (for example, writing `if` statements to test ownership) or were confused about when ownership was transferred between references. The version of the language used in the present study includes changes that resulted from that work, such as fusing typestate and ownership in the language syntax, making ownership transfer explicit in transaction signatures, and removing local variable ownership annotations. Our research questions were centered around evaluating the revised language:

RQ 3.3: What fraction of Obsidian participants could use ownership to fix the multiple-deposit vulnerability?

RQ 3.4: Does using ownership to prevent the multiple-deposit vulnerability take less time than using a traditional dynamic approach?

We gave participants 35 minutes to complete the task. Because ownership-based approaches are not checked by the Solidity compiler, we wanted to allow Solidity participants who proposed ownership approaches to try again. Therefore, when participants informed the experimenter that they were done, the experimenter inspected their code. If they had used static notions of ownership instead of a dynamic approach, participants were permitted to try again in the rest of their 35 minutes.

9.2 Prescription Results

Table 6 summarizes the results. Regarding **RQ 3.3**, six of the ten Obsidian participants successfully used ownership to solve the problem. The dynamic Obsidian solution that we judged to be correct tracked global state by making `Prescription` mutable, despite a comment indicating that `Prescription` should be immutable.

Five of the ten Solidity participants tried to use ownership, even though Solidity does not check ownership. Only three of the Solidity participants said that they were done within the time limit, and of those, only two had a correct solution. The incorrect Solidity solution attempted to solve the problem by making `Prescription` mutable to track remaining refills globally, but in addition, although the participant tried to track the number of refills across all pharmacies, the code did not update the global number of refills when refilling a prescription.

We separated time Solidity participants spent on static solutions from the time spent on dynamic solutions, since these attempts were likely prompted by the training materials; Table 6 shows both sets of times. However, we included all time spent by Obsidian participants on correct solutions because it is realistic that some Obsidian users would have tried each approach.

Regarding **RQ 3.4**, we did not observe a significant difference in completion times. However, a large fraction of Solidity participants spent significant portions of their time attempting static solutions. In retrospect, it might have been more informative to have told Solidity participants explicitly that they needed to use a dynamic solution, and this approach might have led to a more interesting comparison. However, due to the small amount of code required for the static solution, we suspect that there is a significant learning effect to be leveraged here in Obsidian, and the participants who succeeded could likely do so again in a similar situation much faster. Furthermore, the fact that only two of three Solidity participants who finished this task did so correctly underscores the benefit of static enforcement of these kinds of safety properties.

9.3 Prescription Discussion

One might have expected that applying ownership would be challenging, since the concept was new to the participants, but since only half of those who said they had completed a dynamic solution had correct solutions, it would appear that using the new static construct may not be harder than writing global state tracking code. In fact, using ownership to solve programming problems is *teachable*: six of nine Obsidian participants who completed the task used ownership to do so. We expect that the remaining three could be taught to do so with additional practice.

In an earlier study [Coblenz et al. 2019a], which used a very similar task, four of six participants were able to solve the problem, but all of them received significant help from the experimenter. The remaining two did not solve the problem even with help. Although those numbers cannot be compared with the results of this study directly, the fact that six of our participants were able to complete the task without any help suggests that the changes since then have improved usability.

Security experts have long argued in favor of immutable data structures [Oracle Corp. 2019; Seacord 2013], which is one reason why we specified that `Prescription` was immutable. However, these results point out that this approach may not be tenable: specifications of immutability may be ignored or removed, and attempts to maintain immutability require substantial work, which may itself be bug-prone. Indeed, when language-based mechanisms do not provide the required safety properties [Coblenz et al. 2017], it may be safer and cheaper to use a *mutable* design than to bear the cost of immutability. In this case, making `Prescription` mutable would have obviated the need to implement separate data structures keeping track of how many refills each prescription

Table 6. Summary of Prescription task results. Times are shown as mean (standard deviation). N=10 in each condition. Two Solidity participants tried both static and dynamic approaches, and one Solidity participant made no changes, resulting in 11 Solidity attempts.

	Solidity	**Obsidian**
Attempted a static solution	5 participants	6 participants
Correct static solution	N/A	6
Attempted a dynamic solution	6	3
Correct dynamic solution	2	1
Made `Prescription` mutable	2	1
Completed within time limit	3	9
Mean time among successful participants; [95% CI]	20 min.; [0, 45]	22 min. [12, 33]
Mean time among successful participants after removing Solidity time spent on static attempts	18 min.	22 min.

had left, since that information could be kept in the `Prescription` object directly. That approach appeared to be more natural for the participants, and certainly required less code.

The benefits of Obsidian's ownership system in the Prescription task contrast with the benefits of other kinds of ownership. For example, ownership in Rust [Mozilla Research 2015], which is understood to be one of the more challenging aspects of learning Rust [Yegulalp 2018], introduces constraints on mutation. However, in Rust, only owning references can mutate objects. In Obsidian, mutation is restricted only as much as is needed to provide sound typestate specifications, since concurrency is not a concern. Therefore, in Obsidian, only changing *which named state an object is in* is restricted, and then only through `Unowned` references. This contrasts with Rust, in which *all* modifications to fields of objects are restricted. Six of 10 participants were able to use the linear aspects of ownership alone in the Prescription task, suggesting that languages that adopt just linearity (and not mutability restrictions) may be usable. Perhaps by integrating a more flexible permissions system, languages such as Rust could be made more convenient for common cases, and thus have a more gradual learning curve. Alternatively, making these changes in Rust might come at the expense of expert efficacy, so the impact of these changes in the long term would need to be evaluated.

Although six of the Obsidian participants used ownership successfully, four participants did not. One of the four participants did not complete any of the three tasks. Another "fixed" the issue by modifying code in `Patient` that we had provided as an example of how a nefarious patient might exploit the bug; perhaps this participant did not really understand the task. The other two seemed to need more time studying ownership in order to use it effectively.

The instructions included: "Please use what you have learned today to fix this problem (avoiding runtime checks if possible)." Perhaps as a result, a similar fraction of participants tried to use ownership in both conditions. We wanted to encourage the Obsidian participants to use ownership so that we could assess to what extent they could use it effectively, but the instructions persuaded some of the Solidity participants to use it even though doing so was not checked by the compiler.

10 CASINO TASK

10.1 Casino Task Design

The casino task was more open-ended than the other tasks, addressing criterion C2. We gave participants a web page with a diagram showing invocations that needed to be supported (Figure 4). The web page included a list of requirements:

(1) If a `Bettor` predicts the outcome correctly, the `Bettor` gets twice the `Money` they put down. For example, if `Bettor` b puts down 5 tokens on the correct outcome, they should receive 10 tokens after the Game is played.
(2) If the `Bettor` predicted incorrectly, the `Casino` keeps their tokens.
(3) Bets can only be made before the `Game` starts.
(4) Winnings can only be distributed after the `Game` is finished.
(5) `Bettors` must collect winnings themselves from the `Casino` after a Game by calling code, which you need to write. Until winnings are collected, the `Casino` keeps track of them.
(6) A `Bettor` can have one active bet per game. If a `Bettor` bets more than once, their original bet should be replaced by the new one and any previous bet should be refunded.
(7) A `Bettor` MUST put down tokens at the same time that they're making a Bet.
(8) If the `Casino` does not have enough tokens available to pay out winnings, the invocation to collect winnings can fail.

We provided starter code for `Casino`, `Game`, and `Bet`. Obsidian participants also received implementations of appropriate containers (Solidity has suitable built-in containers).

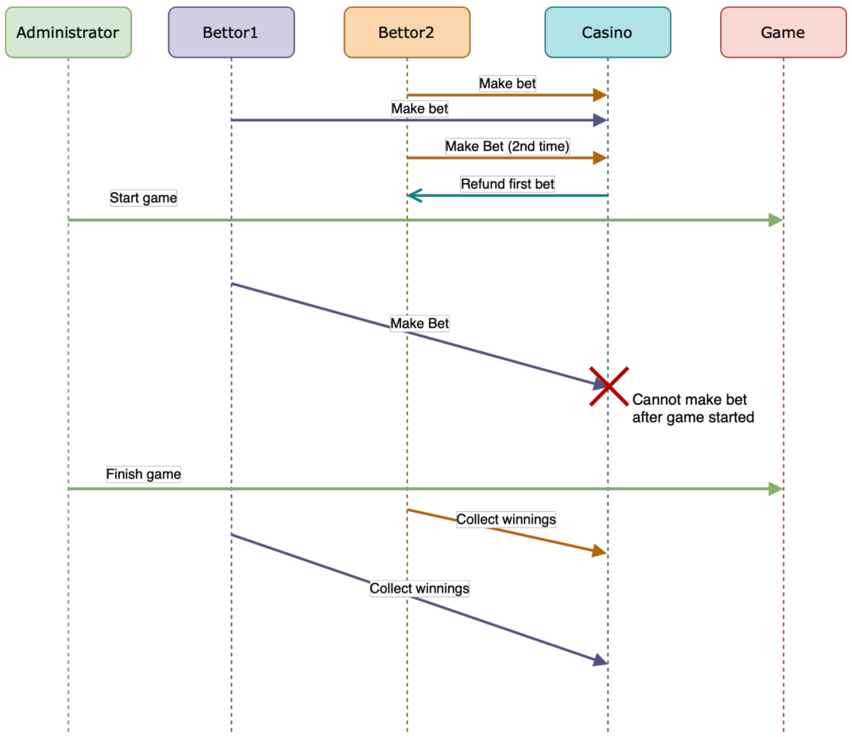


Fig. 4. Sequence diagram given to participants to show what operations the Casino contract should support.

We used the Casino task to investigate five research questions:

RQ 2.2: How frequently do Solidity participants lose assets in the Casino task?

RQ 3.5: To what extent do Obsidian participants leverage typestate in transaction signatures to avoid dynamic checks (criterion C2)?

RQ 3.6: In both versions, programs represented funds with Token objects. Does Obsidian's type system help participants avoid losing Token objects compared with Solidity (criterion C3)?

RQ 3.7: Do Obsidian participants view Token objects as resources that should not be created or destroyed, or as data, which could be created and destroyed as needed (criterion C4)?

RQ 3.8: How do task completion times compare between Solidity and Obsidian participants (criterion C4)?

Of the original ten participants in each condition, we excluded one Obsidian participant who should have received an error from the compiler but, due to a bug, did not. We also excluded one Solidity participant and four Obsidian participants who did not have enough of their four hours remaining, and in the time available, were not satisfied with their solution. This left nine Solidity participants and five Obsidian participants whose results we analyzed. The results below pertain to these 14 participants. The discrepancy between the number of participants who had enough time across the two conditions was due to two factors: the faster task completion times by Solidity participants, and the high variance among Obsidian participants for time required for tasks that occurred *before* Casino.

Of the 14 participants whose results we analyzed, one participant in the Obsidian condition gave up after 1 hour, 15 minutes. That participant had chosen an unnecessarily difficult implementation

Table 7. Summary of Casino task completions.

	Solidity	**Obsidian**
Had enough time to try Casino	9	5
Completed Casino with a program that compiled	8	4

Table 8. Summary of Casino task results among completed programs that compiled, showing correct solution rates among errors made by more than one participant.

	Solidity ($N = 8$)	**Obsidian** ($N = 4$)
Completed task correctly (no identified bugs)	12.5%	0%
Winnings collection emits error if Casino is out of tokens	62.5%	25%
Casino keeps tokens when a bet is lost	100%	50%
Bettor's extra bets result in refunds	100%	75%
Only used disown safely	N/A	0%
Managed tokens correctly (not fabricating or losing them)	50%	0%
Mean completion time	37 min.	64 min.

strategy, requiring implementing a new container (implemented as a linked list). Also, the participant delayed trying to compile until after writing a lot of code, resulting in a large collection of compiler errors. The remaining four Obsidian participants all wrote code that compiled successfully. One participant in the Solidity condition gave up after 39 minutes, having received a parser error that they were not sure how to fix.

The Casino task results should be interpreted in the context of some additional limitations relative to the other tasks. The results of a comparison between conditions on this task may be biased because the Obsidian participants who were included for analysis in this task are those who completed the earlier tasks fastest. As a result, the Obsidian participants may have been stronger programmers on average than those in the Solidity condition (in which almost all the participants had time to try the task).

Casino was more open-ended than the other tasks, resulting in more variance, as participants made varying implementation choices. Given this, the small numbers, and the potential bias, we use this primarily as an opportunity to develop hypotheses, design insights, and identify future opportunities for improvement.

10.2 Casino Results and Discussion

We analyzed the code participants wrote, comparing the code to the requirements we gave them. As with the other tasks, in order to isolate the bugs from each other, our analysis was manual; the source code produced by each participant is included in the supplement. Results are summarized in Tables 7 and 8. Before discussing the results for the four research questions, we discuss the errors that participants made while working on Casino.

Except for the one Solidity participant who completed the task correctly, all of the other participants across both conditions inserted various bugs. For example, three participants in each condition failed to emit an error when attempting to collect winnings from the Casino when the Casino lacked enough resources to distribute them.

The most common bug among Solidity participants was some kind of token loss or improper fabrication; four of the eight made this mistake, addressing **RQ 2.2**. For example, one participant, when accepting a new bet, first credited the bettor's account for any prior bet, and then returned if the bet amount exceeded the bettor's balance. This meant that a second bet, when disallowed, would incorrectly fabricate tokens out of thin air, since the code that debited the bettor's balance occurred after the early `return`. Another participant neglected to debit accounts for extra wagers, also fabricating tokens out of thin air. This shows that protection against incorrect asset manipulation *is* important.

All four Obsidian participants who finished the Casino task used `disown` improperly to throw away assets. We found this surprising, since we had warned the participants against improper use of `disown`. The tutorial included an example of how `disown` might be needed inside the implementation of a `Money` contract, and wrote below the example:

> IMPORTANT: `disown` should be used only when you really want to throw something out. Above, `disown` is required because of the manual arithmetic used to manipulate `amount` inside the implementation of `Money`, but it is not needed in most normal code.

It would not have sufficed to remove `disown` from the language; in addition to the fact that it is needed in certain (rare) cases, programmers could build a `Trash` contract to hold discarded objects, thus suppressing any errors the compiler would emit. We have several hypotheses regarding why `disown` was abused:

- Some participants may have used `disown` to silence the compiler when they felt they had a correct solution. One participant discarded the old wager, using money from the casino's pot to pay out bets. The participant also disowned the bet when a losing bettor tried to retrieve their winnings (a correct solution would have put the tokens in the casino's pot).
- Some participants may not have read or understood the tutorial's warning about `disown`; in retrospect, we should have assessed understanding explicitly.
- Some participants did not sufficiently understand the notion of assets. For example, when disbursing winnings, one solution disowned the previous wager and created new `Tokens` when needed, rather than reusing the tokens from the wager.
- Some participants may have used `disown` as a workaround for an unsolved problem. In code to accept a new bet, one participant disowned any previous bet by that bettor, and then wrote: `// Currently just throws the bettor's money away and hopes they find it eventually.`

This motivates a question for future research to characterize the use of these *escape hatch* constructs, which can be used in both safe and dangerous ways. Some languages include warnings in the names of such constructs, as in `unsafePerformIO` in Haskell [of Glasgow 2001], but such approaches may not be effective in explaining the danger.

Risk compensation refers to the idea that people compensate for safety features by taking additional risks. For example, drivers may drive faster when wearing seat belts because the seat belts may mitigate the risk caused by higher speeds. However, in many cases (such as in the seat belt example), overall risk is reduced despite potential compensating behavior [Houston and Richardson 2007]. Further study is needed to consider the question of whether risk compensation occurs with strong type systems. Perhaps some participants who used `disown` assumed that if there were a bug, the compiler would report it.

Regarding **RQ 3.5**, the `Game` contract that we provided defined states for `Game` (`BeforePlay`, `Playing`, `FinishedPlaying`). When implementing `Casino`, participants wrote code in their `Casino` transactions (to implement requirements 3 and 4) to dynamically check the state of the `Game`. An alternative approach, which we had expected some participants to use, would have involved observing that the state of Game was related to the state of Casino. If a Casino was in a state that permitted bets, then the Game must not have started. Likewise, collecting winnings was only possible from a Casino whose Game had ended. Our starter code defined states in Game. Therefore, participants could have avoided dynamic checks in Casino by defining multiple states in Casino, each of which corresponded to a particular state of Game. All of the participants added the dynamic checks. The participant who gave up tried to check the state with a static assertion, evidently not understanding that the assertion was static, not dynamic.

The lack of usage of static state information is unfortunate because it represents a missed opportunity to rule out bugs in calling code. Perhaps more-experienced programmers would be more interested in leveraging this language feature; alternatively, it might require more training or a more-convenient language design. The design the participants chose may have been best given their incentives; the typestate-based approach would have required adding more structure to reflect the typestate relationships between Casino and Game. However, motivated by the results of this study, we hope to consider future language design changes that make typestate coupling of different objects convenient.

Perhaps *creating* new interfaces that use novel verification-related features (such as typestate) is harder than *consuming* them, in which case further research should consider novel ways of scaffolding interface design and creation. In Obsidian, it is possible to use states to define different fields (each with its own typestate specification) for each possible state of referenced objects. This approach would couple the states of the two objects, letting the programmer avoid runtime checks that would be otherwise required, but we did not train participants in this approach. Perhaps this technique is sufficiently subtle that we should have provided training directly.

Surprisingly, in **RQ 3.6**, we observe that in fact, Solidity participants were probably more likely to correctly have the casino keep tokens when a bet is lost (Table 8, $p \approx 0.09$, Fisher's exact test). Likewise, Solidity participants may be more likely to successfully issue refunds for bets after the first bet ($p \approx 0.33$, Fisher's exact test). We believe this is related to abuse of disown by Obsidian participants.

Regarding **RQ 3.7**, all four of the Obsidian participants who wrote solutions that compiled treated tokens as data rather than assets, i.e., at some point in their code, they either created new tokens or disowned tokens. The Obsidian participant who gave up did not do either of those things, likely viewing tokens more as assets, but became mired in a list of type errors. We conclude that use of assets was likely *not* natural for our participants. This interpretation is consistent with the post-study survey results (§ 11), in which we observed that participants said they felt ownership and states were more useful than assets.

For **RQ 3.8**, Obsidian participants spent significantly longer on the Casino task than Solidity participants did ($p \approx 0.02$, Mann-Whitney U test, $d \approx 1.9$). Therefore, the stronger type system provided by Obsidian likely had a significant cost in development time for our participants. We hypothesize that this cost is greater with more open-ended tasks, which would explain why we did not observe this difference in the two prior tasks. Of course, the additional cost may be worthwhile since Obsidian would rule out some classes of bugs statically, particularly when used by skilled programmers who do not abuse disown. The time to task completion in this experiment might relate in practice to the time required to complete a *prototype* version of software rather than the time required to create a production-quality version.

11 POST-STUDY SURVEY RESULTS

We conducted a post-study survey asking participants about their opinions. We asked participants (on a 5-point scale) how well they understood particular concepts and how useful they thought those concepts were. The results from participants who completed the survey *before* being de-briefed

Table 9. Perceptions of ownership, states, and assets on a 1–5 scale (5 is best). Cells show average (standard deviation). * indicates that a Mann-Whitney U test shows a significant difference at $p < 0.05$.

	Solidity (N=6)	**Obsidian (N=8)**
How much did you like the language you used?	3.7 (0.82)	4.0 (0.53)
How well do you feel you understand the concept of ownership?	3.8 (0.98)	3.75 (0.99)
*How useful do you think ownership is?	3.0 (1.1)	4.88 (0.36)
*How well do you feel you understand the concept of states?	4.8 (0.41)	4.1 (0.64)
How useful do you think states are?	4.3 (0.81)	4.1 (0.64)
How well do you feel you understand the concept of assets?	3.2 (0.98)	3.4 (1.3)
How useful do you think assets are?	2.7 (0.52)	3 (1.2)

from the study are summarized in Table 9. Although the Obsidian participants said they thought ownership was significantly more useful than the Solidity participants did ($p \approx 0.002$, $d \approx 2.5$), the Solidity participants indicated that they felt they understood states better than the Obsidian participants did ($p \approx 0.04$, $d \approx 1.3$). Perhaps the existence of an unfamiliar `state` construct in Obsidian, or the unfamiliarity of the relationship between states and types, led to less confidence. There was no significant difference in views of the utility of states. This may be due to the tasks we gave, which did not particularly rely on the static aspects of states.

We also compared across questions. Obsidian participants said that they felt both ownership and states were *more useful* than assets ($p \approx .0027$, $d \approx 2.1$ and $p \approx .037$, $d \approx 1.2$, respectively, according to a Mann-Whitney U test and Cohen's d). The differences between perceptions of understanding between ownership and assets and between states and assets were not significant at $p < .05$. Of course, perceptions of understand and utility were influenced by the particular tasks the participants did and their perceived success at doing those tasks, but these results correspond with the Obsidian participants' failure to regard tokens as assets in the Casino task.

The survey also included a free-form box in which participants could respond to the question "Please use this space to write any additional comments you have about the language or the study." We reviewed the free-form comments, and describe the most interesting ones here. Due to the sparse and brief nature of the data, we did not conduct a more formal analysis. The complete data are included in the supplement.

Several of the Solidity participants expected that the language would have included constructs for states or ownership. For example, one participant wrote in the post-study survey:

> It also seemed like there should be some syntactic sugar for writing things like:
>
> ```
> enum State { Foo, Bar, Buzz }
> State s
> ```
>
> since they are so common.

Three Solidity participants expected that the compiler would check ownership. For example:

> On [semantics] — I was hoping ownership / assets / states would be statically verified. When I wrote code during the 2nd phase of the study, I found that I didn't really document ownerships / asset status.

Similarly, from another Solidity participant:

Table 10. Numbers of participants completing different numbers of tasks correctly in the two conditions.

Tasks completed correctly	**Solidity participants**	**Obsidian participants**
0	7	1
1	2	3
2	0	6
3	1	0

> I think it would be nice to have some static analysis to check ownership information rather than relying on the programmer to have good comments documenting ownership because in practice documentation is never perfect and often overlooked.

Participants using Obsidian had differing opinions regarding how easy it was to learn. One wrote:

> The tutorial and the exercises are well-written and they helped me a lot in understanding the concepts of new language!

Another Obsidian participant wrote:

> The smaller coding exercises were nice to follow and complete. The open-ended part was a little overwhelming to finish, for somebody that just got introduced to new concepts of ownership, states and assets.

One Obsidian participant commented on how ownership seemed natural after some practice:

> . . . the general concepts of ownership [were] a little unintuitive but after working with the language they started to make more sense and seem more natural. . . .

12 SUMMARY OF RESULTS

In order to summarize the overall results across the three tasks, we computed the fraction of successfully completed tasks for each participant, taking into consideration the tasks that were not included in the analysis above (only nine Solidity participants and five Obsidian participants in the Casino task). The median fraction of tasks completed by Solidity participants was 0%; the median fraction of tasks completed by Obsidian participants was 67% (since these are ratios, we do not take their mean). We compared these fractions using a Mann-Whitney U test, observing a significant difference with $p < 0.008$. Table 10 summarizes *successful* task completions: tasks that were completed correctly within time limits.

We now return to the initial research questions.

> RQ1: Could we obtain actionable data about the usability of a novel programming language (that uses a type system that would be unfamiliar to our participants) in a short-duration user study (less than one day), which would be representative of real-world smart contract development?

Our results regarding the other research questions show that we were able to identify both strengths and weaknesses of Obsidian relative to Solidity in the context of smart contract programming tasks that were drawn from real-world scenarios. Perhaps more importantly, the study led to insights about the type system and its potential risks that can be used to refine future language designs. For example, strong type systems can be learned and used to significant benefit in short periods of time, but escape hatches can be frequently abused.

> RQ2: Do programmers using Solidity insert more of the kinds of bugs that Obsidian is designed to catch?

In Auction, we observed that Solidity participants lost assets frequently (seven of nine Solidity participants who finished lost assets) (RQ 2.1). Similarly, four of eight Solidity participants lost assets in the Casino task (RQ 2.2). We conclude that linear type systems likely have value in helping programmers of smart contracts detect bugs earlier. However, the abuse of `disown` in Casino shows that training or language refinement may be needed to help users of linear type systems obtain these guarantees effectively.

> RQ3: Can programmers who were previously familiar with object-oriented programming (but not with Obsidian, typestate, ownership, or linear type systems in general) successfully use Obsidian to complete relevant smart contract programming tasks? If so, is there a significant impact on task completion times?

In Auction, we observed no significant difference in completion times, which is promising for strong type systems (RQ 3.1). Furthermore, participants who used Obsidian were more likely to finish Auction correctly (RQ 3.2). In Prescription, 60% of Obsidian users were able to use ownership to fix the vulnerability we gave them, suggesting that earlier work iterating on the language design using user-centered methods may have been effective in making the language more usable than it had been initially (RQ 3.3). Using ownership does not necessarily take less time than using a dynamic approach, but seven of the ten Obsidian participants successfully completed the Prescription task, suggesting that enabling use of ownership can empower programmers to be more effective (compared to two of ten Solidity participants who succeeded) (RQ 3.4). In Casino, we observed that all of the Obsidian participants who finished the task abused the `disown` keyword, which serves as a caution that escape hatches from safety features are easily misused; this misuse can significantly hamper a language's ability to achieve its safety goals (RQ 3.5). Likely due to this misuse, Solidity participants were more likely than Obsidian participants to have the Casino keep funds from losing bets and to issue refunds correctly for revised bets. Likewise, the Obsidian participants appeared to create and destroy tokens at will, rather than treating them as assets; more training is likely required if we want programmers to achieve the safety benefits of adopting this perspective, which may not be a particularly natural one for them. Finally, Obsidian participants spent significantly longer on the task than the Solidity participants did, confirming that in some more complicated tasks, a stronger type system likely increases the cost of an initial implementation.

There has been long-standing debate about a hypothesis that dynamically-typed languages may be better for prototyping work than statically-typed languages [Hanenberg et al. 2014; Meijer and Drayton 2004]. Our results provide a limited form of support for this hypothesis, since the Solidity participants were able to finish the Casino task faster than the Obsidian participants. Of course, since there were many differences other than type system between the two languages, the study does not address this hypothesis directly.

13 LIMITATIONS AND THREATS TO VALIDITY

The pre-screening instrument may have introduced bias, either by being unrepresentative of common content knowledge among typical object-oriented programmers or by discouraging some people from participating. The student participants may not be representative of the population of smart contract programmers. However, since most of the students had some professional experience, they were likely representative of entry-level programmers in industry [Stack Overflow 2019]. Also, other studies have found no significant differences in code correctness in programming studies between students and non-students [Acar et al. 2017]. The tasks were more constrained than real-world programming tasks, although smart contracts tend to be small in practice, averaging 322 lines [Pinna et al. 2019]. Table 11 describes solution lengths in our study.

The participants were new to the programming languages and to smart contract development in general, so it is possible that experienced programmers would have behaved differently. Although we tried to infer how particular aspects of the languages and their type systems affected participants' behavior and performance, because this was a summative study, the results may have been influenced other aspects of the experience, such as the way in which different aspects of the type systems interacted or the details of the particular tasks we gave participants.

The order of the tasks was consistent among all the participants, so the results of the tasks cannot be regarded as being independent of each other due to learning effects. Because of the way we allocated time, only five Obsidian participants completed the Casino task, so those results should be considered exploratory.

We assessed results by manually analyzing code for correctness rather than by test cases. It is possible that using unit tests would have revealed additional bugs that we did not identify. Furthermore, disallowing execution might have biased the results to favor Obsidian, since without testing, writing correct code may be easier with the stronger type system that Obsidian provides.

Providing unit tests to participants (and allowing them to be run) might have allowed the participants to identify minor bugs that had gone undetected without incurring the time and variance cost of having participants write their own tests. This approach would also likely have increased external validity, although it adds a concern that participants might simply iterate until the tests pass rather than ensuring that they really understand how to write the code correctly. Furthermore, given our research focus on *static* correctness techniques, it is not clear which unit tests should have been included.

Our study was potentially subject to the Hawthorne effect, in which participants may change their behavior when they know they are being observed [Sadler and Kitchenham 1996]. However, this effect would have been equally present in both conditions. To minimize this effect, although the experimenter remained in the room, the experimenter minimized direct observation by recording screen videos and source code rather than continuously watching participants.

14 IMPLICATIONS ON PLIERS

PLIERS is a process for language design that integrates user-centered methods into the design and evaluation process for programming languages [Coblenz et al. 2019a]. In addition to evaluating Obsidian, the study also served in part to evaluate the PLIERS process. We were able to show a safety benefit of Obsidian in the Auction task, and were able to show that most of the Obsidian participants were able to use ownership successfully in the Prescription task. This shows that the tutorial method was mostly successful (though more success could likely have been obtained with more practice) and that the language design was effective overall (modulo the abuse of `disown` that we observed). Every study design involves making tradeoffs. The results here may show a tradeoff between training time and success rates; users of PLIERS will need to decide, based on their own design and research goals, how to balance the risks when designing their studies. However, the

Table 11. Ranges of all (whether correct or not) solution lengths in lines of code.

	Solidity		**Obsidian**	
	Min	Max	Min	Max
Auction	291	355	352	395
Prescription	455	570	457	518
Casino	257	425	135	416

overall PLIERS design process did result in a language that had significant benefits relative to the status quo, which we were able to measure in a relatively low-cost study.

In retrospect, since only one of 20 participants completed the Casino task successfully (across both conditions), that task was too hard for the amount of time we allowed. We recommend that users of PLIERS carefully select success criteria in pilot studies in order to set appropriate task time limits and difficulties.

Designing and executing an effective RCT is extremely challenging and labor-intensive. The tutorial and tasks were initially drafted much earlier than the RCT, but even with a six-participant usability study (which had nine pilots) and four pilots for the RCT, we were still surprised at how difficult the Casino task was for some participants. In addition, executing the RCT was a lengthy effort; we estimate that recruiting participants, managing 21 participants plus four pilots, analyzing the data, etc. took about two months.

15 RELATED WORK

Rust [Developers 2017] supports a version of ownership. Permissions and typestate features have not been included in popular programming languages, so we lack data regarding their broader usability. We are not aware of prior quantitative user studies of any of these kinds of type systems.

Other programming languages were designed to improve smart contract safety. Flint [Schrans et al. 2019] is a typestate-oriented language that supports linear assets, but omits a permissions system for flexible referencing of objects. Pact [Kadena 2019] is Turing-incomplete, avoiding nonterminating behavior. Scilla [Sergey et al. 2019] is an intermediate language whose semantics were formalized in Coq; it represents programs as communicating automata, avoiding complex inter-contract transactions (instead requiring that these be implemented as continuations). Nomos [Das et al. 2019] uses linear types for safety and provides automatic resource analysis to facilitate gas usage prediction, but does not operate on any blockchain platforms. None of the above languages were evaluated empirically.

Delmolino et al. [2016] described a user study of Serpent, which was a precursor to Solidity, showing several classes of bugs that occurred in the lab. Among these was *asset loss*, which motivated our study to see whether our participants would avoid these bugs when using Obsidian.

In the past, we argued for using a variety of methods when designing programming languages [Coblenz et al. 2018]. Stefik and Hanenberg [2014] focused on the need for empirical evaluation of programming languages. Stefik et al. developed methodology to evaluate syntax choices for novice programmers [Stefik and Siebert 2013]. Hanenberg et al. compared static and dynamic type systems [Hanenberg et al. 2014], finding benefits of static type systems in both documentation and compile-time checking. The general question of the benefits of the Obsidian type system relates to the more general question of static vs. dynamic types, which has also been addressed in other contexts [Hanenberg et al. 2014]. Our evaluation approach is related to that used by Stefik, Hanenberg, and others, but our method integrates teaching a novel type system into the experimental procedure. Also, our study was between-subjects, whereas the experiment comparing static to dynamic types was within-subjects [Hanenberg et al. 2014]. Our design is robust to learning effects, but more subject to variance among participants. In comparison to the Hanenberg et al. paper, we focus more on an experimental design that minimizes time required by participants and on reporting detailed data on errors made by participants.

Uesbeck et al. evaluated the benefit of C++ lambdas [Uesbeck et al. 2016], finding that no benefit even for the purposes for which lambdas were created. The only other quantitative empirical studies we are aware of for complete, novel programming languages (as opposed to ones that were already familiar to the participants) were of Quorum [Stefik and Siebert 2013] and HANDS [Pane et al. 2002]. Other work has focused on language extensions, such as for software transactional memory [Pankratius and Adl-Tabatabai 2014] and immutability [Coblenz et al. 2017].

Several languages have integrated support for linearity. Wadler [Wadler 1990] proposed the use of linear types for programming languages. In *functional* languages, linearity may take the form of session types [Caires and Pfenning 2010]. This approach mirrors typestate, since channel types (expressed as session types) change as messages are sent through them. Typestate was originally proposed by Strom and Yemini [Strom and Yemini 1986]. Drossopoulou et al. [2002] introduced typestate for object-oriented languages in Fickle. In Fickle, objects could dynamically change class, but the static type did not reflect the changing classes. More recent work by DeLine, Aldrich, Bierhoff, and others describes how object-oriented languages can be used with typestate-specifying references [Aldrich et al. 2009; Bierhoff and Aldrich 2007; DeLine and Fähndrich 2004]. Garcia et al. [2014] gave a formalization of typestate. None of the these languages were empirically evaluated with users, although Sunshine et al. [2014] showed that typestate in documentation can be beneficial. We designed Obsidian [Coblenz et al. 2020b] using user-centered design [Coblenz et al. 2019a]; that paper focuses on the design methodology, rather than on an empirical comparison between Obsidian and Solidity, but it describes a partial, previous version of the experimental materials that we used here.

High-quality error messages are a key component of a usable compiler. Some work in this area has produced guidelines for creating error messages that are effective for novices [Becker et al. 2019]. Our work focuses on professionals, but the guidelines given in that work (for example, *show solutions or hints*) are relevant to Obsidian.

16 FUTURE WORK

Casino, which included significant use of nominal states (i.e., statically-defined states with their own names), resulted in Obsidian participants writing dynamic checks. Future work should investigate the extent to which typestate, as provided in Obsidian and other typestate-oriented languages, can be made more compelling for programmers. For example, when pairs of objects have states that are coupled, the language could provide features to make representing and using this relationship convenient. Likewise, the existence of risk compensation among programmers should be investigated in future studies.

This study investigated whether Solidity programmers insert bugs that Obsidian detects, but a corpus study could show how prevalent these kinds of bugs are in real-world Solidity code. Bugs involving asset loss can occur in any language in which programs may manipulate assets; a corpus study involving a larger corpus of programs might be fruitful and still generalize to smart contract development. Indeed, it may be that the kinds of safety properties needed in smart contract development are generally applicable to many kinds of programs, regardless of whether they are hosted on blockchains.

A study that included testing, debugging, and code review would be more representative of real-world use. Also, the participants were new to the language they used in the study, and the Casino task may have been too difficult for the amount of time we allocated. A longer-duration study would likely have increased validity — both by allowing enough time for difficult tasks and by allowing participants to become more comfortable with the language they were using. In a study of experienced Obsidian and Solidity programmers, we hypothesize that the task completion time difference would diminish significantly but Obsidian users would continue to take longer on complex development projects. On the other hand, there would likely be fewer serious bugs in the completed Obsidian projects, since Obsidian rules out classes of important bugs.

The tradeoff between internal and external validity seems fundamental to programming studies. Above, we suggested increasing external validity, but doing so might compromise internal validity further, since the tasks would include more kinds of work. Another approach to study design might trade external validity for additional internal validity. Asking participants to write nontrivial programs, such as in the Casino task, may reflect real-world programming tasks, but the task complexity results in high variance and significant amounts of time spent on programming issues that are not necessarily directly related to the research questions. In future studies, it might be worthwhile to consider more restricted tasks that only investigate a small portion of the development workflow. For example, if the research question pertains to creation of interfaces or architectures, then the task might isolate that part of the programming process rather than integrating it into a complete programming problem. Another approach to increasing external validity would be to investigate whether the results generalize to other languages or contexts. Do Java programmers who write auction programs also tend to accidentally lose assets? How does task performance compare between Obsidian and other linearly-typed languages, such as Nomos [Das et al. 2019]?

Some of the usability results for Obsidian may generalize to other languages that use ownership, such as Rust. Future work should investigate whether the usability tradeoffs we observed here occur in other languages.

We observed many Obsidian programmers making unsafe use of `disown`. Future research should investigate how to more safely provide features that are safe in unusual situations but *unsafe* in situations that arise commonly. Some languages use naming conventions, such as "unsafe," to denote features that defeat the language's type system (e.g., Haskell's `System.IO.Unsafe.unsafePerformIO`), but here, the feature is a required part of the type system rather than a way of escaping from it.

17 CONCLUSION

As programming languages are tools for empowering programmers, empirical methods offer an opportunity for designers to provide evidence of the benefits of their work. In this study, we showed that programmers who used Obsidian were able to complete more of the tasks correctly than those using Solidity (§ 12). We also showed that ownership alone can be used effectively with a short training period and that assets can be used to detect bugs that would otherwise likely be inserted. However, we also identified areas of risk relating to language features that weaken the checks that the compiler provides. Although few language designs have been evaluated in this way, our work shows that it is possible to empirically evaluate a novel language, to both support hypotheses of usability as well as identify areas for potential improvement. We also hope that our findings of usability for the less-common type system features we analyzed will lead to more adoption of safer, more sophisticated type systems in future languages.

ACKNOWLEDGMENTS

This research was sponsored by the National Security Agency Department of Defense award H9823018D0008; by the National Science Foundation awards CNS1423054 and CCF1901033; by the United States Air Force Office of Scientific Research award FA8702-15-D-0002; by two IBM PhD Fellowship awards; and by Ripple. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

REFERENCES

Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L Mazurek, and Sascha Fahl. 2017. Security developer studies with GitHub users: Exploring a convenience sample. In *Proceedings of the Thirteenth USENIX Conference on Usable Privacy and Security*. 81–95.

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented Programming. In *Companion of Object Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*. 1015–1022. https://doi.org/10.1145/1639950.1640073

Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE-WGR '19)*. 177–210. https://doi.org/10.1145/3344429.3372508

Kevin Bierhoff and Jonathan Aldrich. 2007. Modular Typestate Checking of Aliased Objects. In *Object-oriented programming systems, languages, and applications (OOPSLA '07)*. 301–320. https://doi.org/10.1145/1297027.1297050

Luís Caires and Frank Pfenning. 2010. Session types as intuitionistic linear propositions. In *International Conference on Concurrency Theory (CONCUR '10)*. https://doi.org/10.1007/978-3-642-15375-4_16

Michael Coblenz, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2020a. Obsidian vs. Solidity RCT Replication Package. (8 2020). https://doi.org/10.1184/R1/12771074.v1

Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2018. Interdisciplinary Programming Language Design. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '18)*. 133–146. https://doi.org/10.1145/3276954.3276965

Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2019a. PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design. arXiv:1912.04719

Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2017. Glacier: Transitive Class Immutability for Java. In *International Conference on Software Engineering (ICSE '17)*. IEEE Press, 496–506. https://doi.org/10.1109/ICSE.2017.52

Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. 2020b. Obsidian: Typestate and Assets for Safer Blockchain Programming. *ACM Transactions on Programming Languages* 42 (2020). Issue 3. https://doi.org/10.1145/3417516 To appear.

Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2019b. Smarter Smart Contract Development Tools. *2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. https://doi.org/10.1109/WETSEB.2019.00013

J. Czerwonka, M. Greiler, and J. Tilford. 2015. Code Reviews Do Not Find Bugs. How the Current Code Review Best Practice Slows Us Down. In *International Conference on Software Engineering (ICSE '15, Vol. 2)*. 27–28.

Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. 2019. Resource-Aware Session Types for Digital Contracts. arXiv:1902.06056 [cs.PL]

Robert DeLine and Manuel Fähndrich. 2004. Typestates for Objects. In *European Conference on Object-Oriented Programming (ECOOP '04)*. https://doi.org/10.1007/978-3-540-24851-4_21

Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. 2016. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International conference on financial cryptography and data security*. https://doi.org/10.1007/978-3-662-53357-4_6

The Rust Project Developers. 2017. What is Ownership? (2017). Retrieved November 15, 2017 from https://doc.rust-lang.org/book/second-edition/ch04-01-what-is-ownership.html

Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. 2002. More Dynamic Object Reclassification: Fickle II. *ACM Trans. on Programming Languages and Systems* 24, 2 (March 2002), 153–191. https://doi.org/10.1145/514952.514955

Ethereum Foundation. 2020a. *Ethereum Project*. Retrieved February 18, 2020 from http://www.ethereum.org

Ethereum Foundation. 2020b. *Solidity*. Retrieved February 18, 2020 from https://solidity.readthedocs.io/en/develop/

Ethereum Foundation. 2020c. *State Machine*. Retrieved February 18, 2020 from https://solidity.readthedocs.io/en/v0.4.24/common-patterns.html#state-machine

Ethereum Foundation. 2020d. *Withdrawal from Contracts*. Retrieved February 25, 2020 from https://solidity.readthedocs.io/en/v0.6.3/common-patterns.html#withdrawal-from-contracts

Ethereum Foundation. 2020. *Simple Open Auction*. https://solidity.readthedocs.io/en/v0.6.3/solidity-by-example.html#simple-open-auction

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Trans. on Programming Languages and Systems* 36, 4, Article 12 (Oct 2014), 44 pages. https://doi.org/10.1145/2629609

Luke Graham. 2017. *$32 million worth of digital currency ether stolen by hackers*. CNBC. Retrieved November 2, 2017 from https://www.cnbc.com/2017/07/20/32-million-worth-of-digital-currency-ether-stolen-by-hackers.html

Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering* 19, 5 (oct 2014), 1335–1382. https://doi.org/10.1007/s10664-013-9289-1

David J. Houston and Lilliard E. Richardson. 2007. Risk Compensation or Risk Reduction? Seatbelts, State Laws, and Traffic Fatalities. *Social Science Quarterly* 88, 4 (2007), 913–936. https://doi.org/10.1111/j.1540-6237.2007.00510.x

Kadena. 2019. PACT. https://pact.kadena.io

Erik Meijer and Peter Drayton. 2004. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA '04 Workshop on Revival of Dynamic Languages*.

The University of Glasgow. 2001. *System.IO.Unsafe*. https://hackage.haskell.org/package/base-4.12.0.0/docs/System-IO-Unsafe.html

John F. Pane, Brad A. Myers, and Leah B. Miller. 2002. Using HCI techniques to design a more usable programming system. In *Human Centric Computing Languages and Environments (HCC '02)*. 198–206. https://doi.org/10.1109/HCC.2002.1046372

Victor Pankratius and Ali-Reza Adl-Tabatabai. 2014. Software engineering with transactional memory versus locks in practice. *Theory of Computing Systems* 55, 3 (2014), 555–590. https://doi.org/10.1007/s00224-013-9452-5

Andrea Pinna, Simona Ibba, Gavina Baralla, Roberto Toonelli, and Michele Marchesi. 2019. A Massive Analysis of Ethereum Smart Contracts Empirical Study and Code Metrics. *IEEE Access* 7 (2019). https://doi.org/10.1109/ACCESS.2019.2921936

Mozilla Research. 2015. *The Rust Programming Language*. Retrieved February 18, 2020 from https://www.rust-lang.org

Oracle Corp. 2019. *Secure Coding Guidelines for the Java SE, version 4.0*. Retrieved February 18, 2020 from https://www.oracle.com/technetwork/java/seccodeguide-139067.html

Chris Sadler and Barbara Ann Kitchenham. 1996. Evaluating Software Engineering Methods and Tool—Part 4: The Influence of Human Factors. *SIGSOFT Softw. Eng. Notes* 21, 5 (Sept. 1996), 11–13. https://doi.org/10.1145/235969.235972

Franklin Schrans, Daniel Hails, Alexander Harkness, Sophia Drossopoulou, and Susan Eisenbach. 2019. Flint for Safer Smart Contracts. (2019). arXiv:1904.06534

Robert C Seacord. 2013. *Secure Coding in C and C++*. Addison-Wesley Professional.

Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer smart contract programming with Scilla. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '19)*. https://doi.org/10.1145/3360611

Emin Gün Sirer. 2016. *Thoughts on The DAO Hack*. Hacking, Distributed. Retrieved February 18, 2020 from http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/

Stack Overflow. 2019. *Developer Survey Results 2019*. Retrieved February 18, 2020 from https://insights.stackoverflow.com/survey/2019

Andreas Stefik and Stefan Hanenberg. 2014. The Programming Language Wars: Questions and Responsibilities for the Programming Language Community. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Portland, Oregon, USA) *(Onward! 2014)*. 283–299. https://doi.org/10.1145/2661136.2661156

Andreas Stefik and Susanna Siebert. 2013. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)* 13, 4 (2013), 19. https://doi.org/10.1145/2534973

Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Engineering* SE-12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. 2014. Structuring Documentation to Support State Search: A Laboratory Experiment about Protocol Programming. In *European Conference on Object-Oriented Programming (ECOOP '14)*. https://doi.org/10.1007/978-3-662-44202-9_7

Nick Szabo. 1997. Formalizing and Securing Relationships on Public Networks. *First Monday* 2, 9 (1997). https://doi.org/10.5210/fm.v2i9.548

The Linux Foundation. 2020a. Hyperledger. (2020). Retrieved February 18, 2020 from https://www.hyperledger.org

The Linux Foundation. 2020b. *Hyperledger Fabric*. Retrieved February 18, 2020 from https://www.hyperledger.org/projects/fabric

Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. 2016. An Empirical Study on the Impact of C++ Lambdas and Programmer Experience. In *International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, 760–771. https://doi.org/10.1145/2884781.2884849

Philip Wadler. 1990. Linear types can change the world. In *Programming concepts and methods*, Vol. 2. 347–359.

Serdar Yegulalp. 2018. *Rust language is too hard to learn and use, says user survey*. https://www.infoworld.com/article/3324488/rust-language-is-too-hard-to-learn-and-use-says-user-survey.html