Obsidian: a Safe and Natural Programming Language for Blockchain Applications

Dr. Mark Sherman

Director, Cyber Security Foundations

March 9, 2018

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM18-0301

Existing blockchain programs are vulnerable

- Over **$40M** were stolen from TheDAO due to a bug in the implementation (June 2016)
- **$32M** were stolen due to a bug in a commonly used blockchain program (i.e., smart contract, June 2017)

Bugs in blockchain programs (smart contracts) cannot be fixed after deployment

We want to build correct software, but current approaches have been shown to have security vulnerabilities:

Re-entrancy attacks
Separation of money accounting from money storage

Obsidian: a new programming language

- Obsidian is a blockchain-based language with the goals
  - Make certain vulnerabilities impossible
  - Make it easier to write correct programs
  - Show effectiveness and correctness
- Obsidian programs consist of
  - **contracts**—similar to classes in Java—which contain fields, states
  - **transactions**—similar to methods

Obsidian: Core Features

- Obsidian contains core features to allow users to write safe programs easily and effectively.
- First-class typestate programming
  - Natural way to express sequences of transaction steps
  - Formal methods exist to prove effectiveness
  - Shown to be helpful in documentation, but no studies of ease of writing code
- Linear types
  - Natural way to express consumable resources
  - Formal methods exist to prove safety
  - Novel integration with an imperative language

Object Orientation vs Typestate Orientation

A file object has defined operations:

Open
Read
Write
Close

A file object has defined states:

isOpen
isClosed

Operations and States combine to define legal invocations

		Operations			
		Open	Read	Write	Close
States	isOpen	Illegal	Permitted	Permitted	Permitted
	isClosed	Permitted	Illegal	Illegal	Illegal

Object oriented programming: organize by operation

```
Object FileObject {
   Open(File){
     if isOpen(File)error
     else {
        fopen(File);
        FileState = isOpen;
    }
   Read(File)returns char{
     if isClosed(File)error
     else return fread(File);
    }
   Write(File, char){ … }
   Close(File){
        if isClosed(File) error;
        fclose(File);
        FileState = isClosed;
     }
}
```

Typestate programming: organize by state

```
Type FileState {
   isOpen(File){
      Read(File) returns char {
         return fread(File);}
      Write(File,char) { … }
      Close(File) {
         fclose(File);
         FileState = isClosed;}
   }
   isClosed(File){
      Open(File) {
         fopen(File);
         FileState = isOpen;
      }
   }
}
```

Typestate

- Blockchain programs commonly type state-oriented
- Obsidian makes type state first-class
  - An object in Obsidian has a **typestate** that restricts which **transactions** (operations) can be invoked on it.
- **State transitions** in a transaction can change the type state of an object
  - State transition sequences frequently can be inferred by analyzing program flow at compile time
- **States** and the **transactions** that can change state are organized into modules called **contracts**

Example Obsidian Program

```
contract LibraryPatron {
  state NoCard {




  }
  state HasCard {




  }
}
```

- A `LibraryPatron` is always in either the `NoCard` or `HasCard` state

Adapted from Barnaby, et al.

Example Obsidian Program

```
contract LibraryPatron {
  state NoCard {




  }

  state HasCard {
    transaction getBook(){
      …
    }
  }
}
```

- A `LibraryPatron` is always in either the `NoCard` or `HasCard` state
- `getBook` can only be called in `HasCard` state
  - Calling from `NoCard` state results in compile-time error

Adapted from Barnaby, et al.

Example Obsidian Program

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

  state HasCard {
    transaction getBook(){
      …
    }
  }
}
```

- A `LibraryPatron` is always in either the `NoCard` or `HasCard` state.
- `getBook` can only be called in `HasCard` state
  - Calling from `NoCard` state results in compile-time error
- `->HasCard` is a state transition

Adapted from Barnaby, et al.

Linear Types

- Blockchain programs often manage some kind of resource
  - e.g., cryptocurrency, votes, items in supply chain
- **Resources,** defined as **Linear types,** allow the compiler to enforce “resource safety”:
  - Variables of linear type must be used exactly once in their defined scope, hence
    - Resources cannot be used more than once
    - Resources must be used before leaving the current scope (i.e., don’t lose it)

Linear Resource Example

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

    transaction DoubleSpend(Money m) {
        spendMoney(m);
        Bond b = exchangeForBond(m);
    }

    transaction ForgetToSpend(Money m) {
        return;
    }
}
```

Compiler error – m used twice

Compiler error – m never used

Adapted from Barnaby, et al.

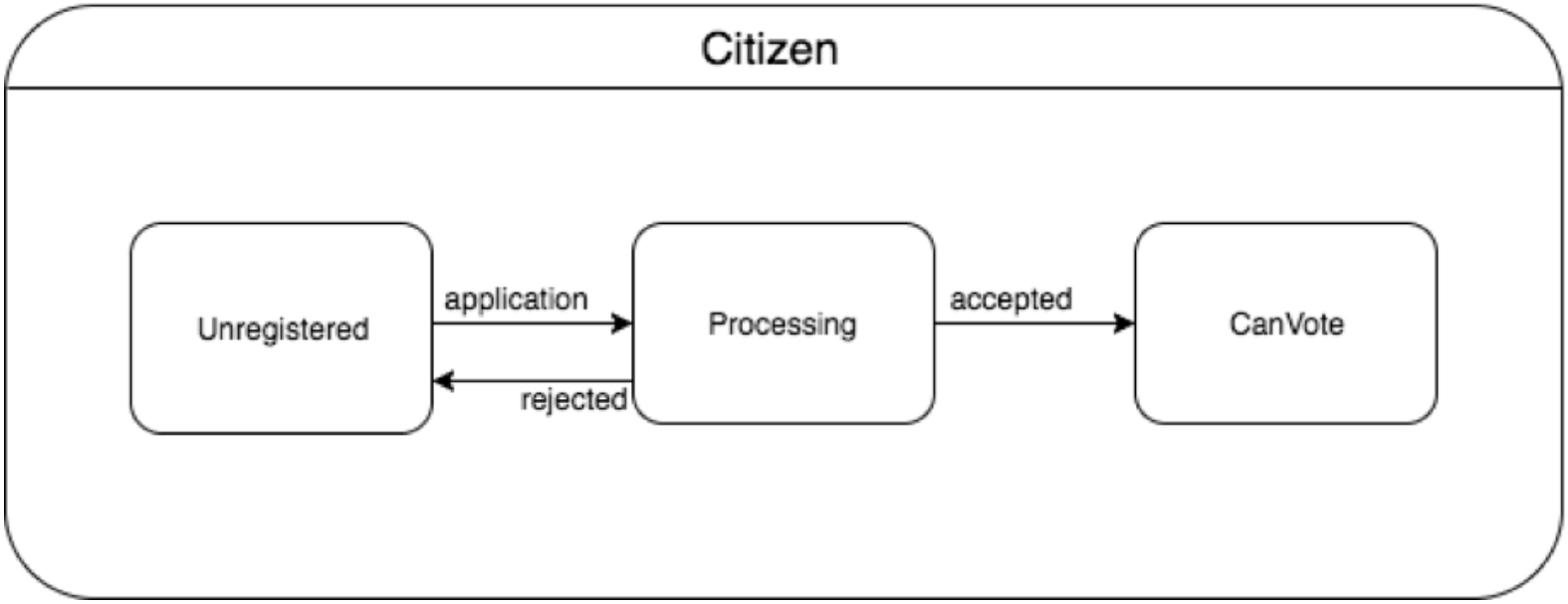
Usability

Programmers should be able to write correct Obsidian code easily and effectively.

Creating an intuitive language is hard! Many difficult design choices exist

Usability study

Participants were given a description of a voter registration system for a hypothetical democratic nation.



Usability study

1. Write pseudocode to implement program.
2. Given a state diagram modeling the voter registration system, modify pseudocode.
3. Given Obsidian tutorial (with no information on state transitions) invent syntax for state transitions and complete an Obsidian contract.
4. Shown three options for state transitions, complete a brief contract for each option.
5. Choose one of the three options and use it to complete the Obsidian program from part 3.

Usability study – Findings

- Programmers do not naturally consider state-based design when architecting code
- Most intuitive design: include all possible state actions explicitly within the state

Summary

- Obsidian contains core features—including **first-class typestate** and **linear resources**—to allow users to write safe programs easily and effectively
- Usable programming language design requires iteration and user testing
  - Obsidian is an active research language and continues to evolve

Research Team

Michael Coblenz, Jenna Wise, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Tyler Eltzel	Institute for Software Research, School of Computer Science, Carnegie Mellon University
Elli Kanal	Software Engineering Institute, Carnegie Mellon University
Celeste Barnaby	Wesleyan University
Rick Hull	IBM

Additional Reading

Michael Coblenz, “Obsidian: A Safer Blockchain Programming Language,” 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), http://ieeexplore.ieee.org/document/7965268/

Michael Coblenz, Elli Kanal, Jenna Wise, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Rick Hull, “Obsidian: a Safer Blockchain Programming Language,” 2017, https://resources.sei.cmu.edu/asset_files/Presentation/2017_017_001_506530.pdf

Eliezer Kanal, Michael Coblenz, “Obsidian - A Safer Blockchain Programming Language,” October 2017, https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=506444

Celeste Barnaby, Michael Coblenz, Tyler Etzel, Eliezer Kanal, Joshua Sunshine, Brad Myers, Jonathan Aldrich, “A User Study to Inform the Design of the Obsidian Blockchain DSL, SPLASH 2017, ” https://2017.splashcon.org/event/plateau-2017-a-user-study-to-inform-the-design-of-the-obsidian-blockchain-dsl

Source:

https://github.com/mcoblenz/Obsidian

Contact Information

Mark Sherman

Director, Cyber Security Foundations

CERT, Software Engineering Institute

4500 Fifth Ave

Pittsburgh, PA 15213

Telephone: +1 412.268.5800

Email: info@sei.cmu.edu, mssherman@sei.cmu.edu