Empirical Studies on the Security and Usability Impact of Immutability

Sam Weber (NYU), Michael Coblenz (CMU), Brad Myers (CMU), Jonathan Aldrich (CMU), Joshua Sunshine (CMU)
Acknowledgements

• This research is supported in part by NSA lablet contract #H98230-14-C-0140 and by NSF grant CNS-1423054.
• We are grateful for the assistance of Forrest Shull and Whitney Nelson on the prior work
Motivation

• APIs have a large impact on usability and security, very long-lasting
• Most guidance to API designers is very superficial: naming of functions, etc
• Goal: determine higher-level design principles
  • For both usability and security
  • Evidence-based
Immutability

• In programming languages, a data structure is *immutable* if its value cannot be changed after the structure has been created

• Security community likes immutability:
  • Immutable data structures are immune to TOCTTOU attacks
  • Real-life vulnerability:
    In Java, Class.getSigners() returns array of signers of the class. In early release, implementation returned actual internal array, not copy. Since arrays are mutable, malicious code could change signers!
Outline of Talk

• Background:
  • Stylos and Clarke, “Usability Implications of Requiring Parameters in Object’s Constructors”

• Summary of our project
  • Two main papers
    • “Exploring Language Support for Immutability” by Coblenz et al (ICSE 2016)
    • “Glacier: Transitive Class Immutability for Java” by Coblenz et al (ICSE 2017)

• Implications on the design of studies in this field
“Usability Implications of Requiring Parameters in Object’s Constructors”

• Two patterns for object construction:
  
  • Required Constructor
    
    ```javascript
    var foo = new FooClass(barValue);
    foo.Use();
    ```
  
  • Create-Set-Call
    
    ```javascript
    var foo = new FooClass();
    foo.Bar = barValue;
    foo.Use();
    ```

• Which is better?
Methodology

• Lab study, with think-aloud protocol
• Subjects did six tasks with different conditions
  • Each session 2 hours and 15 minutes
• Tasks were designed to assess code readability, debug-ability and initial writability
• Thirty participants
  • Recruited to cover multiple work styles (systematic, pragmatic, opportunistic)
  • All participants current or retired professionals with recent experience and proficiency in the language used
  • No students
Tasks

1. Write code you’d expect to read file and send its contents in email
2. Wrote code to do 1), using different conditions
3. Create objects using both conditions
4. Message queue debugging, using both conditions
5. Implement part of a small application
   1. Inventory of an online store
   2. APIs of both kinds were available to subjects
6. Reading code on paper

Follow-up interviews were done
Results

• Create-set-call greatly preferred and more usable by programmers!
  • ALL participants used create-set-call when designing their “expected” code
• Create-set-call better even for “systematic” programmers!

• This motivated our project:
  Is there really a trade-off between security and usability? Can we measure it?
Our project

1. Interview study with expert software engineers about how they controlled state.

2. Classification system for mutability restrictions created from review of literature and programming languages.

3. Language extension to Java, called Glacier, implementing “transitive class immutability”

4. Evaluations of Glacier
   a) Retrofitting existing applications
   b) Laboratory experiments

- ICSE 2016 paper
- ICSE 2017 paper
Interview study

• Semi-structured interviews
• Eight subjects
  • Engineers who work on large software projects
    • Typically projects with millions of lines of code, hundreds of developers
  • Minimum of seven years experience, mean of fifteen
Results – bugs caused by state change

• One participant complained about library boundaries. Third-party library modified state. By the time this was discovered, two project subgroups were permanently distrustful of each other.

• “Oh God, like, most of them! . . . my favorite is where you have data that is supposed to be immutable and is only settable once in theory but that’s not well enforced and so it ends up getting re-set later either because it gets re-initialized or because someone is doing something clever and re-using objects or you have aliasing where two objects reference the same other object by pointer and you make changes. . . .”
Use of immutability and related features

• Immutability was used frequently, at least conceptually
• Several projects were described which had mutability restrictions as core part of architecture
• Very poor language support for it
  • Complaints about C++’s copying of objects, difference between abstract and concrete states, etc
## Dimensions of state restrictions

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Possible choices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>immutability, read-only restriction,</td>
</tr>
<tr>
<td></td>
<td>assignability, ownership</td>
</tr>
<tr>
<td>Scope</td>
<td>object-based, class-based</td>
</tr>
<tr>
<td>Transitivity</td>
<td>transitive, non-transitive</td>
</tr>
<tr>
<td>Initialization</td>
<td>relaxed, enforced</td>
</tr>
<tr>
<td>Abstraction</td>
<td>abstract, concrete</td>
</tr>
<tr>
<td>Backward compat.</td>
<td>open-world, closed-world</td>
</tr>
<tr>
<td>Enforcement</td>
<td>static, dynamic</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>polymorphic, non-polymorphic</td>
</tr>
</tbody>
</table>
Choices

• Type of restriction
  • **Immutability**: prevents changes via all references to the object
  • **Read-only restriction**: mutation cannot be done via a particular reference, but might change via other references
    • In C “const int *x” means that x cannot be used to change the memory location to which x points
  • **Assignability**: disallow assignment (like final in Java)

• Transitivity
  • Do the restrictions pertain to all the objects reachable via a given object? For example, if a list is immutable, is each object in the list immutable?
Summary

• Found that immutability is highly valued by programmers of large systems
• Language support is very poor and doesn’t match use cases
• Transitive immutability, in particular, seems desired
Glacier: Transitive Class Immutability for Java

• Definition and formal model of transitive class immutability in Featherweight Java

• Implementation in tool called Glacier
  • Static typechecker
  • Using Java’s annotations (so can be parsed by standard Java tools)

• Evaluations:
  • Case studies on two real software projects
  • Formal user study comparing Glacier with Java’s final
Glacier

• Types can be annotated with @Immutable to indicate immutability
  • Other types are implicitly @MaybeMutable

• All fields must be @Immutable, and fields cannot be assigned outside class’s constructors

• Subclasses of @Immutable class must be @Immutable

• No subtyping relationship between @Immutable and @MaybeMutable types

• Complications to deal with arrays, templates, etc
Featherweight Java extension

Mod ::= assignable | final

CL ::= \([@\text{Immutable}]\) class C extends C implements T \{ Mod \(\overline{C} T; K \overline{M}\) \}

IF ::= \([@\text{Immutable}]\) interface I extends T \{ M-Dec \}

K ::= C(\overline{C} T) \{ super(\overline{T}); this.\overline{f} = \overline{f}; \}

M-Dec ::= C m(\overline{C} \overline{T})

M ::= M-Dec \{ return t; \}

t ::= x | tf | t.m(\overline{T}) | new C(\overline{T}) | (C) t | tf = t | v ::= new C(\overline{T})

\[\begin{align*}
\text{fields}(D) &= \overline{g} \\
&\text{OK in } C \\
K &= c(\overline{g}, \overline{T}) \{ \text{super}(\overline{g}); \text{this.}\overline{f} = \overline{f}; \} \\
&\text{MUT-FREE} \\
\forall i. C_i \text{ IMMUTABLE} \\
\text{methods}(I) \subseteq (\text{decl}(M) \cup \text{methods}(D))
\end{align*}\]

\[\begin{align*}
\text{Immutable class } C \text{ extends } D \text{ implements } T \{ \text{Mod } \overline{C} T, K \overline{M} \} \text{ OK}
\end{align*}\]
Case Study: ZK Spreadsheet

• ZK Spreadsheet is commercial, partially open-source Java spreadsheet
  • Can import Excel documents, provides server-bases spreadsheet component that can be inserted into web pages via Ajax client-side component

• Originally did not use immutable structures, because of fear of performance cost of copying
  • However, modifications of cell styles end up effectively copying style structure

• Refactored model portion of app (~36 KLOC) and updated remainder of app appropriately (~21 KLOC)
  • Took Michael Coblenz about 20 hours
  • Discovered two bugs in ZK Spreadsheet
Case Study: Guava ImmutableList

• Google’s Guava project provides useful library, including immutable collections
• Annotated immutable collections with Glacier
  • Only problem was with a cache
User Study

• Compared effectively users could use Glacier to implement immutability as compared to using `final`

• 20 subjects: experienced Java programmers
  • Between 4 and 19 years of programming experience, mean of 9.5 years, at least a year of Java experience

• Each pair of subjects randomly assigned to Glacier or final condition
  • Participants received training in Glacier or the use of final. Participants were allowed to ask questions during training.

• Four programming tasks assigned, study took about 1.5 hours

• Participant’s audio and video of computer screen was recorded for analysis
Tasks

1. Making “Person” class immutable (persons associated with Address)
2. Make “Accounts” immutable, representing user accounts on a computer system
3. Replication of structure of “getSigners()” bug
4. Preventing introduction of mutation in a hash map implementation
Results

<table>
<thead>
<tr>
<th></th>
<th>final</th>
<th>Glacier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctly enforced immutability in Person</td>
<td>0/10</td>
<td>10/10</td>
</tr>
<tr>
<td>Correctly enforced immutability in Accounts</td>
<td>0/10</td>
<td>9/10</td>
</tr>
<tr>
<td>FileRequest.execute() tasks</td>
<td>4/8</td>
<td>7/7</td>
</tr>
<tr>
<td>without security vulnerabilities</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HashBucket.put() tasks without bugs</td>
<td>3/10</td>
<td>7/7</td>
</tr>
</tbody>
</table>

- Despite `final` being core part of Java language, programmers consistently had problems using it to implement immutability
- Replicated `getSigners()` bug: very easy to make mistake using `final`
- Main threat to validity: simplicity of tasks
Methodological Observations

• Our studies came to oppose conclusions to Stylos and Clarke’s. Why?
  • Programming style might have changed since 2005
    • Personal impression: web programming greatly encouraged the use of immutability
  • We asked about projects’ core data structures. S&C primarily considered utility classes. Was this the key factor?
    • Although S&C’s study did include programming online store app.
  • Perhaps time limitations resulted in over-stressing learnability?
Conclusions

• Immutability valuable in APIs
  • Many non-trivial design issues
• In general, although the Usability community has started work on usability in API designs, the security community needs to play a role
• Recommend empirical investigations of secure design recommendations
• We recommend qualitative studies, such as interviews, in addition to quantitative studies
  • Quantitative studies usually limited in duration, and this may affect results
  • Qualitative studies provide validity-check on quantitative studies