Verifying functional purity in Java

$f(x)$ ≠ `f(x)`

Matt Finifter, Adrian Mettler, Naveen Sastry, **David Wagner**
UC Berkeley

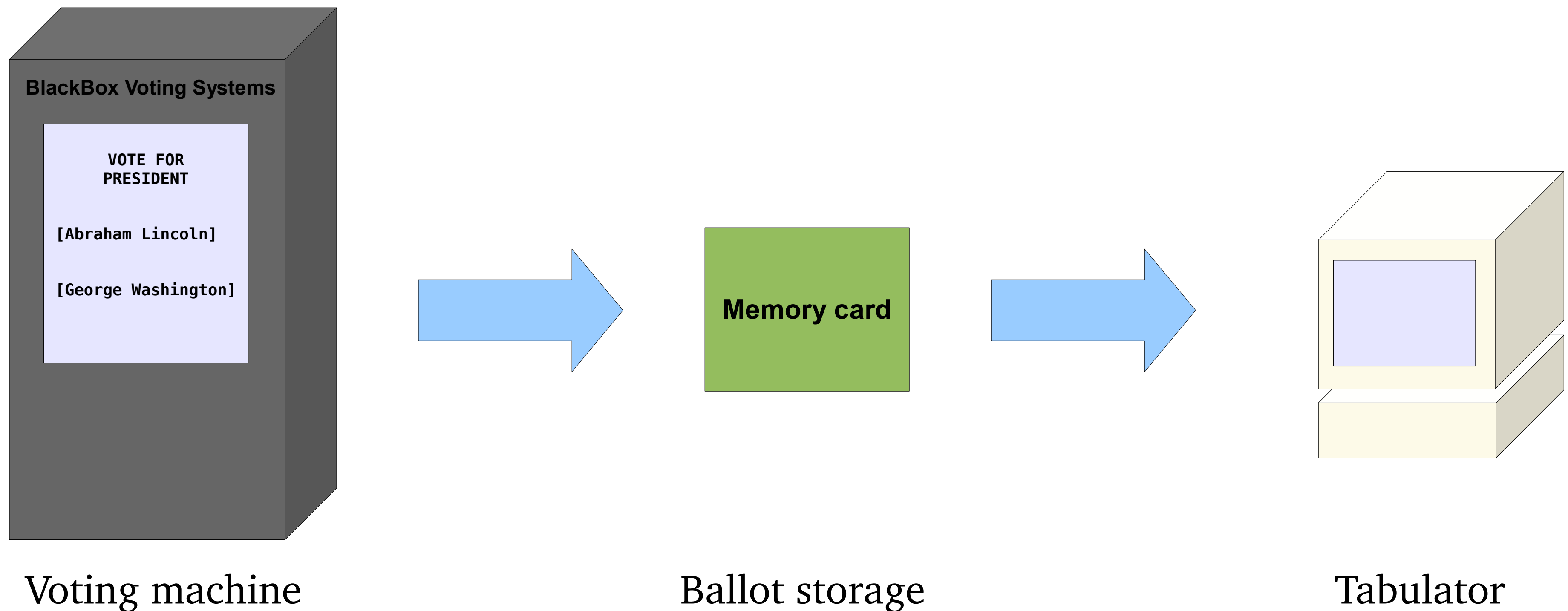
http://www.joe-e.org

Outline

- **Functional purity and applications**
- How we verify functional purity
- Evaluation of our techniques

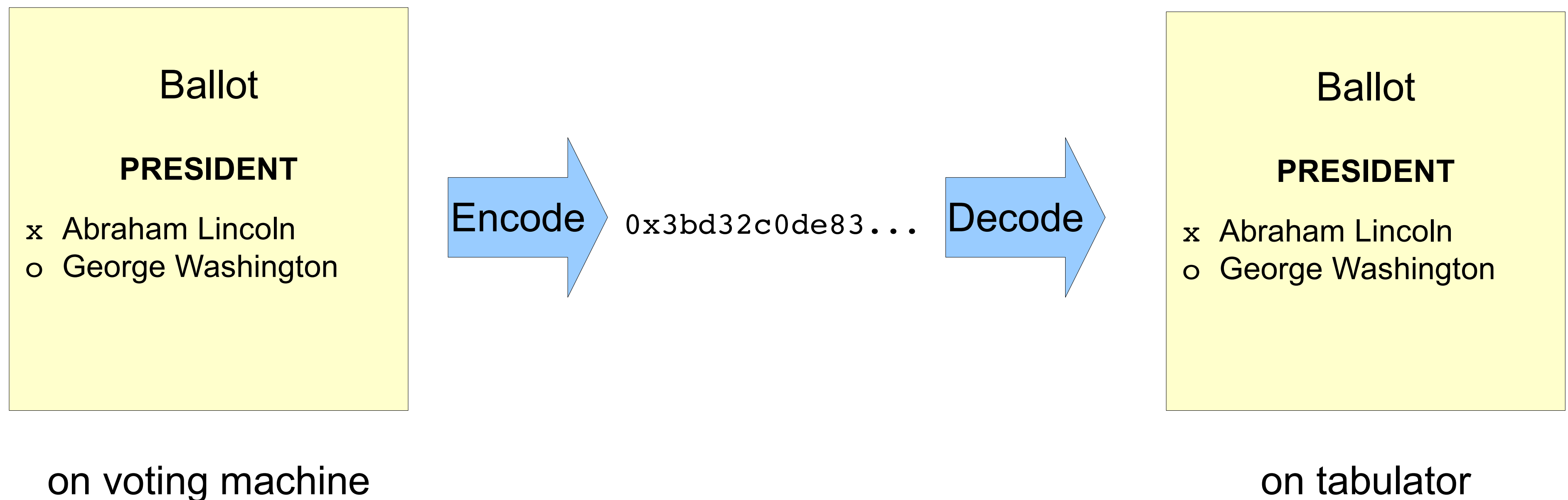
Example: a voting system

- Votes are recorded, then serialized to disk or network for tally



Property: Invertibility

- Security goal: ballot as decoded on tabulator should be the same as serialized by voting machine
- `decode` should be an inverse of `encode`



Solution

- On the voting machine, check the serialization before storage:

```
data = encode(ballot);
Ballot decoded = decode(data);
if (!decoded.equals(ballot)) abort();
write(data);
```

Will it work?

```
data = encode(ballot);
Ballot decoded = decode(data);
if (!decoded.equals(ballot)) abort();
write(data);
```

- `decode` must give the same answer on the tabulator as it does on the voting machine
- `decode` should also have no side effects

Functional purity

- *Functionally pure* methods behave like mathematical functions
- Fixed mapping from inputs to outputs
- Have no side effects

Not all methods act like functions

- Side effects
  - to arguments
  - to other program state
  - to the outside world
- Non-functional dependencies
  - on program state other than arguments
  - on the outside world

$f(x) \neq$ `f(x)`

Functionally pure methods

- Have no side effects
  - Only modify objects *created within the method*
  - Have no effects on outside world
- Are deterministic in their arguments
  - Always behave identically with “equivalent” arguments

Some nontrivial functions

- Sorting
- Image decoding
- Parsing
- Compilation

Applications for functional purity

- Many security and reliability applications
  - Verifying invertibility
  - Reproducible document rendering
  - Sandboxing of untrusted code
  - Non-interference of invocations
  - Thread-safety without locking
  - Facilitating testing and debugging

Outline

- Functional purity and applications
- **How we verify functional purity**
- Evaluation of our techniques

Problem

How can methods be verified as functionally pure?

One approach: functional languages

- In purely functional languages, all methods are functionally pure
- Disadvantages
  - Inflexible: entire program must be functional
  - Can’t directly use imperative algorithms and coding style
  - Unfamiliar to many programmers

Our approach: a new language class

- “Deterministic object-capability languages”
- Memory safety used to limit what methods can see and do
  - Some methods are verifiably functionally pure
  - Others are not pure
- Advantages
  - Allows use of most existing algorithms and idioms

Joe-E: a new language

- Defined as a strict subset of the Java language
- Uses the existing Java toolchain, JVM
- Verifier checks additional properties that make functional purity easy to achieve

Idea

- Memory safety restricts access to objects
  - Cannot modify an object without a reference to it
  - Cannot depend on the value of an object without a reference to it
- If all access to external state is represented as objects
  - Can't have side effects without access to shared *mutable* objects
  - Can only depend on *varying* objects to which it has access

Properties of Joe-E

- Inherits Java's memory safety
- Verifies that types annotated to be *immutable* contain no mutable state
- Ensures that the observable global scope only contains objects that are immutable and unvarying across executions

Functional Purity in Joe-E

- Any method with only immutable arguments is functionally pure
  - It has no access to shared mutable state
  - Only variable state is its arguments
- Purity is a restriction of the method's interface
- No additional restrictions are imposed on the body of pure methods

Verifying immutability

```
class IntPair implements Immutable {
     final int first;
     int second;     ← error

     ...

     void mutate() {
        second++;
     }
}
```

- Joe-E checks that all fields of an immutable class are final and of an immutable type

Verifying immutability

```
class IntPair implements Immutable {
     final int first;
     final int second;

     ...

     void mutate() {
        second++;     ← error
     }
}
```

- Joe-E checks that all fields of an immutable class are final and of an immutable type

Enforcing global scope properties

```
static int invocations = 0;    ← error

static void sideEffecting() {
     invocations++;
}
```

- Joe-E requires all static variables to be final and immutable

Enforcing global scope properties

```
void sideEffecting() {
    System.out.println("Hello");
}
```

- Joe-E provides a whitelisted subset of the Java library
- Non-final and non-immutable static variables are excluded

Enforcing global scope properties

```
void sideEffecting() {
     System.setProperty(“foo”, “bar”);
}
```

- Static methods that have side effects excluded

Enforcing global scope properties

```
void sideEffecting() {
     new java.io.File("/etc/passwd")
        .delete();
}
```

- Constructors and static methods that return objects with external effects excluded

Enforcing global scope properties

```
String nondeterministic() {
     return System.getEnv(“TERM”);
}

long nondeterministic() {
     return new Date().getTime();
}
```

- Constructors and methods that return variable results excluded

A functionally pure method

```
int[] sort(IntArray array) {
    int[] arr = new int[array.length()];
    for (int i = 0; i < array.length(); ++i) {
        int newval = array.get(i);
        int j;
        for (j = i; j > 0 && arr[j-1] > newval; --j) {
            arr[j] = arr[j-1];
        }
        arr[j] = newval;
    }
    return arr;
}
```

Another functionally pure method

```
// this method is still functionally pure
int[] sort(IntArray array) {
    int[] arr = array.toIntArray();
    inplaceSort(arr);
    return arr;
}

// even though this method is not functionally pure
inplaceSort(int[] arr) {
    for (int i = 0; i < array.length(); ++i)
        int j, newval = arr[i];
        for (j = i; j > 0 && arr[j-1] > arr[i]; --j) {
            arr[j] = arr[j-1];
        }
        arr[j] = newval;
    }
}
```

Outline

- Functional purity and applications
- How we verify functional purity
- **Evaluation of our techniques**

Evaluation

- Methodology: ported Java programs to Joe-E to verify purity of their top level methods
- Verified purity of block cipher encryption and decryption, ballot deserialization, HTML parsing

	Source lines of code		**Number of classes**		**Number of methods**	
	Before	After	Before	After	Before	After
AES Block Cipher	319	276	1	1	9	9
Voting Machine	688	692	25	25	80	79
HTML Parser	12652	10848	94	99	965	947

HTML parser

- Open-source HTML parser, 12K SLOC
- To make it Joe-E:
  - Mutable static flags used for options changed to parameters
  - Fix use of `System.out` and the default locale
- Wrapper method performing a default parse (with no options) was then pure

```
public NodeList parse(String html)
```

Waterken

- REST-style application server with transparent persistence written by Tyler Close
- 132 classes, 8246 SLOC of infrastructure code written in Joe-E in addition to a trusted base of Java code
- Substantial portion of methods happened to be functionally pure

	Num. pure	Total num.	% Pure
Methods	89	524	17
Constructors	37	128	29

Conclusions

- Functional purity is useful for security
- It is achievable in imperative languages without making the whole language functional
- Retrofitting existing code is possible, but our approach better suited to new code

Thank You

- For more information about Joe-E
  - http://www.joe-e.org

Examples

- `max(3, -1)` = `3`
- `md5("foo")` = `acbd18db4cc2f85cedef654fccc4a4d8`
- `sort([3,6,-4,2,-21,0])` = `[-21,-4,0,2,3,6]`
- `compile("int main()...")` = contents of binary executable
- `jpeg_decode(0xffd8ffe0...)` =

Methods verified to be pure

- In an AES encryption/decryption library:
  - public ByteArray encrypt(ByteArray plain)
  - public ByteArray decrypt(ByteArray cipher)
- In a voting machine implementation:
  - public static boolean deserializesTo(ByteArray serialized, BallotMessage bm)
- In an HTML parsing library:
  - public NodeList parse(NodeFilter filter) throws ParserException

Invertibility

- A pure decode function would allow a fail-stop check that encoding is the inverse of decoding, i.e. that votes are recoverable as cast
- We just add a check after encoding:

```
data = encode(ballot);
if (!decode(data).equals(ballot)) abort();
```

- Side-effect freeness ensures that adding the check doesn't break the program.
- Determinism ensures that if the ballot decodes correctly to match the original right after encoding it, it will decode correctly in the future when it is tabulated.

Outline

- What is functional purity?
- What is it good for?
- **How do we verify functional purity?**

Outline

- **What is functional purity?**
- What is it good for?
- How do we verify functional purity?

Outline

- What is functional purity?
- **What is it good for?**
- How do we verify functional purity?

Reproducibility: document rendering

- Consider a contract between two parties, Alice and Mallory
- Mallory creates a PDF document specifying the contract in January; both parties view the document, agree to its terms, and cryptographically sign the file.
- On March 1, as agreed, Alice sends Mallory $100.

Jan. 1

Alice will pay Mallory **$100** on March 1

Reproducibility: document rendering

- On March 2, Mallory goes to court and claims that she was underpaid. She shows the signed contract to a judge.
- The document now states that she is owed $200 instead of $100 as agreed.
- Even when Alice opens it on her computer, it looks wrong.
- Its rendering depends on the system date, making the document look different from when it was signed.
- If the renderer was pure, it would be a deterministic function of the bytes of the file and this kind of trickery would not be possible.

Jan. 1

Alice will pay Mallory **$200** on March 1

Untrusted code execution

- If untrusted code is verified to be pure, the damage it can do is strictly limited: when invoked it can give us the wrong answer, but cannot corrupt or spy on the rest of the program
  - Side-effect freeness prevents it from modifying any program state or causing external effects to the machine or network
  - Determinism prevents it from observing any data other than its legitimate inputs
- The above properties apply to malicious inputs as well as untrustworthy code
  - Damage that can be done is restricted to returning arbitrary output from the method

Untrusted code execution

- In cases where the adversary can already cause arbitrary output without exploiting bugs in the method, a verified-pure method can be completely untrusted (Bernstein)
- Examples:
  - Email delivery-address extraction (to a string)
  - Image or video decoding and transcoding

Defensive consistency

- A module that interfaces with multiple clients is said to be *defensively consistent* if it always provides correct service to clients that follow its preconditions even if other clients violate its preconditions
- Pure methods are always defensively consistent provided each invocation only includes input from one client as there is no information flow between invocations

Thread-safety and Testability

- Pure methods are always thread-safe, requiring no locks
- Pure methods are easy to test
  - Reproducing a failing invocation requires reproducing only the arguments
  - Other program state or configuration details cannot cause or prevent the same failure from recurring

Verifying methods as pure

- Recall a functionally pure method is *side-effect free* and *deterministic*
- So how do we verify that a method has these properties?
- Our approach: an *object-capability language*

$$f(x) = \texttt{f(x)} \;?$$

Object-Capability language

- References to objects in the language act as capabilities
  - All state that can be communicated between methods is stored in objects
  - Objects can only be accessed via references
  - References can only propagate by being passed as arguments or return values, or by a field write on a shared object
  - References are unforgeable
  - Use of references is limited by lexical scope
- All actions are performed using appropriate capabilities
  - The global scope does not provide methods that affect the world, e.g. filesystem or network access, without an appropriate capability
  - It also does not provide means to affect other computation, e.g. mutable global variables or data structures

Deterministic Object-Cap. Language

- In addition to restrictions on effects, for functional purity we need similar restrictions on nondeterminism.
- The language cannot provide any nondeterministic primitives
  - Nondeterministic thread scheduling
  - Recovery from errors such as running out of memory
  - Specification nondeterminism potentially an issue
- The observable global state must be the same at any time during any invocation
  - Includes all values and effects observable by calling global methods as well as instance methods on objects thus obtained
  - Prohibits dependencies on the time of day, system configuration, values read from I/O, shared PRNG, non-transparent caching, etc.

Functional computation

- In a deterministic object-capability language, a method that has only immutable arguments must always give the same result if given the same* arguments
  - It always starts off with the same view of the world
  - It can only read from the global scope, which is always the same, and the arguments, which are the same on equivalent invocations
  - Lack of linguistic nondeterminism means that since the invocations start in observationally equivalent states, the invocations will proceed equivalently and terminate* in equivalent states
- Note that there is no additional restrictions on the *body* of the method; purity is a property of the method's *interface*
  - It is free to call impure methods internally, which may arbitrary modify internal data structures
  - Purity of methods can be verified by inspection

Joe-E: a Deterministic Object-Capability Subset of Java

- Makes use of Java's memory-safety and type-safety
- Restricts Joe-E code's view of the global state to exclude methods and fields that allow side-effects and nondeterminism
- An `Immutable` interface is used to mark classes that are transitively immutable. Joe-E verifier ensures that immutable types' fields are all `final` and of primitive or immutable type

Evaluation

- Retrofitting existing code can be hard, especially if it uses idioms that are not possible in Joe-E