Discovering API Usability Problems at Scale

Emerson Murphy-Hill, NC State*
Caitlin Sadowski, Google
Andrew Head, UC Berkeley*
John Daughtry, Google
Andrew Macvean, Google
Ciera Jaspan, Google
Collin Winter, Google

*Emerson and Andrew completed this work while a visiting scientist and intern at

Detailed API usability problems at scale?

API Misuse is widespread: 88% of Google Play applications have at least one API usage mistake [3]

Better API design can improve API usability

API usability evaluation techniques are expensive (experiments/interviews)

Scalable techniques lower fidelity (e.g. Stack Overflow, MSR)

[3] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In Proceedings of the Conference on Computer & Communications Security. 73–84.

Context: Software Development at Google

- Giant codebase (2+billion LOC)
- Browsable patches
- FUSE-based file system stores every save

Approach: Stop Motion

1. Identify **patches** of interest (java files)
2. Identify file **snapshots** associated with the patch
3. Compare adjacent file snapshots using JDT (Java Development Tools) and Gumtree (AST differencing tool) to produce AST **diff**
4. **Identify** API changes of interest

obj.a(...) → obj.b(...)

Study

About 3 weeks of patches from July 2017

Two patterns:

- Method call: obj.a(...) to obj.b(...)
- Static method call: Class.a(...) to Class.b(...)

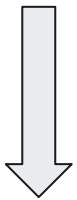
APIs that frequently satisfied pattern

Limitations

- Requires fine-grained snapshots
- Does not work predictably in the presence of syntax errors
- Only finds method replacement patterns
- Currently manual analysis, summarization, and visualization
- Results are most frequent changes, not most important changes

Results: Collections

```
void foo(List<String> s){

    ImmutableList.of(s);
```



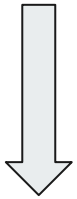
```
void foo(List<String> s){

    ImmutableList.copyOf(s);
```

174 times, engineers changed:

`of` to `copyOf`

Results: Collections

```
void foo(List<String> s){

    ImmutableList.of(s);
```



```
void foo(List<String> s){

    ImmutableList.copyOf(s);
```

`static <E> ImmutableList<E>`	**`of()`** Returns the empty immutable list.
`static <E> ImmutableList<E>`	**`of(E element)`** Returns an immutable list containing a single element.
`static <E> ImmutableList<E>`	**`of(E e1, E e2)`** Returns an immutable list containing the given elements, in order.
`static <E> ImmutableList<E>`	**`of(E e1, E e2, E e3)`** Returns an immutable list containing the given elements, in order.
`static <E> ImmutableList<E>`	**`copyOf(Collection<? extends E> elements)`** Returns an immutable list containing the given elements, in order.
`static <E> ImmutableList<E>`	**`copyOf(E[] elements)`** Returns an immutable list containing the given elements, in order.
`static <E> ImmutableList<E>`	**`copyOf(Iterable<? extends E> elements)`** Returns an immutable list containing the given elements, in order.
`static <E> ImmutableList<E>`	**`copyOf(Iterator<? extends E> elements)`** Returns an immutable list containing the given elements, in order.

https://google.github.io/guava/releases/21.0/api/docs/com/google/common/collect/ImmutableList.html

Results: Protocol Buffers

27 times, engineers changed:

`copyFrom` to `copyfromUTF8`

Results: Protocol Buffers

27 times, engineers changed: `copyFrom` to `copyFromUtf8`

com.google.protobuf

Class ByteString

java.lang.Object
    com.google.protobuf.ByteString

All Implemented Interfaces:

`Serializable`, `Iterable<Byte>`

```
public abstract class ByteString
extends Object
implements Iterable<Byte>, Serializable
```

Immutable sequence of bytes. Substring is supported by sharing the reference to the immutable underlying bytes, as with `String`. Concatenation is likewise supported without copying (long strings) by building a tree of pieces in `RopeByteString`.

static ByteString	**copyFrom**(byte[] bytes) Copies the given bytes into a ByteString.
static ByteString	**copyFrom**(byte[] bytes, int offset, int size) Copies the given bytes into a ByteString.
static ByteString	**copyFrom**(ByteBuffer bytes) Copies the remaining bytes from a java.nio.ByteBuffer into a ByteString.
static ByteString	**copyFrom**(ByteBuffer bytes, int size) Copies the next size bytes from a java.nio.ByteBuffer into a ByteString.
static ByteString	**copyFrom**(Iterable<ByteString> byteStrings) Concatenates all byte strings in the iterable and returns the result.
static ByteString	**copyFrom**(String text, Charset charset) Encodes text into a sequence of bytes using the named charset and returns the result as a ByteString.
static ByteString	**copyFrom**(String text, String charsetName) Encodes text into a sequence of bytes using the named charset and returns the result as a ByteString.
static ByteString	**copyFromUtf8**(String text) Encodes text into a sequence of UTF-8 bytes and returns the result as a ByteString.

https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/ByteString

Results: Optional

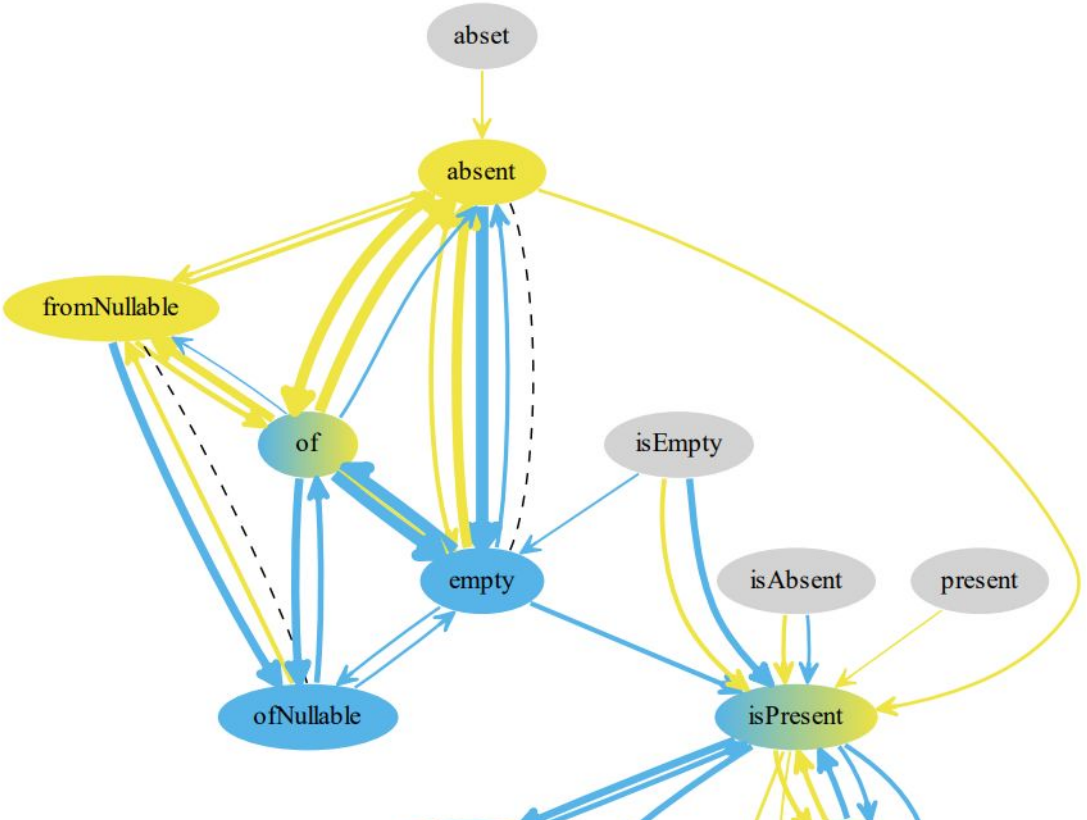
```
Optional<String> optStr =

  Optional.of(getString());

if(optStr.isPresent() {

  return optStr.get();

}
```

Hundreds of changes between two versions of this API:

- Java Platform 8
- Guava

Results: Optional

```
Optional<String> optStr =
  Optional.of(getString());
if(optStr.isPresent() {
  return optStr.get();
}
```



Results: Logging

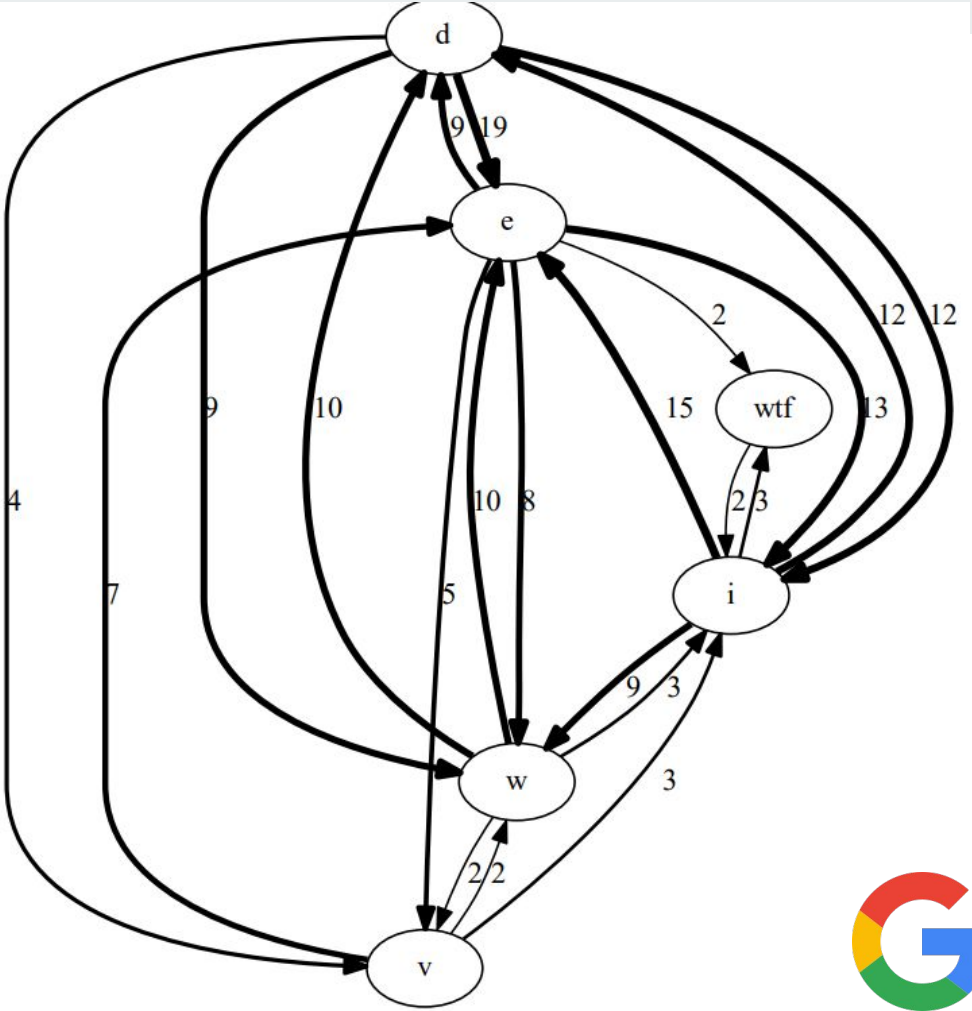
```
Log.v()

Log.d()

Log.i()

Log.w()

Log.e()

Log.wtf()
```

When to use each logging method?

- Number one question on StackOverflow for Android logging

Results: Logging

When to use each logging method?



Stop Motion: API updates into insights

- Names which don't document difference are confusing (`of` vs `copyOf`)
- Developers will confuse similar names (`copyFrom` vs `copyFromUtf8`)
- Having two similar but not the same APIs is costly (`optional`)
- Getting the right level of log statements is hard

Future Work

- Automation
- Address limitations
- Browsable results for API maintainers
- Suggestions when making common edits.

Google

Stop Motion: API updates into insights

- Names which don't document difference are confusing (`of` vs `copyOf`)
- Developers will confuse similar names (`copyFrom` vs `copyFromUtf8`)
- Having two similar but not the same APIs is costly (`optional`)
- Getting the right level of log statements is hard