How C++ Developers Use Immutability Declarations: an Empirical Study

Jonathan Eyolfson
University of California, Los Angeles
Los Angeles, CA, USA
eyolfson@ucla.edu

Patrick Lam
University of Waterloo
Waterloo, ON, Canada
patrick.lam@uwaterloo.ca

***Abstract*—Best practices for developers, as encoded in recent programming language designs, recommend the use of immutability whenever practical. However, there is a lack of empirical evidence about the uptake of this advice. Our goal is to understand the usage of immutability by C++ developers in practice. This work investigates how C++ developers use immutability by analyzing their use of the C++ immutability qualifier, `const`, and by analyzing the code itself. We answer the following broad questions about `const` usage: 1) do developers actually write non-trivial (more than 3 methods) immutable classes and immutable methods? 2) do developers label their immutable classes and methods? We analyzed 7 medium-to-large open source projects and collected two sources of empirical data: 1) `const` annotations by developers, indicating an intent to write immutable code; and 2) the results of a simple static analysis which identified easily `const`-able methods—those that clearly did not mutate state. We estimate that 5% of non-trivial classes (median) are immutable. We found the vast majority of classes do carry immutability labels on methods: surprisingly, developers `const`-annotate 46% of methods, and we estimate that at least 51% of methods could be `const`-annotated. Furthermore, developers missed immutability labels on at least 6% of unannotated methods. We provide an in-depth discussion on how developers use `const` and the results of our analyses.**

I. Introduction

Many modern programming languages include mechanisms that allow developers to declare that objects (or parts thereof) are immutable. Immutability enables both developers and compilers to better reason about code and potentially enables compiler optimizations. For instance, objects without mutable state are immune from unexpected changes in mutable state, thus simplifying debugging and program maintenance. Furthermore, immutable objects can be shared between concurrent threads of execution without the need for locking, helping to enable automatic parallelization.

Informally, an immutability declaration encodes a developer's assertion that some part of the program state does not change. Different languages (or proposed extensions to existing languages) support immutability declarations specific to those languages. C++ provides the `const` keyword for developers to specify immutability, and best practices for C++ developers include the extensive use of `const` [1]. However, C++ developers using `const` have great freedom to choose which form of immutability they will implement.

Our goals are to empirically investigate whether immutability language features are used in real codebases and, furthermore, to understand which particular features developers actually use (particularly class versus method immutability). Many kinds of language features are difficult to advocate for. Often, there is no hard evidence that a feature improves code quality, especially before developers have a chance to use that feature. We can, however, observe the code that developers write in practice. Empirical studies can affirm or dispel myths about best practices versus actual practices, and provide language designers with objective data to consider when crafting new language features or paradigms.

We collect data from 7 moderate-to-large open-source C++ programs across a range of application domains. We believe that if developers choose to use `const` in their code, then they find it to be useful; after all, developers may simply omit `const` annotations[1]. Hence, if immutability declarations are present in real programs, then there is a good chance that they are useful in language designs. Two sources of data support our findings: 1) developer-provided `const` qualifiers, and 2) results from our static analysis, which identifies so-called easily `const`-able methods.

We pose our research questions at the level of classes and methods. Classes are a fundamental encapsulation unit for C++. In C++, developers may only apply the `const` qualifier to methods that are class members, and not to classes as a whole. Informally, a developer who `const`-qualifies a method m is stating that m does not change the state of the `this` receiver object; that is, `this` is immutable across the execution of m. C++ is flexible with respect to the implementation of immutable methods, leaving it to the developer to ensure immutability. Types may be `const`-qualified, but that only means that a purportedly-

This work was supported in part by Canada's National Science and Engineering Research Council. Jonathan Eyolfson contributed to this work while at the University of Waterloo.

[1]While developers may be required to use `const` to call libraries, they are free to omit it in internal code.

non-mutating subset of the type's interface is available.

We investigate immutability at class and method levels with the following Research Questions:

RQ 1. How often do developers use C++'s flexible `const` annotations to indicate immutable classes and methods?

RQ 2. What is the proportion of immutable methods and classes in typical C++ projects?

The first RQ investigates developer-provided annotations, which we'll denote as "immutable" (in quotes), while the second RQ investigates the immutability of both classes and methods as-written. We found:

RQ 1 Findings. Developers labelled a median of 5% (between 0% and 50% across case studies) of non-trivial (more than 3 methods) classes as "immutable" (all methods `const`). Also, 46% of methods had `const` annotations.

RQ 2 Findings. We found that developers correctly annotated immutable classes as "immutable", so we estimate the true number of immutable classes to be around 11%. The proportion of non-trivial immutable classes computed over all classes (trivial and non-trivial) is 2%, which is quite small. We estimated that about 52% of methods were immutable, including at least 12% of immutable methods (6% of all methods) that developers did not annotate.

Contributions. The contributions of this paper are:

- We explored the usage of `const` for 7 moderate-to-large C++ case studies and identified trends in `const` usage that hold across our set of case studies.
- We developed and implemented a static analysis that identifies immutable methods (which could easily be labelled `const`) and applied it to our set of case studies, finding many such methods.
- Combining the `const` counts and the static analysis results, we estimated the true prevalence of immutable classes and methods in codebases that resemble our case studies. To our knowledge, we are the first to empirically investigate the use of both `const` and immutability in actual codebases.

Our work contributes observations of developer behaviour across real-world codebases, thus informing developers and language designers about best practices today.

Furthermore, programming language designers often add features to languages based on intuition and experience. (Tunnell Wilson et al [2] discuss shortcomings of crowdsourcing language design.) At language design level, a language's definition of immutability may be too strict or too broad. A too-strict definition limits the ability of developers to label objects as immutable; in such a case, developers are likely to under-annotate code, and code that is annotated is likely to be trivial. A too-broad definition limits the ability of the developers to leverage immutability definitions and leaves them no further ahead: they must still inspect allegedly-immutable code to verify that it is, indeed, not mutating state. Languages must provide the right level of protection to make immutability declarations useful for developers and compilers.

We hope that our work can guide future language designers when considering features to add to languages. We have also released the source code to our Immutability Check tool and believe that it can help developers better understand how they use `const` in their own projects. One project has already expressed interest in adding our tool to their Continuous Integration workflow.

II. Motivating Examples

Our research questions empirically examine how developers might use immutability-related language features in their codebases, at both class and method levels. RQ1 explores, in part, whether the class is an appropriate level of granularity for an immutability declaration. In practice, do developers write classes that are immutable? An immutable class has no user-visible mutable state. We also explore all-mutating classes, which are never invariant under method calls. (C++ somewhat confounds the issue: a class that has no `const` annotations could be all-mutating, or it could simply be unannotated. Section V discusses this in more detail.) Many classes are in neither of these categories—they contain a mix of mutating and immutable methods. RQ2, in part, explores how common immutable methods are in practice, and compares the number of methods that developers declare to be "immutable" (using `const`) to the number of methods that developers could declare to be immutable. Our answer to RQ2 estimates the number of methods that could be declared immutable using our notion of an easily `const`-able method—a method that can quickly be seen to not change internal state.

Immutable class. The most straightforward example of an immutable class is an immutable tuple implementation. Consider, for instance, this `Point` class:

```
class Point { int x, y;
public: Point(int x, int y) : x(x), y(y) {}
  int getX() const { return x; }
  int getY() const { return y; } };
```

Once a `Point` object is constructed, it cannot be changed, since it does not expose its state, nor does it provide functions that mutate its state.

All-mutating class. On the other hand, an all-mutating class contains no methods that do not modify state. Random number generators are typical examples:

```
class RandomNumberGenerator { int64_t state;
public: RandomNumberGenerator() { state = 17L; }
  int64_t getNext() {
    state = state * 134775813 + 1;
    return state; } };
```

Mix class. We find that most classes contain some methods that do not modify state and some methods that do modify state. Any mutable class with at least one `const`-annotated getter will be a mix class.

```
class RNG2 { int64_t seed, state;
public: RNG2() { seed = 17L; state = seed; }
  int64_t getNext() {
    state = state * 134775813 + 1;
    return state; }
  int64_t getSeed() { return seed; } };
```

In this case, `getNext()` modifies state, while `getSeed()` does not. These methods are unannotated. However, the ground truth is that `getSeed()` is an immutable method, so this class contains a mix of immutable and mutable methods.

Easily `const`-able methods. Three methods in our examples above are immutable. `Point::getX()` and `getY()` are labelled as `const`, while `RNG2::getSeed()` is not.

Our static analysis would identify all 3 of these methods as easily `const`-able, by observing that these methods do not change any state associated with `this`. For instance, there are no writes to fields in `getSeed`, and this method returns a copy of the field. Since it cannot mutate, and returns a field copy, `getSeed` ought to be `const`.

RQ2 explores how many methods are like `getSeed`: immutable yet not labelled as such. We have reported such methods to developers and some proposed annotations have been merged into the upstream repositories.

III. On immutability and C++

The C++ standard [3] allows developers to `const`-qualify types and methods. C++ compilers aim to verify non-transitive abstract immutability for `const` types and methods. Non-transitive means that immutability does not extend to pointee objects reached through an immutable object. Abstract means that the bits of an immutable object may change, but not its abstract state. For objects of `const` type, non-`const` methods are not callable and fields are not writable. `const` methods are compiled with `this` being `const`. (Because the bits may change, it is difficult for a C++ compiler to optimize based on `const`. C++ `const`[2] is strictly for developers' benefit at compile-time; developers must explicitly opt-in to the use of `const`.)

In the absence of `mutable` fields and casts, `const`-qualifying a C++ method implies concrete immutability. However, in the full language, C++ `const`-qualified methods only need to satisfy abstract immutability: developers can write `const`-qualified methods that mutate state. C++ specifies that `const`-qualified methods are not to change the receiver (`this`)'s abstract state. But abstract state is domain-specific and it would be impossible for the language specification to define what it means to not change abstract state. It would be challenging for compilers to verify invariance of abstract state, and it is thus primarily the responsibility of the developer to ensure that the abstract state remains unchanged. Developers may follow project-specific conventions that specify stricter versions of immutability.

C++ immutability is fine-grained: developers may add immutability annotations to methods (not just types). A `const`-qualified method shows developer intent about that method's immutability. As long as a method satisfies the language requirements, C++ leaves it up to the developer whether to `const`-qualify the method or not—C++ developers may write non-`const`-annotated immutable methods.

RQ1 and RQ2 investigate immutability from two angles. RQ1 investigates the use of the `const` qualifier; as discussed above, `const` does not actually guarantee immutability. RQ2 aims to estimate transitive concrete immutability in codebases, combining counts of `const` annotations and the output of our easily-`const`-able analysis (Section IV), which statically analyzes methods to determine whether or not they mutate the receiver object's state.

A simple `grep` does not suffice to understand `const` usage, as the `const` keyword applies differently to types, methods, and fields. Our RQs address developers' actual and potential use of `const` in their implementations.

Section VI discusses read-only references and other notions of immutability.

IV. Technique

We designed an Immutability Check tool to answer our research questions, building on the LLVM compiler infrastructure [4] and the Clang frontend. Our tool records compiler invocations executed during a build and re-runs the front-end stages to collect information on `const` usage.

Immutability Check analyzes all C++ code in each source code repository, with the following exceptions. Our tool ignores classes with no source; abstract classes; and classes with no `public` members (such classes cannot be used in external code; if subclassed, we consider the subclass). It also ignores classes based on inheritance—classes which inherit (transititively) from a class with no source; and classes whose inheritance hierarchy includes a template parameter (i.e. where the subclass is unknown). We handle classes that otherwise use templates, e.g. `list<string>`. Within each included class, Our tool analyzes all non-`static` public methods except constructors/destructors, conversion operators, and `operator=`.

Immutability Check stores its results in a web-accessible database. This database holds the public members of every class, plus additional metadata. For methods, Immutability Check stores whether or not each method is `const`-qualified, along with the results of the `const`-able static analysis for that method. For fields, Immutability Check stores whether or not each field is `mutable`, and whether or not the outermost type of the field is `const`-qualified.

[2]`gcc` includes a non-standard `const` attribute for use in optimizations. It is not well documented, and we do not investigate it here.

Limitations. We designed Immutability Check to keep running times reasonable. It only considers calls to methods in the same class or within the current class hierarchy (reachable through subclasses and superclasses). The lack of a complete class hierarchy is the key source of unsoundness. As discussed below, Immutability Check also verifies that arguments are not fields, e.g. `o.m(this.f)`, assuming that local and global variables are not reachable from `this`. In practice, we found that methods within the same class hierarchy do not pass fields of the receiver method through arguments, instead accessing them directly in the callee.

Explicit `const`. We call a field *explicitly* `const` if the field's outermost type is `const`-qualified. In that case, the field contains an object that is at least *shallow immutable* (i.e. the fields in the class itself do not change, but data pointed to by its fields may change). On the other hand, if a field is a pointer, then the pointer itself cannot change, but the data pointed to may. Contrast shallow immutability with *transitively* `const`, which forbids mutations to the object as well as anything obtained by dereferencing that object.

To answer RQ1 (how often developers use `const`), we look for classes that only have `const`-qualified methods and explicitly `const` fields. We label such classes "Immutable", based on developer `const` annotations. To answer RQ2 (what proportion of methods and classes immutable), we manually validate the immutability of the classes labelled "Immutable" on a sample of classes. Furthermore, we look for classes without `const`-qualified methods and without public fields. Such classes have all methods declared as mutable and no accessors. In that case, either the developers neglected to use `const`, or all methods could potentially change the class's state. We investigated whether method implementations for such classes mutated state or not.

We found that a non-trivial fraction of "Unannotated" classes indeed had all of their methods potentially mutate object state (i.e. contain 0 immutable methods), which was somewhat surprising to us; we did not expect that developers would actually write such classes.

This-origin. A key property of an expression in an object-oriented language is whether it refers to something reachable from `this`; e.g. `this.f` refers to a field of `this`. By assumption, locals and globals are not reachable from `this`.

We implemented a simple static analysis to determine that either an expression is maybe reachable from `this`, or definitely not, subject to the limitations above. Figure 1 shows some expressions and whether they are this-origin.

Queries. This-origin information answers two queries about methods: whether a method mutates the `this` object, and what the method returns. The two options for whether a method mutates are: 1) does not mutate, and 2) maybe mutates. A method maybe mutates if either: 1) there is an assignment to any this-origin expression, 2) there is a call to a member function that maybe mutates, or 3) there is a call to a function with a this-origin argument. Properties 2) and 3) depend on information computed while traversing the partial call graph as described above (using class hierarchy analysis to resolve calls and collapsing cycles). We use query results to estimate the proportion of immutable methods (RQ2).

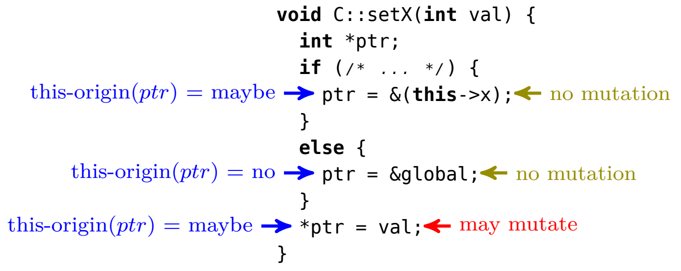
```
void C::setX(int val) {
  int *ptr;
  if (/* ... */) {
    ptr = &(this->x);
  }
  else {
    ptr = &global;
  }
  *ptr = val;
}
```

Fig. 1. Our static analysis calculates whether a method is easily `const`-able. This-origin analysis (on the left) calculates whether local variables contain values reachable from `this`. The query stage (on the right) calculates whether a statement may mutate `this`.

Static analysis example. Figure 1 shows the results of our analysis on a (contrived) example. The this-origin analysis, whose results are shown on the left in blue, determines whether expressions are "this-origin" (evaluate to something reachable from `this`). In the example's `if` branch, local variable `ptr` gets a value reachable from `this`. In the `else` branch, we assume globals are not this-origin, so `ptr = &global` is not this-origin. At the merge, we take the union of `ptr`'s values and conclude that it may contain a value reachable from `this`. The query stage of our analysis, depicted on the right, detects mutations of the receiver object. Both writes to `ptr` are ignored in the `if` and `else` branches since `ptr` is a local variable and hence not an object field. However, since `ptr` is dereferenced on the last line, where it may point to state reachable through `this`, we would conclude the last line may mutate `this`.

Easily `const`-able methods. Combining answers from our queries and transitivity, we can determine whether a method is easily `const`-able. An easily `const`-able method m: 1) does not mutate `this` (according to our query); and 2) all methods it depends on through method calls, or inheritance, do not mutate `this`. This ensures we do not report stub methods (methods that return a constant and are overriden by subclasses) as easily `const`-able. We count easily `const`-able methods to estimate the proportion of immutable method implementations for RQ2.

Almost all methods reported as easily `const`-able were indeed `const`-able. (Return values that were references prevented the application of some of our suggested `const`s—we estimated that this happened less than 1% of the time).

Immutable methods and classes. Our definition of immutable methods includes both easily `const`-able methods as well as methods that implement abstract immutability; we can only find easily `const`-able methods and `const`-labelled methods, and aim to estimate the true number of immutable methods. As for classes, we distinguish between "Immutable" classes and immutable classes (no quotes).

TABLE I
We chose 7 open-source codebases as case studies, ranging from 13 000 to over 3 million lines of code.

Project	Version	kLOC	Classes	Methods	Description
LLVM	4.0.0	≈ 3 200	10 518	55 229	compiler infrastructure
OpenCV	3.2.0	1 167	2 220	6 624	computer vision library
Protobuf	3.3.1	625	407	1 813	compiler for data serializer/unserializer
fish	2.5.0	112	129	299	modern shell
Mosh	1.2.6	14	74	302	mobile shell
Ninja	1.7.2	13	36	165	build system
libsequence	1.8.7	18	33	199	library for evolutionary genetics

"Immutable" classes have all methods and fields `const`-qualified. Immutable classes include "Immutable" classes as well as those implementing non-transitive abstract immutability.

Pointer analysis not needed. For this work, we do not need a pointer analysis, since our analysis knows which expressions may be a field. Our analysis is insensitive to mutation of objects pointed-to by local variables, and ignores those writes. If a local variable is dereferenced before a write, our analysis checks whether that local variable may point to a field of `this`. If so, the analysis would conclude the method may mutate. That is, we use the may-be-field information as may-points-to analysis.

V. Experimental Results

We evaluated 7 open source projects: fish, libsequence, LLVM, Mosh, Ninja, OpenCV, and Protobuf. Table I lists characteristics of our chosen projects. We took care to mitigate bias in our project selection as best we could. We chose these projects because they are popular, well maintained, and span a variety of domains, including I/O-focussed applications as well as both graph-manipulating and array-manipulating codes. About half of the projects were libraries while the rest were applications. The projects vary as to the development community (public or primarily company-sponsored). Our projects include many of the 10 most-starred GitHub C++ projects. We included two projects from GitHub's top 10 (OpenCV and Protobuf), and we believe that Clang/LLVM, were it developed on GitHub, would surpass Swift/LLVM's popularity (which is top 3). fish-shell would also be in the top 10, but GitHub identifies it as a shell project. We believe it is valuable to include both smaller and less-popular projects alongside larger more-popular projects.

Table II summarizes class-level information for all of our case studies. For each case study, we include the distribution of all of its classes with a public interface. Figure 3 summarizes method-level information for all our case studies. Each case study includes the proportion of developer annotated `const` and non-`const` methods, as well as the proportion of these methods we identifiy as easily `const`-able. Our tool's running time is within an order of magnitude of a full build for each case study; we omit precise times due to space, but have made the tool available for readers to run.

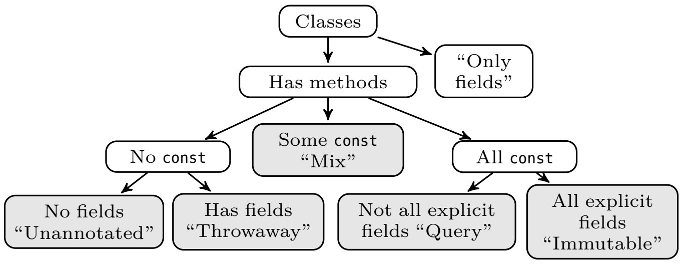

Fig. 2. Immutability Check divides classes with methods into 5 main categories, depending on which members they contain.

Reading the class table (Table II). Figure 2 illustrates relationships between the 6 categories of classes. The first two categories, "Immutable" and "Query", only have `const` methods. "Immutable" has every public field explicitly declared `const`. "Query" has public fields which may be modified by member functions when accessed through a non-`const` reference (or by non-member functions, although those are out of scope for our purposes). "Mix" classes have both `const` and non-`const` methods. The last two categories, "Throwaway" and "Unannotated", both have only non-`const` methods. The "Throwaway" classes have public fields and zero or more accessors that could trivially be made `const`. We call these classes throwaway

TABLE II
Percentage breakdown of class categories for all projects. All projects, except libsequence, mostly consist of classes with a mix of `const` and non-`const` qualified methods. A plurality of libsequence classes are "Immutable".

Classes	LLVM	OpenCV	Protobuf	fish shell	Mosh	Ninja	libsequence
Has methods	79.8	72.3	86.5	69.0	91.9	88.9	87.9
— "Immutable"	10.8	15.0	26.3	6.2	6.8	2.8	48.5
— "Query"	4.5	11.1	1.0	17.8	4.1	5.6	3.0
— "Mix"	45.2	31.9	32.7	20.9	70.3	47.2	33.3
— "Throwaway"	5.4	5.9	0.5	8.5	2.7	13.9	3.0
— "Unannotated"	13.8	8.5	26.0	15.5	8.1	19.4	0.0
"Only fields"	20.2	27.7	13.5	31.0	8.1	11.1	12.1

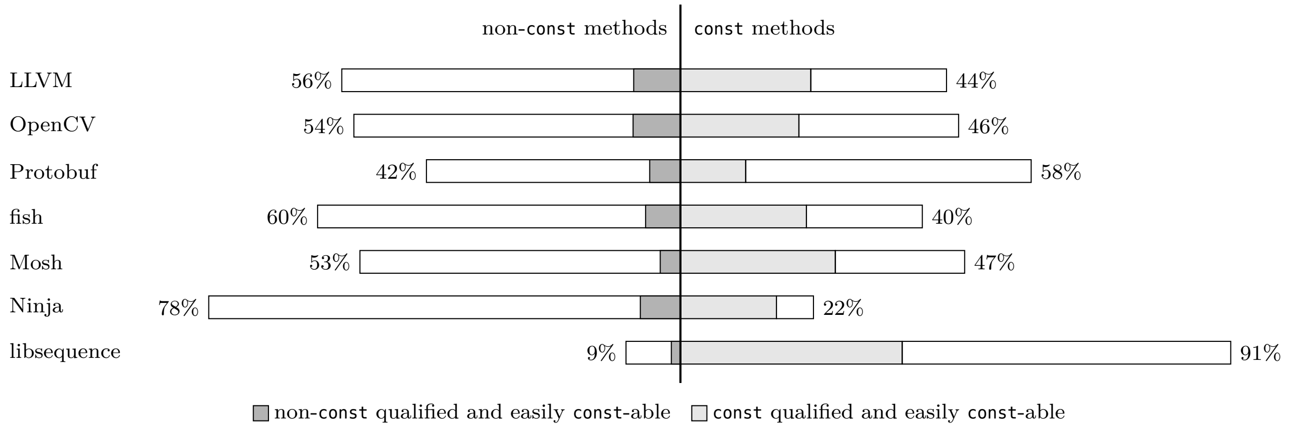

Fig. 3. Developers `const`-qualify a median (across projects) of 48% of methods as `const`. libsequence stands out with 91% of methods already `const`-qualified. A further 12% of non-`const` methods are easily `const`-able (i.e. missed opportunities for `const`). Our easily `const`-able analysis confirmed 49% of `const`-qualified methods as indeed `const`.

since there are no const qualifiers on methods at all, as well as public fields. Typically such classes are a quick and dirty structure. "Unannotated" classes, by contrast, do not have public fields. In such classes, the developer has not said anything about the `const`-ness of the methods in the class. An unannotated class may in fact be an all-mutating class, where every method mutates the object—such a class would be fully annotated yet carry no `const` annotations. Or, the class may include immutable methods for which the developer opted to omit immutability declarations. Finally, our class tables show the number of "Only fields" classes (no methods) for completeness.

Reading the methods figure (Figure 3). Each case study has a corresponding box to the right of the name showing the distribution of methods. The total length of the box represents all methods. The thick black line in the middle of the figure divides methods with no `const` annotation (to the left of the line) from those with `const` annotations (to the right of the line). For clarity, we included the percentage of developer annotated non-`const` and `const` methods to the left and right of each box respectively. The shaded regions within the box represent the amount of easily `const`-able methods found by our tool. The dark grey region is the proportion of methods that developers did not annotate with `const` but our tool determined were easily `const`-able. This proportion of methods represents *missed opportunities* for `const` annotations in the case study. The light grey region is the proportion of methods that developers annotated with `const` and our tool agreed were easily `const`-able. The remaining methods on the developer annotated `const` side of the box represents the number of methods that are beyond our analysis's ability to recognize as `const` (possibly, though unlikely to be, incorrect).

A. LLVM

Compilers extensively manipulate structured intermediate representations, and must maintain invariants to preserve the meaning of the code being compiled. We expect that recording design intent with respect to immutability would be key to successfully developing LLVM.

TABLE III
All sampled LLVM classes that developers declared "Immutable" are in fact immutable; only 8/20 "Unannotated" classes are all-mutating.

Annotated as	Total	Non-trivial	Immutable	All-mutating
"Immutable"	1135	582	20 (of 20)	0
"Unannotated"	1456	546	2 (of 20)	8

Because LLVM has thousands of classes, we randomly sampled 20 "Immutable" and "Unannotated" classes for manual analysis. Table III shows that at a class level, we expect that about $637 = 582 + 10\% \times 546$ of the classes in LLVM will be non-trivial (more than 3 methods) immutable classes: sampling shows that about 100% of the `const`-annotated "Immutable" classes are immutable, while about 10% of the unannotated classes were immutable. This accounts for about 6% of the total number of LLVM classes. On the other hand, we expect that 218 classes will be non-trivial all-mutating, or about 2% of the total number of classes. A plurality of classes, and a majority of classes with methods, contained a mix of `const` and mutable methods. Thus, classes that developers labelled as "Immutable" were indeed immutable, while unannotated classes (with no `const` declarations) could have used them on some of their member functions more often than not.

Discussion. Many immutable classes that we manually inspected in LLVM were either checker or code generator classes. Code generator classes change program state (or emit side effects), but do not change the state of `this`. We also encountered some immutable code generator classes in our inspection of unannotated classes; these classes simply did not have any `const`-labelled methods.

B. OpenCV

To explore numeric codes, we studied the OpenCV computer vision library. The parts of OpenCV that carry out regular calculations on large arrays are more amenable to parallelization than codes that process graphs. As expected, OpenCV mutated many of its arrays. However, we found some usage of `const` on function parameters.

TABLE IV
Fewer than 1/3 of OpenCV's immutable and unannotated classes are non-trivial. Manual inspection showed that almost all sampled classes declared immutable are immutable, while 1/4 of unannotated classes are all-mutating.

Annotated as	Total	Non-trivial	Immutable	All-mutating
"Immutable"	332	45	18 (of 20)	0
"Unannotated"	188	62	0 (of 20)	5

For OpenCV, Table IV shows that we expect 90% of 45 non-trivial classes to be immutable, or about 41, which accounts for 1.8% of its classes. (Many of OpenCV's classes are trivial.) None of the sampled unannotated classes were immutable. Hence, we expect 25% of 62 non-trivial classes to be all-mutating, or about 16, which accounts for 0.7% of its classes. Similar to LLVM, classes that were labelled "Immutable" often were immutable, while unannotated classes often could have had some methods `const`-labelled.

Discussion. OpenCV contains many implementations of mathematical functions. OpenCV convention appears to implement these functions within classes (similar to the Strategy design pattern). An alternate system design could have used function pointers. In any case, OpenCV's function classes fit the definition of an immutable object. These function classes are often trivial (fewer than 4 methods).

C. Protobuf

Protobuf (protocol buffers) serializes structured data. We analyzed the protocol buffer compiler, which generates code (in a number of languages) to serialize and deserialize to/from specified data formats. This compiler also happens to contain generated protocol buffer code.

TABLE V
About half of Protobuf's classes are non-trivial. Manual inspection showed that almost all sampled classes declared immutable are immutable, while 20% of unannotated classes are all-mutating. Additionally, 10 unannotated classes were in fact immutable.

Annotated as	Total	Non-trivial	Immutable	All-mutating
"Immutable"	107	67	19 (of 20)	0
"Unannotated"	106	36	10 (of 20)	4

Table V estimates that about 82 of 407 (20%) Protobuf classes are immutable—95% of 67 non-trivial `const`-labelled classes, plus 50% of 36 non-trivial classes with no `const` labels. We expect about 8 (2%) non-trivial all-mutating classes, or 20% of 36. Protobuf is similar to OpenCV for all-`const`-labelled classes being immutable and non-`const`-labelled classes being all-mutating. But, Protobuf has a higher proportion of immutable classes that are free of `const` labels than our other projects.

Almost two-thirds of Protobuf's methods are immutable. However, most of these immutable methods were already declared as `const`; our easily `const`-able analysis does not contribute much. The Protobuf developers used `const` for complex methods: when looking at methods that the developers labelled `const`, the easily `const`-able analysis only found about half, which is fewer than for other projects.

Discussion. Protobuf, like LLVM, contains many classes that generate code. When we manually inspected them, we verified that they were immutable. Some of the immutable classes had no `const` labels and hence had originally been identified as unannotated classes.

D. fish shell

The fish shell (fishshell.com) is a modern command line shell. Shells are particularly concerned with file-based I/O.

TABLE VI
fish has no non-trivial immutable classes and only 6 non-trivial classes with no `const` annotations, of which only 2 are all-mutating.

Annotated as	Total	Non-trivial	Immutable	All-mutating
"Immutable"	8	0	0	0
"Unannotated"	20	6	0	2

Since fish and the subsequent projects have no more than 20 classes in each category, we exhaustively examined each of the "Immutable" and "Unannotated" classes.

As seen in Table VI, we expect fish to have 0 (and hence 0%) non-trivial immutable classes. It is possible that some of the classes that have some, but not all, `const`-labelled methods might actually be immutable; however, we believe that this is unlikely—if developers have `const`-labelled some of the immutable methods, they would be likely to have labelled all of them. Since we estimate that fish contains 0 immutable classes, we believe that developers did not leave out any `const` annotations on the immutable classes. About a third of classes without any `const` annotations could have used some.

In fish, 10% of non-`const` methods were easily `const`-able. Also, 52% of `const` methods in fish were easily `const`-able—the rest use more sophisticated immutability.

Adoption. We created a pull request by manually adding `const` qualifiers to the methods our tool identified as easily `const`-able. We were not able to add qualifiers to 3 out of 14 easily `const`-able methods, due to fields being taken by reference; we did not want a `const` method to return a mutable reference. The developers merged our pull request and asked about adding our tool to their Continuous Integration workflow.

E. Mosh

Mosh is a utility for maintaining terminal connections over low-quality (e.g. cellular data) networks. It sends and receives encrypted data over the network.

TABLE VII
MOSH AND NINJA HAVE 0 NON-TRIVIAL IMMUTABLE CLASSES AND 0 NON-TRIVIAL UNANNOTATED CLASSES. LIBSEQUENCE HAS 8 NON-TRIVIAL IMMUTABLE CLASSES AND 0 UNANNOTATED CLASSES.

mosh	Total	Non-trivial	Immutable	All-mutating
"Immutable"	5	0	0	0
"Unannotated"	6	0	0	0

ninja	Total	Non-trivial	Immutable	All-mutating
"Immutable"	1	0	0	0
"Unannotated"	7	2	0	0

libsequence	Total	Non-trivial	Immutable	All-mutating
"Immutable"	16	8	8	0
"Unannotated"	0	0	0	0

Table VII shows that we found Mosh (and ninja, below) to have 0 non-trivial immutable classes and 0 non-trivial all-mutating classes. Again, it is possible but unlikely that Mosh actually has immutable classes with some but not all methods `const`-annotated.

Discussion. Every class in Mosh has a mixture of mutating and immutable methods. Mosh contains a number of developer annotated `const` methods within 1% of the median across projects, and a low percentage of non-`const` methods that are easily `const`-able (6.3%).

Adoption. We initiated a pull request to the developers with our easily `const`-able methods, and they were interested in the possibility of including our tool as part of their continuous integration process.

F. Ninja

The Ninja build system creates a dependency graph and runs commands to rebuild targets when their dependencies change. Ninja outsources almost everything to other tools, e.g. calculations on the dependency graph. It focuses on its core functionality—the processing of the dependency graph and the selection of appropriate commands to call.

Discussion. Like Mosh, all classes in Ninja have a mixture of mutating and immutable methods. Unlike Mosh, the majority of methods in Ninja are non-`const` (78%). However, Mosh had the highest proportion of `const` methods that are easily `const`-able (72%). This may indicate that developers only annotated clearly-`const` methods and there are more opportunities for `const` in the codebase. Few non-`const` methods are easily `const`-able (8.5%) (i.e. few missed opportunities), suggesting that Ninja developers are diligent about using `const` in their project.

G. libsequence

The libsequence project is a library for evolutionary genetics. This library is primarily a collection of mathematical functions. Mathematical functions are a use case where we would expect mutation to be kept at a minimum.

For libsequence, Table VII shows 8 non-trivial immutable classes, as manually verified, accounting for 25% of all classes. However, libsequence has 0 unannotated classes: every class in libsequence has at least one `const`-labelled method. Our manual inspection, which was exhaustive in this case, did not find any missing `const` annotations on the unannotated classes nor any inappropriate `const` annotations on the immutable classes.

Discussion. Like OpenCV, libsequence contains many mathematical functions. The developers have also already `const`-annotated these functions. Our easily `const`-able analysis does not find many immutable functions for libsequence, so it would have been difficult to understand its immutability structure without the developer annotations.

H. Overall observations

Tables VIII (classes) and IX (methods) summarize our findings with respect to the research questions. Counts are from manually inspecting every class in the smaller projects and 60 randomly-selected classes across the larger projects (20 per project). For all projects but LLVM, our random sampling covers more than a third of the classes in that project.

RQ 1. How often do developers use C++'s flexible `const` annotations to indicate immutable classes and methods?

Finding 1. Over our projects, developers declared about 12% of all non-trivial classes as "Immutable". Across manually-inspected "Immutable" classes, we found that 96% of them were properly `const`-annotated. We found that 4% of non-trivial "Unannotated" classes we manually inspected were all-mutating. As for methods, 46% of methods were marked `const`.

RQ 2. What is the proportion of immutable methods and classes in typical C++ projects?

We used the developer-provided `const` label as an estimate of the ground-truth number of immutable methods and classes, and added an estimated number of unannotated methods that should be immutable.

Finding 2. Over our projects, a median 5% of non-trivial classes were immutable. Classes with all methods declared `const` ("Immutable") account for the vast majority of the immutable classes in the codebases; for Protobuf, we estimate the actual rate of non-trivial immutable classes is 7% higher than that of "Immutable" classes, but for the other projects, the proportion of "Immutable" classes is comparable to the number of classes with all methods `const`. Note, however, that 5% of non-trivial classes accounts for about 2% of all classes, which is quite few.

We found that developers used `const` on a median of 46% of methods. Furthermore, we estimate that at least 52% of methods (median) are in fact immutable. That is, of the potential methods that could have a `const` annotation, developers miss a median of 6%.

TABLE VIII
Class-level mutability as a % of the non-trivial classes count (more than 3 methods). "Immutable" refers to classes with all `const`-annotated methods and fields. Estimated counts add an expected contribution to the immutable count from "Unannotated" classes. A median of 5% of non-trivial classes are immutable, while a median of 2% are all-mutating.

	# non-trivial classes	% "Immutable" classes	% immutable classes (estimated)	% "Unannotated" classes	% all-mutating classes (estimated)
LLVM	5 842	10	11	9	4
OpenCV	2 221	6	5	8	2
Protobuf	228	29	36	16	4
fish	29	0	0	21	7
Mosh	47	0	0	0	0
Ninja	17	0	0	12	0
libsequence	16	50	50	2	0

TABLE IX
Method-level mutability as-estimated in our project set. Developers write far more immutable methods than immutable classes, with a median of 52% across our case studies. Compared to declared-immutable methods, developers could `const` annotate 6% (median) more methods.

	% immutable methods (estimated)
LLVM	52
OpenCV	53
Protobuf	63
fish shell	46
Mosh	51
Ninja	28
libsequence	93

Implications to language design. We formulated our research questions to guide immutability language design. RQ1 addresses in part how much developers could benefit from support for declaring immutable classes. Our results show that non-trivial immutable classes are present but rare across our case studies. We suggest that developers' needs are not served by simply adding support for immutable classes (as suggested by Glacier [5])—more is needed. Switching our focus from classes to methods, we found that developers do label methods as immutable—almost half, in fact. Yet, addressing RQ2 in part, we also found that even more methods could be labelled as immutable. The fact that developers merged some of our pull requests and asked to integrate our tool into their continuous integration systems suggests that tools to enable developers to add missing `const` annotations to their code would help.

Threats to validity. Our selection of projects poses a threat to external validity. As described above, we sought to minimize this threat by choosing among the most-starred GitHub projects (plus LLVM, which has its own ecoystem). We expect that our results are most applicable to large, open-source C++ codebases. Proprietary code could exhibit different characteristics from open-source code. We also tried to vary the projects as much as possible, choosing a mixture of executables and libraries.

The design of our easily-`const`-able analysis could pose a threat to construct validity, as it may undercount (but not overcount) the number of `const`-able methods. We ignore classes that inherit from a templated class, but not templates in general. When analyzing libraries independent of their clients, it is not possible to know the inherited class, so it is impossible to analyze all of its public methods.

VI. Related Work

We discuss related work in the areas of language feature usage surveys, type systems (particularly for immutability), and type qualifier inference.

Our easily `const`-able analysis finds methods that are free of writes to fields of the `this` object and any transitive state reachable from these fields. This is also known as deep concrete reference immutability [6]. We do not consider object immutability (an object does not mutate over its lifetime), but rather class immutability (every instance of an class does not mutate). In contrast to concrete immutability, abstract immutability allows mutations that do not affect the observable behaviour of the receiver object. Shallow immutability (vs deep) does not allow mutation to fields but does allow mutations transitively through fields. In shallow immutability, the identity of the pointee of a pointer field does not mutate, but the value contained in the pointee may change.

Usage of language features. Our work empirically surveys existing codebases to explore the developers' use of language features—in our case, C++ immutability and the `const` qualifier. Related work in this area is rare. Haller and Axelsson [7] investigate the prevalence of immutability in Scala classes (similar to part of our RQ1) for 4 Scala projects. They found that a majority of types in the Scala library are declared immutable; fewer for other projects. Richards et al [8] surveyed deployed JavaScript code to understand how the `eval` keyword is used in practice, and characterized the uses that they found. Our work shares with theirs the desire to understand code as it exists in the wild. A key difference between our work and theirs is that `const` could be elided without any immediate implications on software behaviour. On the other hand, `eval` makes code maintenance more difficult. Holkner and Harland [9] investigated the use of dynamic features in Python; Okur and Dig [10] investigated the use of parallel libraries in C# code; and Morandat et al [11] count uses of various features in R.

Our work is the first to study multiple large codebases with respect to their usage of existing immutability features, specifically at class and method granularities, as well as potential additional immutability annotations. Our work examines code without advocating for a specific type of immutability. One of our research questions investigated whether developers use immutable classes in their codebases today, as proposed in Glacier (see below). Immutable classes do exist and account for about one-tenth of nontrivial classes. Much previous immutability work typically introduces language support for a specific kind of immutability and argues that existing code uses, or can be modified to use, that kind of immutability; e.g. Gordon et al [12] describe reference immutability extensions to C# and summarize developer experience using these extensions on a large project.

Dynamic `const` usage. To better understand what developers hoped C++ `const` would guarantee, Eyolfson and Lam [13] investigated why developers used `const` while dynamically violating a notion of immutability stricter C++'s. That work focussed on observing program behaviour at runtime and how it compared to programmers' `const` declarations. Specifically, it searched program execution traces for so-called writes-through-`const`, when the program mutates an object's state through a `const`-qualified reference. The present work instead statically examines developers' design decisions as expressed in code, and when developers label members as `const`. We study classes' interfaces and implementations to understand which classes might be mutable or all-mutating, as well as class members that should have been `const`.

Other notions of immutability/purity. In this work, we investigate how developers use `const`, which is how C++'s type system supports immutability. Other languages have different notions of immutability. Scala and Rust have notions that are similar to C++'s: they support read-only references. Scala and Rust encourage read-only references more than C++ does. There is a subtle difference in vantage point between read-only references and immutability: read-only references are a property of client code (which is not allowed to modify the referred-to object), while immutability is a property of the class implementation. This work focusses on implementations' immutability. We consider an implementation which treats its reference to `this` as read-only to be immutable, although such an implementation could modify `this` through an alias.

While Java's support for immutability is limited to the `final` keyword, which prevents re-assignment of a variable or a field (but does not guarantee immutability of an object), researchers have proposed more sophisticated type systems for Java that guarantee (class) immutability. The Glacier system by Coblenz et al [5] adds one feature—transitive class-based immutability—to Java. In Glacier, an object that instantiates a class declared as `@Immutable` will have all fields immutable. All transitively reachable fields are also immutable. Glacier's immutable objects are completely immutable; Glacier does not contain the notion of a "mutable field" and prohibits transitive writes for immutable classes. Any method that our analysis identifies as `const`-able would also be immutable for Glacier. Glacier's validation is through a single case study where they converted existing code to be immutable.

Another Java-based immutability system is ReIm [14], which allows developers to label references as read-only references and infers method purity. Using a significantly more complicated inference system than ours, they found that 41–69% of methods could be marked `readonly`.

Our analysis identifies a subset of those found by purity analyses e.g. [14], [15]. Pure methods may write to freshly-allocated parts of the heap. Our analysis is conservative in declaring methods `const`-able and rejects any writes; more sophisticated analyses would allow unrelated writes.

Inference. In C++, the `const` qualifier serves two roles: it is a type qualifier (for fields and local variables) as well as an annotation for methods. Foster et al [16] inferred `const` type qualifiers for C programs and found that their case studies could have included many more `const` annotations than they did, consistent with our results. More recently, Greenfieldboyce and Foster [17] presented a technique for inferring type qualifiers for Java using their JQual tool. They apply JQual to inferring a variant of the version of `readonly` proposed by Javari [18]. We work at per-class and per-method granularities, determining whether a class is immutable or all-mutating, and whether a method should be `const` or not. Our work inferring `const` annotations is similar to the type inference performed by JQual, but we use an intraprocedural static analysis plus ad-hoc queries to determine whether a method is easily-`const`-able, rather than propagate type qualifiers across the entire program.

VII. Conclusion

We investigated the use of immutable classes and methods across 7 C++ open-source software projects, using both `const` annotations and static analysis to estimate the prevalence of immutability. We found that, on these projects, a median of 11% of non-trivial classes are immutable (or 2% over the entire population of classes). On the other hand, the number of immutable methods was much larger than the number of immutable classes. We found that developers declared a median of 46% of methods as immutable, and estimated that at least 53% of methods could be labelled as immutable (that is, in existing codebases, 6% of methods were missed opportunities for `const`). One project has already incorporated changes suggested by our tool; two projects have expressed interest in adding the tool to their continuous integration.

Artifact. We invite readers to try our tool themselves at `https://github.com/eyolfson/const-checker-artifact`. We have released our tool under an open source license, and provide a Vagrant setup with all required software.

References

[1] S. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, 3rd ed. Addison Wesley, 2005.

[2] P. T. Wilson, J. Pombrio, and S. Krishnamurthi, "Can we crowdsource language design?" in *Onward!*, Vancouver, Canada, 10 2017.

[3] ISO/IEC, "ISO International Standard ISO/IEC 14882:2014(E)—programming language C++." https://isocpp.org/std/the-standard, 2014.

[4] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization (CGO)*, Mar. 2004, pp. 75–88.

[5] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine, "Glacier: Transitive class immutability for Java," in *ICSE*, 5 2017.

[6] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull, "Exploring language support for immutability," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 736–747. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884798

[7] L. Axelsson and P. Haller, "Quantifying and explaining immutability in Scala," in *Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES '17)*, V.T.Vasconcelos and P. Haller, Eds., vol. EPTCS, no. 246, 2017, pp. 21–27, doi:10.4204/EPTCS.246.5.

[8] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do: A large-scale study of the use of eval in javascript applications," in *Proceedings of the 25th European Conference on Object-oriented Programming*, ser. ECOOP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 52–78. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032497.2032503

[9] A. Holkner and J. Harland, "Evaluating the dynamic behaviour of python applications," in *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ser. ACSC '09. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2009, pp. 19–28. [Online]. Available: http://dl.acm.org/citation.cfm?id=1862659.1862665

[10] S. Okur and D. Dig, "How do developers use parallel libraries?" in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 54:1–54:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393660

[11] F. Morandat, B. Hill, L. Osvald, and J. Vitek, "Evaluating the design of the R language: Objects and functions for data analysis," in *Proceedings of the 26th European Conference on Object-Oriented Programming*, ser. ECOOP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 104–131. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31057-7_6

[12] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy, "Uniqueness and reference immutability for safe parallelism," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. New York, NY, USA: ACM, 2012, pp. 21–40. [Online]. Available: http://doi.acm.org/10.1145/2384616.2384619

[13] J. Eyolfson and P. Lam, "C++ const and immutability: An empirical study of writes-through-const," in *European Conference on Object-Oriented Programming (ECOOP)*, 2016, pp. 8:1–8:25.

[14] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst, "ReIm & ReImInfer: Checking and inference of reference immutability and method purity," in *OOPSLA*, 2012.

[15] A. D. Sălcianu and M. Rinard, "Purity and side-effect analysis for Java programs," in *VMCAI 05*, 1 2005, pp. 199–215.

[16] J. S. Foster, M. Fähndrich, and A. Aiken, "A Theory of Type Qualifiers," in *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, 5 1999, pp. 192–203.

[17] D. Greenfieldboyce and J. S. Foster, "Type qualifier inference for Java," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, 2007, pp. 321–336. [Online]. Available: http://doi.acm.org/10.1145/1297027.1297051

[18] M. S. Tschantz and M. D. Ernst, "Javari: Adding reference immutability to java," in *OOPSLA*, Oct. 2005, pp. 211–230.