C++ `const` and Immutability: An Empirical Study of Writes-Through-`const`

Jon Eyolfson[1] **and Patrick Lam**[2]

1 University of Waterloo
Waterloo, ON, Canada
`jeyolfso@uwaterloo.ca`

2 University of Waterloo
Waterloo ON, Canada
`patrick.lam@uwaterloo.ca`

Abstract

The ability to specify immutability in a programming language is a powerful tool for developers, enabling them to better understand and more safely transform their code without fearing unintended changes to program state. The C++ programming language allows developers to specify a form of immutability using the **`const`** keyword. In this work, we characterize the meaning of the C++ **`const`** qualifier and present the ConstSanitizer tool, which dynamically verifies a stricter form of immutability than that defined in C++: it identifies **`const`** uses that are either not consistent with transitive immutability, that write to mutable fields, or that write to formerly-**`const`** objects whose **`const`**-ness has been cast away.

We evaluate a set of 7 C++ benchmark programs to find writes-through-**`const`**, establish root causes for how they fail to respect our stricter definition of immutability, and assign attributes to each write (namely: synchronized, not visible, buffer/cache, delayed initialization, and incorrect). ConstSanitizer finds 17 archetypes for writes in these programs which do not respect our version of immutability. Over half of these seem unnecessary to us. Our classification and observations of behaviour in practice contribute to the understanding of a widely-used C++ language feature.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases empirical study, dynamic analysis, immutability

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.

1 Introduction

Immutability is an important concept that simplifies reasoning about programs and eases software maintenance. Most importantly, immutability circumscribes possible side effects, so that (in some cases) a user of a function may avoid closely examining the implementation of the function and its callees. One concrete application of immutability is: if a developer knows that a library function does not modify one of its arguments (including transitive arguments), then they know that it is safe to call that library function with that argument from multiple threads, as the function only requires read access to its argument.

C++ [6] is a popular language that allows programers to specify immutability using **`const`**[1]. C++ experts such as Meyers recommend judicious use of "**`const`**-correctness" [7] in C++ codebases. It is generally clear when developers could use **`const`**, but not (in non-obvious cases) when they should use **`const`**.

[1] Some of our discussion also applies to C, but we focus on C++ in this paper.

We distinguish between two uses of C++'s `const` qualifier: `const`-qualified global/stack objects, whose data may never change (i.e. immutable objects), and `const`-qualified references/pointers to objects (i.e. read-only references [9]). (For now, assume that there are no casts that remove `const` qualifiers and no `mutable` storage class specifiers; these language features violate immutability.) An *immutable object*'s fields may never change, and mutable references to that object may never be created. A *read-only reference*, on the other hand, only guarantees immutability for accesses through qualified references. An object with a read-only reference to it may still be mutated through other, mutable, references. C++ enforces a shallow immutability guarantee (also known as bitwise `const`) for writes through the read-only reference: while it is illegal to reassign the fields of such an object, any referents of fields may change. We expand on this in Section 3.

C++'s type system admits several workarounds to `const`'s supposed immutability guarantees. Much research defines type qualifiers similar to C++ `const`, but with stronger guarantees. These type systems do not have any holes in the type system, such as unsafe casts. Furthermore, they not only ensure that the field values do not change, but also ensure that objects referenced through fields also do not change (i.e. deep, or transitive, immutability; again, see Section 3 for more details).

On the other hand, our industrial contacts have indicated that, in their codebases, `const` has been used to deter new developers from modifying certain variables. Such variables may be modified by an experienced developer ready to assume the consequences [4]. In such cases, `const` serves an advisory role, but does not provide any guarantees.

Our goal is to explore the space of possible meanings for immutability declarations in C++ and to examine what guarantees developers appear to be expecting in practice. We developed a tool, ConstSanitizer, that instruments programs to identify source locations that modify `const`-qualified objects, using more restrictive semantics than guaranteed by C++. ConstSanitizer monitors writes-through-`const`, i.e. writes performed on `const`-qualified objects or references, either transitively (which is allowed in C++), or through C++'s `const` escape hatches. To better understand `const` usage in practice, we ran ConstSanitizer on a benchmark suite and manually classified all writes-through-`const`.

Specifically, the goal of this work is to answer the following research questions:

(**RQ1**) Do developers perform shallow and transitive writes-through-`const`?
(Answer: Yes to both.)

(**RQ2**) How do developers write-through-`const`?
(Answer: By directly writing to fields of `const`-qualified objects and through transitive writes; both are about equally common.)

(**RQ3**) Why do developers write to fields of `const`-qualified objects?
(Answer: Buffers and delayed initialization were important reasons, but over half the time, we couldn't find any clear reason motivating developers' decisions to write through `const`.)

Section 8 presents our detailed answers to these questions.

Our contributions include:

- the design and implementation of a novel dynamic analysis for C++ that detects writes-through-`const`-qualified variables (both shallow and transitive);
- an empirical study of `const` usage (including writes-through-`const`-qualifiers) on a suite of 7 C++ benchmarks; and,
- based on the empirical study, a novel classification of writes-through-`const`-qualifiers in the wild according to a root cause and a set of attributes.

2 Motivating Example

We continue with an example of a `const` usage that must be accepted by C++ compilers [6] but leads to undefined behaviour when used with the C++11 standard library specification.

Listing 1 contains function `writeId()`, which modifies field `id` of its parameter. Function `evil()` takes a `const`-qualified parameter, casts away the `const` qualifier, and calls `writeId()`. Both these functions must be accepted by C++ compilers.

Listing 1 Method `evil()` violates the spirit of `const` by causing a write to an externally-visible field of `const` object "`a`". Circled numbers used for subsequent explanations.

```
    class A { public: int id; };
    size_t std::hash(const A& a) { return std::hash(a.id); }
    std::unordered_map<A, std::string> m;
①  const A a;

    m.insert(a, "Value");
②  evil(a);                                       ③
    m.find(a);

    void evil(const A& a) { writeId(const_cast<A*>(&a)); }
④  void writeId(A *pa) { pa->id = 5; }
```

Function `writeId()` does not perform anything unexpected. The write to `id` is legitimate with respect to `const`: `writeId()` has a non-`const` reference to data and is therefore entitled to write to it. (`writeId()` conflicts with a different part of the library specification—one oughtn't write to fields used as hashes—but that conflict is beyond the scope of this work.)

Function `evil()` accepts a `const` parameter "`a`"; the `const` qualifier intuitively suggests that the state of "`a`" should not change across a call to `evil()`. (Section 3 explains the C++ semantics of `const` in more detail; Section 4 explains the writes that ConstSanitizer monitors.) `evil()` then casts away the `const` qualifier and calls `writeId()`, which writes to the `id` field, thus changing the state of "`a`".

Listing 1 also contains client code. This client code provides a `hash()` function for when objects of type "`A`" are used with the standard library. It continues with a declaration of an `std::unordered_map m`, which gets a `const`-qualified object "`a`" added to it. (In C++, objects like "`a`" and `m` are constructed upon declaration.) Between the `insert()` call and the `find()` call, the program calls `evil()`, which changes the `id` field, thus causing the `hash()` of "`a`" to change. As a result, the `find()` call may unexpectedly return `m.end()`, indicating that it did not find "`a`" in the expected bucket; that result should be a surprise to the client.

In our example, the client code is doing nothing wrong, yet it may get an unexpected result from the map. The client should be entitled to believe that its `const`-qualified "`a`" object does not change between the two map calls and that the object is still in the map.

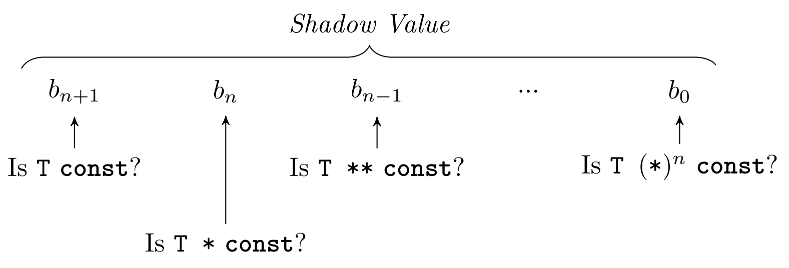


Figure 1 Our shadow values encode the `const`-ness of each level of an n level pointer.

Analysis of example. We informally describe how ConstSanitizer works on our example. Our LLVM-based tool actually operates at the `llvm` bitcode level (assisted by metadata from `clang`), but for clarity we describe shadow values and the effects of statements on them using C++ code. Section 4 describes our analysis as-implemented on top of LLVM.

ConstSanitizer associates a shadow value with each variable. This shadow value describes whether or not the variable, and each of its dereferences, may be written to; Figure 1 shows how shadow values encode **`const`**-ness. We present the semantics of our shadow encoding more thoroughly in Table 1. ConstSanitizer propagates shadow values through the program's execution. At each write, ConstSanitizer verifies that the shadow value permits a write, and indicates a write-through-**`const`**-qualifier if not.

We next show how ConstSanitizer works; circled numbers refer back to Listing 1.

① Program allocates **`const`**-qualified new object "`a`". Using debug information, ConstSanitizer finds the **`const`** qualifier in "`a`"'s type, and associates shadow value $(1)_2$ with "`a`", indicating that "`a`" is **`const`**.

② ConstSanitizer then propagates shadow value $(1)_2$ to the function call `evil()`. Inside function `evil()`, variable "`a`" is a reference, so we shift the shadow value to the left and obtain $(10)_2$ inside the function call boundary.

③ Program casts from type `const A*` to type `A*`. Because "`a`" is a reference inside `evil()`, the address-of operation has no effect on the shadow value. (On a non-reference, taking the address would also result in a logical shift left of the shadow value.)
The cast of the **`const`** qualifier is invisible to LLVM, so ConstSanitizer propagates shadow value $(10)_2$ across the cast. (Had "`a`" originally not been **`const`**, a pointer to it would have shadow value $(00)_2$ and it would have shadow value $(0)_2$.)
Finally, ConstSanitizer passes shadow value $(10)_2$ for parameter "`pa`" of `writeId()`.

④ Inside function `writeId()`, the instrumented write to field "`id`" observes that the shadow value of the address of the containing object is $(10)_2$. Because the left-hand side uses the `->` operator, ConstSanitizer shifts the shadow value back right once, giving shadow value $(1)_2$ for the containing object. Because the containing object is **`const`**, we also apply a shadow value of $(1)_2$ to all writes to fields of that object.
When executing the write, ConstSanitizer checks the right-most bit of the shadow value for the destination. Since this value is $(1)_2$, the program is writing a value through a **`const`** reference. ConstSanitizer therefore signals a write-through-**`const`**.

Section 5 contains our classification of writes-through-**`const`**. We classify this write as root cause "C", a write after casting away a **`const`**, and assign attribute "I", for incorrect.

3 Meaning of const in C++

As discussed earlier, the C++ **`const`** keyword allows programmers to declare, in some sense, that a value should not change. In this section, we explain the specific guarantees that C++ provides, and we present the deep immutability variant of these guarantees that we verify.

Meaning of `const` on C++ primitive and pointer types. The meaning of **`const`** in C++ is an extension of its meaning in C. We start by describing the common meaning of **`const`** across C and C++, applying to primitive and pointer types. In these languages, the **`const`**-ness of a memory location depends on the qualifiers of the variable through which the location is accessed; our motivating example illustrated a change in **`const`**-ness through a cast, in function `evil()`, that is allowed by both C and C++.

Developers may **`const`**-qualify primitive types such as **`int`**, resulting in immutable object types like **`const int`**. When variable v has primitive **`const`**-qualified type, the C++ type system prevents developers from assigning to v after its definition; i.e. it prevents writes to v. In this case, **`const`** behaves like **`final`** in Java, which also prevents re-assignment.

For pointer types, such as **`int *`**, developers may **`const`**-qualify both the pointee and pointer type. The **`const`** qualifier applies to the type directly to the left of it; if there is nothing to the left, then **`const`** applies to the right. If a variable has type **`int *const`**, developers may not change the address value of the variable (where it points to), but they may dereference the location and change the value it points to. A different type is **`int const *`** (also known as **`const int *`**), which allows the address value to change, but not the value pointed-to by the variable. This type represents a read-only reference. The **`const`** qualifier may also apply to both the pointer and pointee types (**`int const * const`**), which prevents writes to both the address value and the value pointed-to. If all pointers to a value are read-only references, then the value pointed-to is immutable. C++ references can be thought of as **`const`**-qualified pointers; a developer may not write to the reference address value. However, in contrast with pointers, developers cannot cast away reference address value **`const`**-ness and re-assign the address value; this property is enforced by the language.

Listing 2 The `const` qualifier may apply to C++ member functions.

```
class Pointish {
private:
  int x;
  int * y;
public:
  int getX() const { return x; }
  void setX(int val) { x = val; }
  void setY(int val) { *y = val; }
};
```

not OK if `setX()` were `const`

transitive write
OK if `setY()` were `const`

Meaning of `const` on C++ object types. We continue by exploring the meaning of **`const`** in C++-specific contexts. When a C++ object type is **`const`**-qualified, the developer may only call member functions declared with a **`const`** qualifier.

`const`-qualifying a member function has two effects. First, **`const`**-qualifying a member function allows it to be called on a **`const`**-qualified receiver object. Furthermore, inside the function, the type qualifiers of the receiver object's fields are treated as **`const`**.

Conceptually, each C++ class provides two interfaces: the **`const`**-qualified interface and the non-**`const`** qualified interface. A **`const`**-qualified reference is meant to be a read-only reference, although C++ enforces no guarantees. One of our goals is to evaluate whether read-only guarantees hold in practice. When an object has non-**`const`** type, then the developer may call all methods on that object[2]. On the other hand, when an object has **`const`** type, then the developer may only call methods on that object that are **`const`**-qualified.

Consider class `Pointish`, defined in Listing 2. As written, the developer could call all methods on a non-**`const`**-qualified object of type `Pointish`. On the other hand, the developer may only call the `getX()` method on a **`const`**-qualified `Pointish` object.

The second effect of **`const`**-qualifying a function changes the type qualifiers of fields inside the function. In our example, field `x` becomes **`int const`** within **`const`**-qualified member

[2] There is a small exception: on a non-**`const`** object, the developer cannot call **`const`**-qualified methods that are hidden due to overloading by a non-**`const`**-qualified method of the same signature.

functions. The compiler successfully compiles `getX()`, since there are no writes to `x` or `y`. But, if `setX()` was `const`-qualified, the compiler would refuse to compile the code, since the type of `x` would be treated as `const int` and `setX()` contains a write to that variable.

In C++, without using `const` escape hatches, developers may re-assign fields in non-`const` qualified methods and may not re-assign fields in `const`-qualified methods. In all methods, developers are permitted to mutate state outside of re-assignment (through references or pointers). This type of immutability is referred to as *shallow immutability*.

A C++ `const`-qualified stack/global object would be considered a shallow *immutable object*. That is, without escape hatches, developers cannot create non-`const` references (including through pointers) to such a `const`-qualified object. However, as we discuss next, developers may indeed remove the `const` qualifier on references to the `const`-qualified object. Therefore, C++ does not strongly enforce the concept of an immutable object.

Working around `const` restrictions. Practical type systems appear to require escape hatches. C++'s escape hatches for `const` include casting (the sole escape hatch in C) and `mutable`. Also, C++ `const` does not specify deep immutability. ConstSanitizer dynamically observes executions to monitor uses of escape hatches and deep immutability.

Most type systems permit casting between types. C-style casts (`(const A)a`) and C++ `const_cast`s can add or remove `const` qualifiers. `const` manipulation may also occur through unions and `reinterpret_cast`s. ConstSanitizer ignores casts, instead using the declared type of a variable or function argument. When there is a mismatch between variable and function argument types, we persist all `const` information.

For `const` member functions, "`mutable`" instructs the compiler to not add the implicit `const` type qualifier otherwise imposed on fields inside those functions. In Listing 2, if `x` were instead declared as `mutable int x`, then `setX()` could be `const`-qualified and still compile. ConstSanitizer would then report the write to field `x` whenever `setX()` was called on a `const` receiver object, as we consider the write to be a breach of that object's immutability.

ConstSanitizer goes beyond C++'s guarantees with respect to transitivity. Consider again Listing 2. Field `y` has type `int *`; within a `const`-qualified member function, its type is `int *const`. C++ therefore prevents writes to `y` inside a `const`-qualified member function. However, it does not prevent writes to `*y` without further explicit markup (i.e. `int const *const`). We call writes to locations like `*y` *transitive writes*.

C++ only guarantees shallow immutability—that is, field values directly stored in a `const`-qualified class do not change. If a field has pointer type, C++ ensures that the pointer value does not change, but does not guarantee anything about the value pointed to. ConstSanitizer verifies *deep immutability* through transitive writes, and enables us to answer the empirical question of whether extant programs preserve deep immutability or not.

4 Technique

Our ConstSanitizer tool generates instrumented code which, when executed, prints out notifications about writes-through-`const`-qualifiers[3]. ConstSanitizer builds upon LLVM [16] and was inspired by existing sanitizers including AddressSanitizer [13] and MemorySanitizer [14].

[3] We previously implemented a static analysis which was an unpublishable dead end. Most of its reported violations required calling context to make sense of; context-sensitive interprocedural analysis would thus be required to get meaningful results. Static counts of mutables and const casts were too overwhelming. Furthermore, imprecision due to pointers made its results unusable.

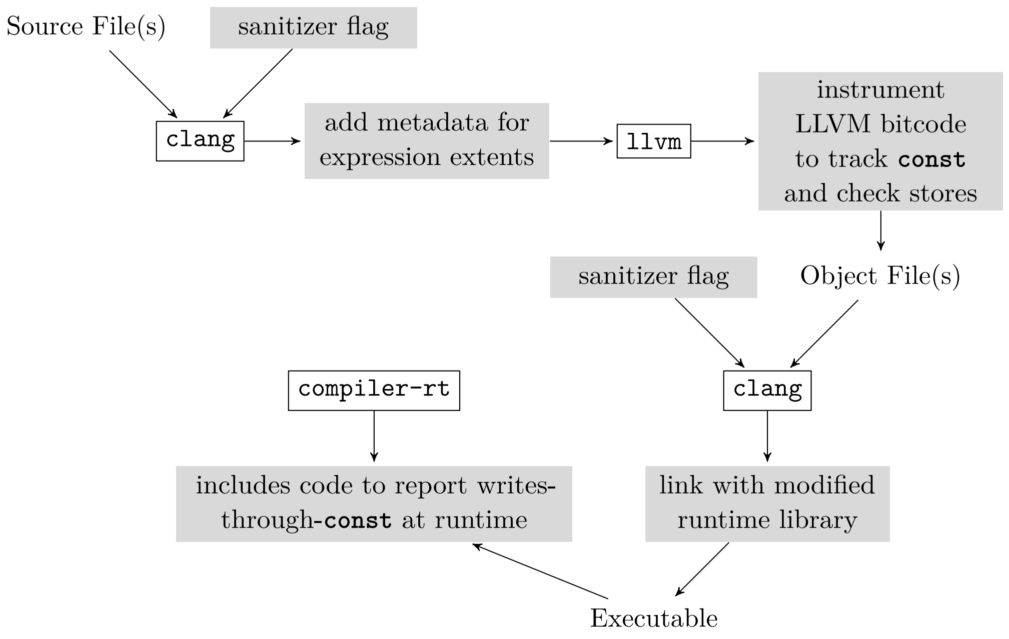

Source File(s)
sanitizer flag
clang
add metadata for expression extents
llvm
instrument LLVM bitcode to track const and check stores
sanitizer flag
Object File(s)
compiler-rt
clang
includes code to report writes-through-const at runtime
link with modified runtime library
Executable

Figure 2 ConstSanitizer generates instrumented LLVM bitcode which reports writes through `const` qualifiers at runtime.

We implemented ConstSanitizer by extending the `clang` frontend and adding instrumentation passes on `llvm` bitcode. The instrumented code calls hooks in our modified version of `llvm`'s `compiler-rt` runtime library. Figure 2 depicts our processes for compiling instrumented code. Plain text indicates inputs and outputs; outlined boxes indicate existing software components; and light gray boxes indicate our modifications.

We first describe our modifications to the `clang` frontend. When the developer enables ConstSanitizer (using a command-line flag), our frontend adds metadata about initialization expression extents to the bitcode. This metadata notifies the `llvm`-level instrumentation about source-level constructs that would otherwise be lost in translation to bitcode.

Specifically, we modified `clang`'s bitcode generator at variable declaration statements. At statements of the form `type var = expr` we mark the instructions making up `expr` so that our `llvm`-level instrumentation can ignore them. The rationale for ignoring those writes is that the primary user-visible write from a `clang` declaration statement is to `var`, on the left-hand side. We empirically observed that other writes within `expr` are almost always initialization writes to `var`, which ought not to be reported even if `var` is `const`. Although a programmer may include explicit (side-effecting) writes within `expr`, we ignore such writes to eliminate the false positives that otherwise occur due to initialization writes.

We use `const`-ness information as provided by declarations, rather than implementing a taint-based approach. Listing 3 shows a false negative caused by our approach. The debugging information for `y` gives shadow value $(00)_2$. Hence, on the write through `y`, we do not report a write-through-`const`, because we do not propagate `const`-ness information from variable initializers. One might expect this write to trigger a report since `y` aliases the read-only reference `x`. (We report a write if the cast is part of a function argument.)

Most of our instrumentation lives in a custom `llvm` pass that generates code to track `const`-ness of the program's values. The instrumentation manipulates shadow values to track `const` qualifiers at every instruction that generates a pointer value. The `const` information

Listing 3 C++ source code showing a false negative due to expression handling.

```
const int * x = new int(0);
int * y = const_cast<int *>(x);
*y = 1;———————————————————————— not reported
```

relies on type tables from DWARF 4 debugging information. In `llvm`, this includes all variables in functions—all local variables are allocated on the stack and are pointers.

Our dynamic analysis returns (as one might expect) no false positives, since it observes program executions. However, it depends on the accuracy of the debugging information and metadata, which it uses to identify which variables are **`const`**-qualified in the source and to identify initialization expression extents. We ran into one false positive in our results which we believe is the result of the metadata being invalidated between LLVM passes. Throughout the remainder of the section, we point out a couple of cases where our analysis must approximate intended **`const`**-ness because actual **`const`**-ness information does not exist.

Structure of shadow values. A shadow value consists of n bits tracking **`const`**-ness (where n is the word length of the processor architecture). Each bit represents whether a pointer or pointee has a **`const`** qualifier or not. The rightmost bit represents the **`const`** qualifier of the value itself. Bits to the left (if the value is a pointer) represent what the pointer transitively points to. Our encoding supports pointers up to $n-1$ levels deep on n-bit processors (64 for our experiments). Figure 1 depicted our encoding of shadow values, while Table 1 shows how shadow values represent sample **`const`**-ness settings and corresponding writes allowed.

Shadow value computation. We next describe how we create and propagate shadow values. Our ConstSanitizer instrumentation dynamically propagates shadow values representing **`const`** qualifiers through a program's instructions. Our goal is to monitor 1) writes to **`mutable`** fields; 2) locations where **`const`** has been cast away; and 3) transitive writes, to pointees of fields, through **`const`** references. Table 2 summarizes the analysis rules.

`llvm` bitcode uses `alloca` instructions to introduce new pointer values. Our `llvm` pass instruments each `alloca` instruction with the appropriate shadow value, as extracted from the type information in the source code, using standard `clang` debug information.

Ultimately, our instrumentation verifies the behaviour of `store` instructions. Recall that we exclude `store` instructions that come from the right-hand side of a declaration

Table 1 Shadow values encode available **`const`**-ness restrictions on variables.

Declaration	Shadow value	Example statement	Allowed
`int x`	$(0)_2$	`x = 5`	✓
`const int x`	$(1)_2$	`x = 5`	×
`int * x`	$(00)_2$	`x = y`	✓
		`*x = 55`	✓
`int *const x`	$(01)_2$	`x = y`	×
		`*x = 55`	✓
`const int * x`	$(10)_2$	`x = y`	✓
(or `int const * x`)		`*x = 55`	×
`const int *const`	$(11)_2$	`x = y`	×
(or `int const *const`)		`*x = 55`	×

statement. For all other `stores`, we check whether the operand—the location being written-to—represents a `const`-qualified type. The rightmost bit of the shadow value provides this information. If that bit is 1, an execution of this `store` instruction is a write to a `const`-qualified location. We insert a call to our runtime library to check the value of the bit and to report a write-through-`const` if the bit is 1. (We later discuss a special case for store instructions where the value being stored is a function argument.)

Conversely, `load` instructions return a pointer that represents a single pointer dereference. To compute the returned shadow value, we right shift the operand's shadow value.

`llvm`'s `getelementptr` (GEP) instruction accesses arrays and fields of objects. This instruction preserves type safety through dereferences in the compilation process and is a safe alternative to directly generating pointer arithmetic code. Our instrumentation performs a logical shift right by one bit for every pointer dereference implied in the GEP instruction. Our treatment of GEP implicitly handles transitive immutability as follows: when a GEP

Table 2 Dynamic analysis rules showing computation of shadow value for result `%1`.

Instruction	New shadow value
`%1 = alloca ...`	from `const` qualifiers in debugging information, consistent with Figure 1.
`%1 = getelementptr %2`	by logically shifting left `%2`'s shadow value once for each dereference this instruction represents. **if field access**: check `const` qualifier of base object; for (immutable) `const` base objects, new shadow value is all ones, otherwise all zeros.
`%1 = call(%2)`	loaded from return shadow value in Thread-Local Storage (TLS). **for pointer arguments `%2`:** also write shadow values to appropriate TLS slots for the function call; if the call and argument are marked as ignored, write all zeros for the shadow value for the argument.
`%1 = phi/select ...`	carry out same operation on shadow value operands.
`%1 = bitcast %2`	from the shadow value for `%2`, if compatible; otherwise all zeros.
`%1 = load %2`	logical shift right of `%2`'s shadow value.
`store %2, %1`	check rightmost bit of shadow value for `%1`, report write-through-`const` if set. (Only applies if the instruction not ignored as an initializer.) **if `%2` is function argument:** load shadow value for `%2` from TLS, left shifted once. New shadow value is bitwise OR of shifted value with previously computed shadow value for `%1`. **if `%2` is "this" function argument:** same steps as above, except skip the bitwise OR step. **if `%2` is "this" function argument for destructor:** shadow value for `%1` is $(00)_2$.
`%1 = extractelement` `%1 = extractvalue` `%1 = inttoptr` `%1 = landingpad`	all zeros.

accesses an object field, and the containing object is **const**-qualified, we generate a shadow value as if the field had a **const** qualifer on every type for the contained field. This treatment implies checks for transitive immutability; generating a non-**const** shadow value here would generate the same bitwise immutability checks for **const** as specified by the C++ standard.

Our instrumentation propagates **const**-ness information (in shadow values) alongside references to that location. In C++, access restrictions to a location depend on whether the program is accessing that location through a **const** reference or not. Therefore, in the presence of casts and pointer arithmetic, there is no ground truth about the **const**-ness of the resulting references and we must make a reasonable under-approximation as to **const**-ness.

We next discuss casting-related `llvm` instructions. The `bitcast` instruction converts a value into a specified type. If a program converts a pointer between equally-indirected pointer types, then we copy the old shadow value to the result. (C++ **const**-casts do not appear at LLVM bitcode level, nor do the component of a C-style cast that manipulates **const**-ness. ConstSanitizer preserves declared **const**-ness for such variables.) Otherwise, we choose to assume that the instruction's result has no **const** qualifiers. We make this assumption in all cases for the `inttoptr` instruction, which represents pointer arithmetic not handled by the GEP instruction, as well as for `extractvalue` and `extractelement`.

Our instrumentation stores shadow values for function calls' arguments and return values using thread local storage (TLS). In the straightforward case, we store shadow values for pointer arguments in TLS slots reserved for each argument. However, we ignore pointer arguments' **const** qualifiers if the call and the argument are both part of a variable declaration. As for (pointer) return values: if a function was instrumented by our tool, then we read the shadow value from the appropriate TLS slot. We also store a mutable shadow value in the return value TLS slot, in case the function had not been instrumented by our tool. Our instrumentation either reads the approximation or, if applicable, the actual return value generated by the called function. In the presence of callbacks from uninstrumented code back to instrumented code, our instrumentation may use stale shadow values and report extraneous results based on these stale shadow values.

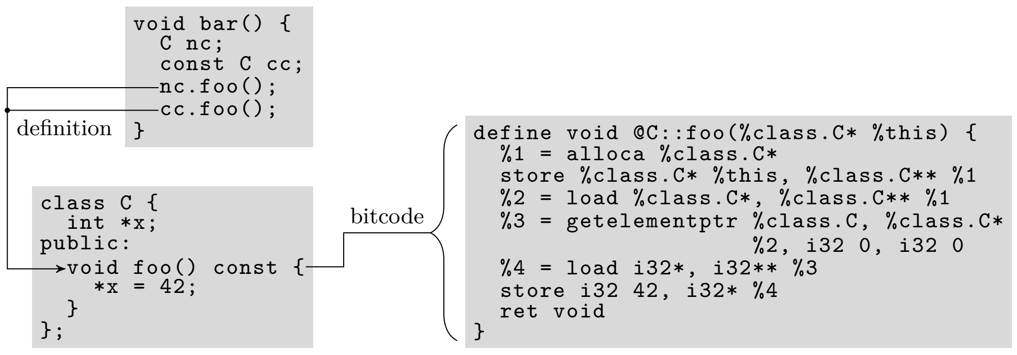
We have a special case for store instructions where the value stored is a function argument, as mentioned above. Consider a store instruction "`store value, location`". We compute the shadow value for "`location`" as follows. First, we get the shadow value for "`value`" from the TLS. Then, we adjust this shadow value to be compatible with the type of "`location`": our encoding requires one logical shift to match the type of "`value`" to that of "`location`". We bitwise OR the shadow value for "`location`" previously computed (from an `alloca` instruction) with the shifted value to get the new shadow value for "`location`". This preserves **const** qualifiers of the original argument and of the local variables in the function.

There are two further special sub-cases for store instructions and function arguments for (i) method and (ii) destructor calls. Listing 4 illustrates sub-case (i). Here, `foo()` is declared **const**. The compiler will hence treat `this` as **const** within `foo()`. However, for our dynamic analysis, we want to detect writes based on the **const**-ness of `this` from the caller; in method `bar()` in Listing 4, receiver object `nc` for `foo` is not **const**, so we do not want to report the call's (transitive) store to `x`. `foo`'s method arguments appear as "`value`" operands of `store` instructions while the "`location`" is an `alloca` within the function. We set the shadow value of the associated `alloca` instruction to the value of the argument after applying a logical shift left by one (since it's a pointer). This treatment properly ignores **const** qualifiers added due to callee method signatures.

For sub-case (ii), destructors, we do not want to report any writes through `this` as the object no longer exists after the call (so that writes to the object aren't visible in any case).

We handle this case by simply assuming that the `this` argument is mutable. For all other arguments, we do a bitwise OR between the `alloca` shadow value and the argument shadow value logically shifted left by one, which maintains all `const` qualifiers.

Listing 4 C++ source code showing calls to method `foo()` (with its definition and associated LLVM bitcode) from `const` context `cc` and non-`const` context `nc`.



```
void bar() {
  C nc;
  const C cc;
  nc.foo();
  cc.foo();
}
```

```
class C {
  int *x;
public:
  void foo() const {
    *x = 42;
  }
};
```

```
define void @C::foo(%class.C* %this) {
  %1 = alloca %class.C*
  store %class.C* %this, %class.C** %1
  %2 = load %class.C*, %class.C** %1
  %3 = getelementptr %class.C, %class.C*
                     %2, i32 0, i32 0
  %4 = load i32*, i32** %3
  store i32 42, i32* %4
  ret void
}
```

Shadow value computation example. Listing 4 presents the C++ source code for `C::foo`, a `const` qualified method, and the associated LLVM bitcode. Consider the `bar` function, which calls `foo` twice, first with mutable (i.e. non-`const`) receiver object `nc` and then with `const` receiver object `cc`. Within `bar`, the shadow value of `nc` is $(0)_2$ and the shadow value of `cc` is $(1)_2$. Our instrumentation assigns shadow values for each LLVM instruction with a pointer result. We instrument `C::foo` as follows:

- The first instruction, `alloca`, stores its result in `%1`. Since it is an `alloca` instruction, we obtain its shadow value from `clang` debugging information. The associated shadow value is $(10)_2$: in this `const`-qualified method, the type of `this` is that of a `const` pointer to the containing class, `const C *`.
- At the `store` instruction, without special handling, we would load the shadow value of argument `%this` from the TLS; logically shift left the shadow value by one to account for the fact that we are performing a `store` to memory allocated for that argument; and bitwise OR the resulting shadow value with the original shadow value for `%1`. In our example, whether the receiver object is `cc` or `nc`, the shadow value for `%1` is $(10)_2$.
- Next, we obtain the shadow value for the result of the `load`, `%2`. As `%2` returns a pointer, we shift `%1`'s (the operand's) shadow value right by one, giving a shadow value of $(1)_2$.
- Next, the `getelementptr` instruction results in a pointer to the class's `x` field. Our instrumentation of `getelementptr` could produce two different shadow values, depending on the instruction's operand. In this case, `%2` is a `const` object, and the resulting shadow value for a fully `const`-qualified `x` field is $(11)_2$.
- Next, we obtain the shadow value for the `load` result `%4` using the same technique as for `%2`. The resulting shadow value is $(1)_2$.
- Finally, we insert a check at the `store` instruction. In this case, the least significant bit of the shadow value associated with location (`%4`) is 1. Therefore we would dynamically report a write-through-`const` at the write to field `x` of the `const` method.

For methods, this instrumentation is not enough. We only want to report a write-through-`const` for the call with `const` receiver object even though both objects call the same static method. Before the call to `foo`, our instrumentation stores the shadow value of the receiver object in its TLS slot. Our instrumentation of `foo` looks for `store` instructions that use the receiver object and recomputes the shadow value of the location. Here, we load the shadow value from its TLS slot and shift left to match the type of the expected shadow value of `%1`. For `nc`'s call, this shadow value is $(00)_2$. Since `foo` is a method, we ignore the original shadow value of `%1` ($(10)_2$) and overwrite it with new shadow value $(00)_2$. Following the remaining steps in `foo` as above, the shadow value of `%4` is now $(0)_2$ and we do not report a write-through-**`const`**. In the `cc` case, we would follow the same steps, but instead report a write-through-**`const`**, because the shadow value would be $(10)_2$.

5 Classification of writes-through-const

One of our contributions is a careful analysis of the **`const`** usages detected by our ConstSanitizer dynamic analysis tool. We propose a classification for writes-through-**`const`**-qualifiers along 2 axes. We manually assigned each write 1) a single cause, from a set of common root causes; and 2) a set of additional attributes. This classification distills our empirical observations about **`const`** use in practice.

Table 3 Root causes of writes through **`const`** and our symbols for these causes.

Root Cause	Symbol
Write to mutable field	M
Transitive write	T
Write after casting a **`const`** qualifier away	C

Table 3 lists all of the root causes for writes-through-**`const`**, along with a one-letter abbreviation that we will use in Section 6's tables. ConstSanitizer detects such writes and reports them to the user. The causes are:

- mutable field (M): the program writes to a **`mutable`**-labelled field of a **`const`** object.

```
class Mutable {
  mutable int x;
public:
  void mutator() const { x = 42; }
};
```

  `mutable` permits method `mutator()` to write to field `x` even though it is a **`const`** method, which would ordinarily prevent (at compile-time) writes to fields of the **`this`** object.
- transitive write (T): the program writes through a field of a **`const`** object.

```
class TW {
  int *x;
public:
  void transitiveWrite() const { *x = 42; }
};
```

  `const`-qualified method `transitiveWrite()` writes to field `x` of the **`this`** object. While the **`const`** qualifier prevents mutation of the `x` field, it does not prevent transitive writes of the memory pointed to by `x`.
- casting away const (C): the program writes through a pointer which has previously been **`const`** but whose **`const`**-ness has been cast away using a `const_cast` or C-style cast.

```
void writeToArg(int *y) { *y = 17; }
const int *x = ...;
writeToArg(const_cast<int *>(x));
```

The write in `writeToArg()` mutates the value pointed-to by `x` while `x` is `const`-qualified. ConstSanitizer reports writes-through-`const`-qualifiers whose `const`-ness has been cast away, using the `const`-ness of the most recent declared type for the value.

Table 4 Observed common attributes of writes through `const` and corresponding symbols.

Attribute	Symbol
Write is synchronized	S
Write is not visible	N
Write is to a buffer/cache	B
Write is delayed initialization	D
Write is incorrect	I

Table 4 summarizes our attributes for writes-through-`const`-qualifiers. We assigned attributes to writes based on our understanding of the code. Writes may have multiple attributes; for instance, a write in our Protobuf benchmark is B & N & S. The attributes are:

- synchronized (S): indicates that the write is always protected by a lock. This attribute is often required under the C++11 standard: all types that are shared between threads and that may be used with the standard library must be either bitwise const, which is clearly not the case when we witness a write, or else protected against concurrent accesses [15]. The following example, from Protobuf, is synchronized using Google mutex primitives.

```
GOOGLE_SAFE_CONCURRENT_WRITES_BEGIN();
_cached_size_ = total_size;
GOOGLE_SAFE_CONCURRENT_WRITES_END();
```

- not visible (N): indicates that the result of the write is never externally visible (e.g. private and with no accessor methods; may be accessed in the same translation unit). Often occurs in the context of testing-related counters.

```
mutable int countFooCalls;
void foo() const { ++countFooCalls; }
```

- to a buffer/cache (B): indicates that the write is of a derived value which can be computed from other currently-available state. Such writes are often optimizations.

```
_cached_size_ = total_size;
```

- delayed initialization (D): indicates that the write initializes state not initialized in the constructor or its transitive callees. Writes with this attribute could have also occurred in the constructor, but the written value was not yet available. Failure to call a delayed initialization method would lead to undesired behaviours (or lack of desired behaviours).

```
bool Generator::Generate(const FileDescriptor* file, ...) const {
  this->file_ = file;
}
```

- incorrect (I): indicates that the write appeared to violate the `const`-ness of the object.

Note that S/N/B/D writes are not necessarily errors and do not necessarily violate immutability properties. We thus chose the word "attribute" to suggest that S/N/B/D indicate an incidental property of a write-through-`const`. If the code containing the writes is properly written, an object with an S/N/B/D write-through-`const` can still appear to be immutable to the client, assuming all references to that object are read-only. A write with attribute I, however, is a client-visible violation of `const`.

6 Results

We evaluated our ConstSanitizer tool on 7 C++ software projects, plus 1 C project. We attempted to choose significant benchmarks using these guidelines:

1. must span a range of application areas: applications and libraries; small, medium, and large projects; interactive and non-interactive;
2. are used by the community: the Google projects are the most popular on GitHub; the applications are popular among FOSS users; contributor-group sizes vary from a core group to a large community; and,
3. must extensively use `const` constructs.

A ConstSanitizer report indicates that a write that would not be allowed under deep immutability occurred through a read-only reference. Such writes are allowed under C++ semantics. They are only a departure from the `const` semantics that we experiment with (i.e. deep immutability with no casts and no mutable). Our experiments classify writes-through-`const` observed in actual `const`-using programs. Classifying these writes provides us valuable insight about `const` usage in practice, which will guide future work.

Our approach was to modify the project's build system to use our tool and to disable optimizations. We then ran the project's test suite, when available, and collected output from our instrumentation. Using this output we categorized the writes that we found, assigning root causes and attributes. Along with the number of static locations of writes that we found (**bolded**), we also report the number of dynamic occurrences of each write over observed executions. All else being equal, dynamic counts can help prioritize writes-through-`const`, with more-frequent locations to be investigated first. We refer to these dynamic occurrences of writes as "occurrences" in the sequel. Table 5 summarizes our benchmark projects.

Table 5 We ran our experiments across 7 C++ (and 1 C) software projects; ConstSanitizer introduces a build slowdown of 1.05×–1.40× across all projects.

	Name	**Version**	**Description**	**Build Slowdown**
C++	Protobuf	2.6.1	Serialization framework	1.40×
C++	LevelDB	1.18	Key/value database	1.05×
C++	fish shell	2.2.0	UNIX shell	1.32×
C++	Mosh (mobile shell)	1.2.5	SSH replacement	1.26×
C++	LLVM TableGen	3.7.0	Domain-specific generator	—
C++	Tesseract	3.04.00	OCR engine	1.10×
C++	Ninja	1.6.0	Build system	1.20×
C	Weston	1.9.0	Wayland compositor	1.28×

We recorded relative overhead introduced by our instrumentation with respect to both building and testing times on the longest-running projects, Protobuf and LevelDB. Table 5 includes build slowdowns induced by our tool, which ranged between 1.05× and 1.40×. Our tool caused a 3.3× slowdown and 1.3× slowdown in test execution times for Protobuf and LevelDB respectively. The remaining projects were either interactive, or did not have long enough running test suites to get meaningful results. We do not any report LLVM TableGen numbers because we built it (with instrumentation) as part of the LLVM build process and were not able to build the executable separately.

6.1 Protobuf

Protobuf is Google's serializing framework for structured data, consisting of about 214 000 lines of C++ code. We analyzed version 2.6.1 of Protobuf by running its test suite, which contains 5 tests. Table 6 summarizes the Protobuf results. ConstSanitizer found **76** static write locations (and 127 644 occurrences). We describe 5 archetypes for these writes. An archetype is a group of writes that we judged to be similar; the writes may happen at different source locations.

Table 6 Protobuf shows 5 archetypes for **76** writes through `const` resulting in 127 644 occurrences.

Archetype	Locations	Occurrences	Root Cause	Attributes
Generator printer	**7**	118464	T	B & N & S
Message cache sizes	**61**	7158	M	B & S
Source code locations	**4**	1898	T	I
Linked list operations	**2**	84	M	I & S
Generate initialization method	**2**	40	M	D & N & S

The "Generator printer" archetype occurred most often. Listing 5 presents a representative expanded stack trace. The function at the top of the listing shows the initiation of the write in `Generator`'s `const`-qualified `Generate` method. This method calls `PrintTopBoilerplate`, passing a pointer to a mutable `io::Printer`. Then, `Printer`'s `WriteRaw` method modifies (root cause T) two fields: `buffer_` and `buffer_size_`. These fields are protected by a lock, act as a buffer, and are not visible outside the class (which is just a printer). This archetype also includes other `Print`-like calls with different source locations but a common explanation.

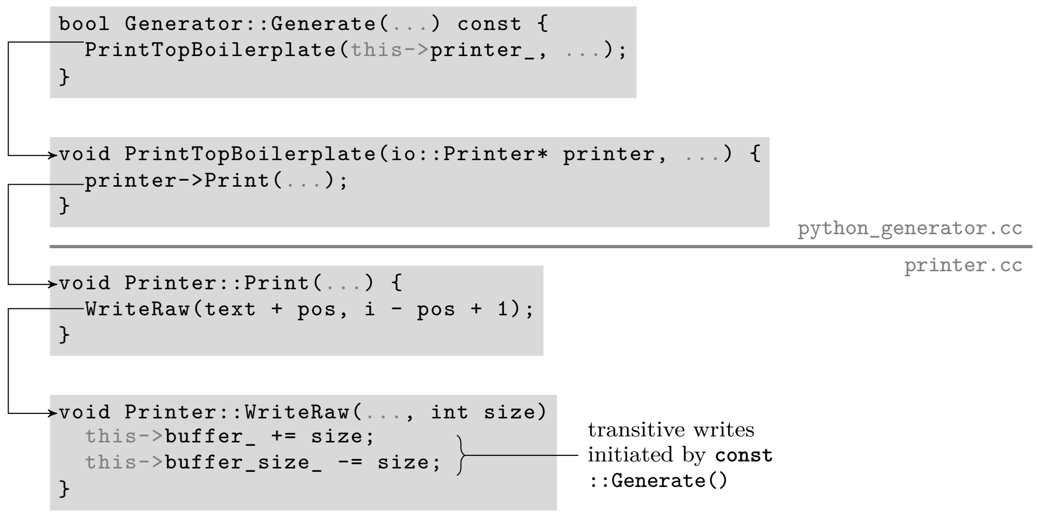
The "Generate initialization method" archetype is related to "Generator printer". Listing 6 shows this archetype. The `printer_` field was initialized as seen above. C++ allows this write due to the `mutable` specifier. Another field, `file_`, is lazily initialized as well. Both of these fields are protected by the same lock, and are not externally visible outside the class.

We show an example of the "linked list operations" archetype in Listing 7. Here, the `depart` method grabs a lock, and uses a pointer with type `linked_ptr_internal const *`, so that the `const` applies to what is pointed to, not to the pointer. The method then modifies the `next_` field of a valid object at the point indicated by the comment. The root cause here is `mutable`: the `next_` field is declared `mutable linked_ptr_internal const* next_`. This write is a delayed initialization, not visible, and synchronized.

Listing 8 shows the "Message cache sizes" archetype. The write is protected by a lock, and is allowed by C++ because the field is `mutable`. However, this write, while involved with caching, is externally visible. The method `void SetCachedSize(int size) const` enables external code to modify this field through a `const` reference to the containing object.

Listing 9 shows the "Source code locations" archetype. The `mutable_leading_comments` method, which includes "mutable" in its name, is not declared as `const`, and thus allows

Listing 5 Protobuf's Generator class performing transitive write-through-`const` to a Printer field.



```
bool Generator::Generate(...) const {
  PrintTopBoilerplate(this->printer_, ...);
}

void PrintTopBoilerplate(io::Printer* printer, ...) {
  printer->Print(...);
}
```

python_generator.cc

printer.cc

```
void Printer::Print(...) {
  WriteRaw(text + pos, i - pos + 1);
}

void Printer::WriteRaw(..., int size)
  this->buffer_ += size;
  this->buffer_size_ -= size;
}
```

Listing 6 Protobuf's Generate initialization method performing lazy initialization.

```
bool Generator::Generate(...) const {
  this->file_ = file;
  this->printer_ = &printer;
}
```

python_generator.cc

Listing 7 Protobuf using Google test linked list that writes internally.

```
bool linked_ptr_internal::depart()
    GTEST_LOCK_EXCLUDED_(g_linked_ptr_mutex) {
  MutexLock lock(&g_linked_ptr_mutex);

  if (this->next_ == this) return true;
  linked_ptr_internal const* p = this->next_;
  while (p->next_ != this) p = p->next_;
  p->next_ = this->next_;
  return false;
}
```

gtest-linked_ptr.h

Listing 8 Protobuf writing to a message's cached size field.

```
int FieldDescriptorProto::ByteSize() const {
  GOOGLE_SAFE_CONCURRENT_WRITES_BEGIN();
  this->_cached_size_ = total_size;
  GOOGLE_SAFE_CONCURRENT_WRITES_END();
}
```

descriptor.pb.cc

writes. Its implementation writes to the `location_` field; we show an example of a caller which causes such a write. The `location_` field is externally-visible, so this is a clearly incorrect externally-visible transitive write; we assign attribute I.

We also found an archetype involving writing data to a message. This included 133 unique source locations, occurring 14 638 times in total. However, the code is heavily inlined and the build system appears to overwrite optimization settings for this subdirectory. Manual inspection of the code revealed no obvious writes. We believe this is a result of optimizations causing invalid debugging information. We thus omitted this archetype from Table 6.

Listing 9 Protobuf writing to a source location object.

```
void Parser::LocationRecorder::AttachComments(...) const {
  this->location_->mutable_leading_comments()->swap(*leading);
}
```

parser.cc

descriptor.pb.h

```
std::string* SourceCodeInfo_Location::mutable_leading_comments() {
  this->leading_comments_ = new ::std::string;
}
```

6.2 LevelDB

LevelDB (1.18) is Google's lightweight key/value database library, consisting of approximately 18 000 lines of C++ code. The test suite contains 23 test drivers. There were 6 archetypes and also **6** root source locations for these writes. These locations contributed to 13 792 occurrences over the test drivers. Table 7 shows a summary of our findings for LevelDB.

Table 7 LevelDB shows writes from **6** source locations, with 13 792 occurrences in total.

Location	Occurrences	Root Cause	Attributes
db/db_test.cc:40	10311	T	N & S
util/cache.cc:315	2841	T	B & S
db/snapshot.h:54	319	T	I
db/snapshot.h:55	319	T	I
helpers/memenv/memenv.cc:274	1	T	I & S
util/testutil.h:42	1	T	N

Listing 10 shows the source location that caused the majority of the occurrences. This code extends the `RandomAccessFile` class to add an atomic counter field, `counter_`, that tracks the number of read calls. The root cause is that `counter_` is a pointer and is transitively written to. The reason for this write is test controllability: this class is part of the test infrastructure. Yet it must override the monitored call (and thus must be **`const`**). This class is meant for testing purposes only, so we concluded the write was not visible outside the class—the counter is only used in the testing code.

Listing 10 LevelDB write in `db_test.cc:40` incrementing counter tracking # of writes to a file.

```
class CountingFile : public RandomAccessFile {
  virtual Status Read(...) const {
    this->counter_->Increment();
  }
};
```

db_test.cc

Listing 11 shows a modification to a caching structure that generates a new identifier. This cache is a field, `block_cache`, in `options`, which is declared as `const Options&` in `Table::Open`. The root cause is a transitive write, since the code dereferences a field of a **`const`** object to do the write. This write is protected by a lock and clearly involved in caching. However, it appears that other code outside of `Options` uses this block cache.

Listing 12 shows a modification of a linked list node accessed through two pointer dereferences. This corresponds to both `snapshot.h` locations shown in the table. This code

Listing 11 LevelDB write in `cache.cc:315` creating a new block cache in `const Options` object.

table.cc
```
Status Table::Open(const Options& options, ...) {
    rep->cache_id = (options.block_cache ?
                     options.block_cache->NewId() : 0);
}
```

cache.cc
```
virtual ShardedLRUCache::uint64_t NewId() {
  MutexLock l(&id_mutex_);
  return ++(last_id_);
}
```

modifies the pointers obtained from following its own nodes, performing a transitive write through a `const` qualifier. We do not know why the developers declared `s` as `const` since it is also destroyed at the end of the method. In any case, we assigned attribute I.

Listing 12 LevelDB write in `snapshot.h` deleting a list element and updates pointers.

snapshot.h
```
void Delete(const SnapshotImpl* s) {
  assert(s->list_ == this);
  s->prev_->next_ = s->next_;
  s->next_->prev_ = s->prev_;
  delete s;
}
```

Listing 13 shows a write-through-**`const`** in the `InMemoryEnv` class. As with the cache, the root cause is a transitive write: in the caller, `options` is declared `const Options&`. Unlike the caching example, this file isn't involved in caching and appears to be a visible change to `options`. This write is protected by a lock, giving attribute I & S.

Listing 13 LevelDB write in `memenv.cc` changing the environment in options object.

db_impl.cc
```
Status DB::Open(const Options& options, ...) {
  s = options.env->NewWritableFile(...);
}
```

memenv.cc
```
... InMemoryEnv::NewWritableFile(...) {
  MutexLock lock(&mutex_);
  this->file_map_[fname] = file;
}
```

Listing 14 shows the final write-through-**`const`**-qualifier that we found for LevelDB. The caller location is the same as in Listing 13 above. In this case, however, the containing class extends `InMemoryEnv` and adds a field to count the number of errors (for testing purposes only). Therefore we attribute this write as being not visible—it is only used in tests.

Listing 14 LevelDB write in `testutil.h` injecting faults into the test suite.

testutil.h
```
... EnvError::NewWritableFile(...) {
  ++this->num_writable_file_errors_;
}
```

6.3 fish shell

fish shell (2.2.0) is a UNIX shell providing advanced features, consisting of approximately 48 000 lines of C++ code. We compiled the project with our tool and executed an instance of the shell. Our workload launched the shell and immediately exited. We found writes from **4** unique source locations for 98 occurrences in total. All locations are within the `exchange` function. Listing 15 shows this function along with a snippet of `_wgetopt_internal` that calls `exchange`. The root cause is that the **`const`**-qualified `argv` variable gets cast to non-**`const`** and then passed to `exchange`. This write shows that the **`const`**-qualifier on `argv` is incorrect and should not be included.

Listing 15 fish shell writing to **`const`**-qualified `argv` object.

wgetopt.cpp

```
... _wgetopt_internal(..., wchar_t *const *argv, ...) {
  exchange((wchar_t **) argv);
}
```

```
... exchange(wchar_t **argv) {
    argv[bottom + i] = argv[top - (middle - bottom) + i];
    argv[top - (middle - bottom) + i] = tem;
    argv[bottom + i] = argv[middle + i];
    argv[middle + i] = tem;
}
```

6.4 Mosh (mobile shell)

Mosh (mobile shell) (1.2.5) is a remote terminal application that is a replacement for secure shell (SSH), consisting of about 13 000 lines of C++ code. Our workload was to launch the mosh server and immediately terminate it. We found writes-through-**`const`** at **8** unique source locations (432 occurrences). Listing 16 shows one of the writes. Mosh parsing code sets a flag to indicate completion. However, the developers declared the parser action as **`const`** in the same method where they modify it. The root cause is that the variable `handled` is declared `public mutable`. We believe this is an incorrectly **`const`** qualified variable.

Listing 16 Mosh handling terminal action with a write-through-**`const`**.

terminal.cc

```
void Emulator::print(const Parser::Print *act) {
  act->handled = true;
}
```

6.5 LLVM TableGen

We instrumented LLVM's (3.7) TableGen executable, which uses domain-specific information to generate files with custom backends. This part of LLVM consists of approximately 34 400 lines of C++ code. It is primarily used in building LLVM itself. We added our instrumentation to the build system and observed reports from an instrumented version of TableGen executing as part of the build process. LLVM itself is a large body of code with too many writes-through-**`const`**-qualifier objects to manually classify. In TableGen, we found writes from **3** unique source locations (282 occurrences).

The handling code for DFAs contains some puzzling writes, shown in Listing 17. The write immediately follows an instantiation of a **`const`** `State` object. The `State` class itself is only available in a file's translation unit (not usable outside the file), which may indicate that the `State` is not intended to be widely used. `State` only contains **`const`** methods and all of its fields (except one explicitly declared **`const`**) are mutable. Since all methods are **`const`** there is no difference in callable methods between non-**`const`** and **`const`**-qualified access. In addition, since all other fields are mutable, developers are allowed to re-assign the same fields in a **`const`** method as they would in a non-**`const`** method. Since only one field doesn't have **`mutable`**, developers could achieve the same effect by making all methods non-**`const`**, removing all **`mutable`** specifiers on fields, and changing the one field that did not have **`mutable`** to be **`const`** qualified.

Listing 17 LLVM DFA code marks `State` **`const`** for no apparent reason.

```
void DFAPacketizerEmitter::run(raw_ostream &OS) {
  const State *NewState;
  NewState = &D.newState();
  NewState->stateInfo = NewStateResources;
}
```

DFAPacketizerEmitter.cpp

The other write is in the code that computes a sub-register index for code generation. Listing 18 shows the containing method. The root cause here is that the `LaneMask` field is mutable. The write caches the value. However, this value is not used in any other methods.

Listing 18 LLVM SubReg writes to a **`mutable`** field in a **`const`** method.

```
unsigned CodeGenSubRegIndex::computeLaneMask() const {
  if (this->LaneMask)
    return this->LaneMask;
  this->LaneMask = ~0u;
  unsigned M = ...;
  this->LaneMask = M;
  return this->LaneMask;
}
```

CodeGenRegisters.cpp

6.6 Tesseract

Tesseract (3.04.00) is an optical character recognition (OCR) engine maintained by Google, consisting of 147 000 lines of C++ code. This project does not contain any easy-to-run tests. We compiled it with our tool and ran it with invalid arguments. With our limited knowledge of Tesseract's usage, we were not able to cause the core algorithm to execute. However, we found a strange write, shown in Listing 19. The root cause is that the `used_` field is mutable. This write appears to be an incorrect usage of **`const`**. Strangely, however, the comments indicate that is a defensive write against possible further writes-through-**`const`**-qualifiers.

Listing 19 Tesseract performs a strange write in its string class.

```
const char* STRING::string() const {
  const STRING_HEADER* header = GetHeader();
  header->used_ = -1;
  return GetCStr();
}
```

strngs.cpp

```
/* mark header length unreliable
because tesseract might cast away
the const and mutate the string
directly. */
```

6.7 Ninja

Ninja (1.6.0) is a build system consisting of approximately 14 900 lines of C++ code. It includes a modest test suite. Our tool reports 39 occurrences from calls to the standard library. All of these warnings have a **single** source location outside the standard library: `src/disk_interface_test.cc:226:3`. Listing 20 shows this static source location. This is a quick hack to run the test suite with the same API as normal clients. This field stores statistics that are checked in the test suite only. The field is mutable and not seen outside the test suite, so we give this write the "not visible" attribute.

Listing 20 Ninja write-through-**`const`** in test code.

```
TimeStamp StatTest::Stat(const string& path, ...) const {
  this->stats_.push_back(path);
}
```
disk_interface_test.cc

6.8 Weston

While we focus on C++ in this work, our technique also works on **`const`** in C programs. We therefore evaluated it on a C application. Since this is the sole C project, we omit Weston from the overall table of results (Table 8). Weston (1.9.0) is a reference implementation of a Wayland compositor. It consists of approximately 85 000 lines of C code and the test suite has 20 tests. We did not expect to see many writes, as most C standard library functions do not require **`const`** (corresponding C++ library functions usually do), and also due to annoyances in using **`const`** in C, which we describe below. However, even with a small test suite, we found **4** unique source locations for writes (accounting for 115 occurrences).

All of the writes-through-**`const`** are transitive and came from parsing code. The argument option parser accounts for **3** locations. Listing 21 shows a write in the parser. Function `handle_option` does not modify the pointer value of `option` but modifies its transitive data field. This does not change any data stored in the `weston_option` structure, maintaining bitwise **`const`**-ness. The `data` field's type is `void *` and the cast does not remove **`const`**. Based on the function name, one might expect a write to the `data` field, not its pointee.

Listing 21 Weston option parser modifying its **`const`** option argument.

```
handle_option(const struct weston_option *option, ...) {
  * (char **) option->data = strdup(value);
}
```
option-parser.c

The final location was in the configuration file parsing code. Listing 22 shows the `weston_config_section_get_uint` function dereferencing and modifying the `value` argument passed in from a field of a **`const`** struct. As above, based on the function naming, one would expect any writes to happen through the `dest` pointer. This write does not modify any data stored in the `config_command` structure and maintains bitwise **`const`**-ness as well.

We made an observation as to why **`const`** may be unattractive to C developers: there is no clean way to initialize a structure analogous to C++ constructors/destructors. A popular C idiom is to assign constructor-like functions signatures like `rec_init(struct rec *r)`. This signature prevents initialization without casting: `const struct rec r; rec_init(&r)` is illegal. However, it is cumbersome to always cast for constructor-like calls. One could change the signature of the function to `rec_init(const struct rec *r)` and perform the

Listing 22 Weston config parser writing to its `value` argument.

hmi-controller.c

```
struct config_command {
  char *key;
  uint32_t *dest;
};
const struct config_command *command = ...;
```

```
weston_config_section_get_uint(..., command->dest, ...);
```

```
... weston_config_section_get_uint(..., uint32_t *value, ...) {
    *value = strtoul(entry->value, &end, 0);
}
```

cast in the function. However, that function would violate shallow immutability—it writes to fields as it initializes them. Using **`const`** in C appears to require developers to ignore the casting away of **`const`** qualifiers for constructor-like functions.

6.9 Summary

Table 8 summarizes the writes-through-**`const`**-qualifiers from benchmarks other than Protobuf and LevelDB. Across the 7 C++ projects we instrumented and ran, we observed 17 unique archetypes across a total of 142 288 dynamic occurrences. We manually divided these archetypes into 17 classifications. The root causes were evenly split between writes through mutable fields and transitive writes (8 of each) with one write-through-**`const`** due to casting. Valid attributes were mostly with-synchronization and because the write was not visible (7 and 6 respectively). The other valid attributes, writing to a buffer/cache and delayed initialization, occurred 4 times and 1 time respectively. The majority attribute, in 9 cases, was that the write was incorrect and violated intuitive notions of what **`const`** should mean. We reported our results to developers. Within a few days the developers simply removed incorrect **`const`** qualifiers in both fish and Mosh.

Table 8 Writes-through-**`const`**-qualifiers in our other benchmark programs were mainly incorrect uses of `const`.

Project	Location	Occurrences	Root Cause	Attributes
fish shell	`wgetopt.cpp`	98	C	I
Mosh	`terminal.cc`	432	M	I
LLVM	`DFAPacketizerEmiter.cpp`	112	M	I
TableGen	`CodeGenRegisters.cpp`	170	M	B & N
Tesseract	`ccutil/strngs.cpp`	1	M	I
Ninja	`disk_interface_test.cpp`	39	M	N

We found 3 projects (LevelDB, Mosh, and Ninja) had writes that only occurred in tests. The writes-through-**`const`** we found were in testing code; writes were to counters only present in test environments. In Mosh, the fact that the writes were only for test purposes was not immediately obvious. However, discussions with the developers revealed that the **`handled`** variable was only used for debugging. All of these writes-through-**`const`** are related to test controllability, suggesting that this idiom should be supported directly in the programming language.

7 Related Work

We discuss a number of areas of related work: immutability (and related approaches) for Java; purity analyses; and dynamic analyses that inspired our approach. The Java programming language has no exact analog to C++'s `const` operator. Related work defined immutability annotations for Java and statically and dynamically verified that programs satisfy their annotations. Potanin et al [9] provide a recent discussion of immutability terminology and compare research implementations in depth. (Our implementation has at its core a dynamic analysis verifying that C++ programs satisfy a strengthened version of their `const` annotations.) A different stream of related work verifies whether (Java) methods are pure, i.e. have no visible side-effects; there is a strong connection between purity and immutability.

`const` for Java, and similar projects. Javari [17, 18] allows its users to specify read-only references in Java programs. It aims to ensure that a `readonly` (effectively, deeply-immutable `const`) typed object does not mutate its state or any state transitively reachable through its references. Javari, like C++, includes a `mutable` keyword, which allows developers to specify that a field may be modified regardless of `readonly` qualifiers. Javari also inserts dynamic checks to verify that downcasts maintain the immutability qualifier of the type. Essentially, Javari provides a safer version of C++'s `const` that, due to the nature of Java, maintains deep immutability unless the developer explicitly opts out of the checks.

Our work provides similar guarantees to those provided by Javari. Since Javari does not have an underlying C++ `const` specification to build on top of, it has to implement all of those checks itself. In terms of what we check, our work matches Javari in terms of transitivity and downcasts. Our work also reports writes to mutable fields at runtime.

Unlike Javari, we could investigate the behaviour of a set of real-world benchmark programs, developed against the C++ `const` semantics (and which, by necessity, satisfy those semantics). Our empirical study therefore points out the difference in practice between `const` semantics as they exist—shallow immutability, mutable, and `const` casting; and a stronger version of these semantics—deep immutability, no mutable, and no const casting.

A related concept to `const` is that of stationary fields, as proposed by Unkel and Lam [19]. A stationary field is similar to a Java `final` field. A Java `final` field can only be written to in a constructor. By contrast, a stationary field is only written to before it is read. In essence, a stationary field acts like a `final` field but with fewer restrictions on where initialization may occur. Nelson et al [8] performed a follow-up study using a dynamic analysis which determined that 72–82% of fields in Java programs are stationary. Their work, like ours, empirically explores how programs use (or could use, in their case) language features.

Purity analyses. A pure method does not modify any state accessible before the method was called. Pure methods may create and modify objects to return to the caller. A function which writes to no global state and has all arguments transitively `const` is pure.

ReIm and its corresponding type inference system, ReImInfer [5], is similar to Javari, except that its type system is context sensitive. ReIm was designed to find method purity. Its type system allows the immutability of the return type to match that of the calling reference. This allows methods to be reused without requiring mutable and read-only versions. ReImInfer is a type inference system that maximizes the amount of methods marked as `readonly`. They report that 41–69% of methods can be marked as `readonly`.

Other systems also verify method purity. JPPA is a combined pointer and escape analysis for Java [12, 11]. Sălcianu and Rinard found over half the methods they analyzed were pure.

Rytz et al [10] instead use a simpler flow-insensitive analysis and find similar results.

Artzi et al proposed a combined dynamic and static analysis for mutability [1]. Their analysis determines the mutability of method parameters. The goal of that work was to scale better and produce more precise results than static analysis alone.

`const`-correctness through abstract machines. Chisnall et al [3] propose a memory-safe abstract machine for C which can be used to verify that immutable objects (declared with `const`) are never mutated through non-`const` aliases. This resembles our approach of using shadow values to track the declared `const`-ness of an object. (They do not investigate transitive immutability.) Our study, by contrast, focuses solely on writes-through-`const`. All of their subject programs removed a `const` qualifier at some point. It was beyond the scope of their work to investigate how and why their subject programs removed the `const` qualifier.

Other dynamic analyses. Our dynamic analysis approach is similar to approaches used in Umbra [20] and Dr. Memory [2]. Like Umbra and Dr. Memory, we use shadow values to detect interesting program behaviours. However, ConstSanitizer builds directly on LLVM and does not use a dynamic instrumentation platform. Furthermore, the properties that we verify are novel `const`-related properties, compared to Dr. Memory, which looks for memory errors such as accesses to unallocated space, and Umbra, which helps developers understand threads' memory access patterns and implements almost-free custom watchpoints.

8 Conclusion

The `const` qualifier in C++ is extensively used in real-world code, but developer intent behind `const` usage is unclear. In this paper, we have presented our ConstSanitizer system. ConstSanitizer dynamically detects writes through `const` qualifiers which are legal in C++ but which modify state transitively starting from a `const` qualifier, write to mutable fields, or write to values whose `const`-ness has been cast away. Our results show that, although writes through `const` are ubiquitous across our 7 C++ and 1 C benchmark programs, there are only a small number (17) of archetypes these writes. We used our results to develop a classification of writes according to root cause (transitivity, mutable field, or const cast) and attributes (synchronized, not visible, buffer/cache, delayed initialization, incorrect). Our work helps understand how the `const` qualifier is used in practice and leads us to conclude:

- Developers definitely violate bitwise `const` (**RQ1**).
- The majority of write-through-`const` archetypes (9/17) are incorrect code which observably change an object's state.
- On our benchmarks, programs write-through-`const` about equally often using transitive writes through fields (8/17) and writes to mutable fields (8/17) (**RQ2**).
- We observed four classes (N, B, D, S, discussed in Section 5) of valid reasons for writes-through-`const`-qualifiers. For instance, sometimes developers write-through-`const` to delay initialization or implement buffer caches. Such writes should be automatically validated by yet-to-be-developed tools.
- About half (9/17) of the observed usages are invalid, consisting of methods which implemented exceptions to an object's const-ness; perhaps the developers chose to add one exception rather than remove const completely. (**RQ3**)

Acknowledgements. This work was supported in part by Canada's Natural Science and Engineering Research Council as well as a Google Faculty Research Award.

References

1 Shay Artzi, Adam Kiezun, David Glasser, and Michael D. Ernst. Combined static and dynamic mutability analysis. In *ASE*, pages 104–113, November 2007.

2 Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *CC*, pages 213–223, 2011.

3 David Chisnall, Colin Rothwell, Robert N. M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the PDP-11: architectural support for a memory-safe C abstract machine. In *ASPLOS*, 2015.

4 Felix Fang. Personal communication, 2015.

5 Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *OOPSLA*, 2012.

6 ISO. Programming languages—C++. N3690, May 2013.

7 Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison Wesley, 3rd edition, 2005.

8 Stephen Nelson, David J. Pearce, and James Noble. Profiling field initialisation in Java. In *RV*, volume 7687 of *LNCS*, pages 292–307, 2012. `doi:10.1007/978-3-642-35632-2_28`.

9 Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. Immutability. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *LNCS*, pages 233–269. 2013.

10 Lukas Rytz, Nada Amin, and Martin Odersky. A flow-insensitive, modular effect system for purity. In *FTFJP*, July 2013.

11 Alexandru Salcianu. *Pointer Analysis for Java Programs: Novel Techniques and Applications*. PhD thesis, MIT, 2006.

12 Alexandru Salcianu and Martin C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, pages 199–215, January 2005.

13 Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

14 Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *CGO*, pages 46–55, 2015.

15 Herb Sutter. GotW #6a solution: Const-correctness, part 1. `http://herbsutter.com/2013/05/24/gotw-6a-const-correctness-part-1-3/`, May 2013. Accessed Dec 2015.

16 LLVM Team. The LLVM compiler infrastructure. `http://llvm.org/`, December 2015.

17 Matthew S. Tschantz. Javari: Adding reference immutability to Java. Master's thesis, Massachusetts Institute of Technology, 2006.

18 Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, October 2005.

19 Christopher Unkel and Monica S. Lam. Automatic inference of stationary fields: a generalization of Java's final fields. In *POPL*, pages 183–195, January 2008.

20 Qin Zhao, Derek Bruening, and Saman P. Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *CC*, pages 22–31, 2010.