# ARC++: Effective Typestate and Lifetime Dependency Analysis

Xusheng Xiao[1,2]
xxiao2@ncsu.edu

Gogul Balakrishnan[1]
bgogul@gmail.com

Franjo Ivančić[5]
ivancic@google.com

Naoto Maeda[3]
n-maeda@bp.jp.nec.com

Aarti Gupta[1]
agupta@nec-labs.com

Deepak Chhetri[4]
deepak.chhetri@nectechnologies.in

[1]NEC Labs America, USA [2]North Carolina State University, USA [3]NEC Corporation, Japan
[4]NEC Technologies India Ltd., India [5]Google, Inc., USA

## ABSTRACT

The ever-increasing reliance of today's society on software requires scalable and precise techniques for checking the correctness, reliability, and robustness of software. Object-oriented languages have been used extensively to build large-scale systems, including Java and C++. While many scalable static analysis approaches for C and Java have been proposed, there has been comparatively little work on the static analysis of C++ programs. In this paper, we provide an abstract representation to model C++ objects, containers, references, raw pointers, and smart pointers. Further, we present a new analysis called lifetime dependency analysis, which allows us to precisely track the complex lifetime semantics of temporary objects in C++. Finally, we propose an implementation of our techniques and present promising results on a large variety of open-source software.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

## General Terms

Verification, Reliability

## Keywords

Bug finding, C++, abstract representation, typestate analysis, static analysis

## 1. INTRODUCTION

The ever-increasing reliance of today's society on software requires scalable and precise techniques for checking the correctness, reliability and robustness of software. Recently, automatic approaches, such as automated test generation [12, 20, 31] and static software verification [7, 22, 19, 23], have been used to detect bugs in software and improve software quality. This paper targets the static analysis of object-oriented programs written in C++ using light-weight abstract interpretation techniques by looking for common programming mistakes that are specific to C++.

In the past decade, there has been strong interest in developing practical techniques to find *bug patterns* [21, 32, 34]. *Bug patterns* are code idioms that generally lead to various types of software issues, such as run-time failures, performance issues, etc. Such code idioms are targeted because they address an often observed misunderstanding of the language semantics or run-time behavior [21]. Recently, it was also shown that bug patterns can be used to detect non-functional bugs, such as performance bugs [24].

Additionally, in systems built using object-oriented languages, such as C++ or Java, *object protocols* or *typestates* [10, 16, 39] have been found to be a useful abstraction for developing and maintaining large software projects. They have also been effectively used to capture errors arising from misuse of various operations on objects. Typestates refer to an abstraction of a concrete object state, and they may change due to the operations performed on the object [39]. For example, a file object $f$ may only be read after being properly opened. Typestates are usually modeled as finite state automata, and can be used to detect bugs by searching for violations of usage constraints [8]. In this paper, we represent bug patterns as typestate automata and seek to develop an effective analysis that can support typestate checking in C++ programs.

### *Motivation: Static Analysis of C++.*

Software development teams have shifted their development from C to object-oriented languages, including C++ and Java. The benefits of using an object-oriented language include reusability, better maintainability, encapsulation, and the use of inheritance. In particular, C++ is often chosen due to its ability to interact with legacy C-based systems, including system-level C libraries. Thus, development in C++ often necessitates a mixed programming style combining object-oriented constructs with lower-level C code. Whereas a large volume of work on verification has focused on C or Java, there has been comparatively little work on the verification of C++ programs.

Handling C++ programs using static analysis is non-trivial because of the complexities of the C++ semantics. One particular issue in handling C++ compared to other object-oriented programming languages such as Java is the com-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*ISSTA* '14, July 21-25, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2645-2/14/07 ...$15.00.

```
1 void test(auto_ptr<int> ptr);
2 int main() {
3   auto_ptr<int> p(new int);                          // initializes p
4   test(p);       // transfer ownership from p to formal parameter
5   return *p ;                                   // segmentation fault
6 }
```

**Figure 1: Example C++ program using auto_ptr.**

plexity of allowed class hierarchies. C++ allows multiple inheritance, where a class may inherit from more than one class. Furthermore, multiple inheritance complicates the semantics of otherwise simple operations such as casts and field accesses. Therefore, techniques developed for Java are not readily applicable to C++ programs. It is important to emphasize that multiple inheritance is used quite frequently by developers even in large C++ projects. We developed the Chrome object representation to address some of this challenge for software model checking in our earlier work [41].

**Exceptions.** Another difference between Java and C++ semantics is the handling of exceptions. The exception specification and checking mechanism in Java is more strongly enforced than in C++. Indeed, in C++, exception specifications other than an indication that a function does not throw any exception, are discouraged due to negative performance impact and the possibility that they are inaccurate (in which case an overloadable function `std::unexpected()` is called). A major difference from Java is that, in C++, when an exception is passed to another scope or leaves a functional scope, all stack allocated objects along the way to the appropriate exception handler need to be deallocated using a process called *stack unwinding*. During stack unwinding, object destructors are called, which may generate further exceptions, leading to a scenario with multiple live exceptions. We address the issue of *exceptions* by building an inter-procedural exception control flow graph (IECFG) [33].

**Standard libraries.** Another challenge is the use of standard libraries, such as STL and BOOST. These libraries introduce containers (such as `vector` and `stack`) and new types of pointers (such as iterators and smart pointers), and heavily use templates or operator overloading to define new operations around these new `container` and pointers. For example, `it = vector.begin()` initializes the iterator `it` (a new kind of pointer) to point to the first item of a vector.

Consider the C++ program shown in Fig. 1. The program has two functions, `test` and `main`. The function `test` accepts an *auto pointer* `ptr`, and performs certain operations using the memory managed by `ptr` (not shown here). An auto pointer is a wrapper class of a pointer to an object, that ensures that the pointed to object is destroyed when control leaves the auto pointer scope. The function `main` allocates memory for an auto pointer `p` and makes a call to `test` (line 4). After the call to `test` finishes, the program modifies the value in the memory managed by `p` (line 5).

Although this simple program seems semantically correct, its execution results in a segmentation fault at line 5. The reason is that when an assignment operation or a copy construction takes place between two `auto_ptr` objects, ownership of the memory is transferred, which means that the `auto_ptr` object losing ownership is reset to `null`. Since the parameter `ptr` of `test` is declared as pass-by-value, a copy constructor is invoked to initialize `ptr` using `p` at line 4. Thus, after line 4, `p` becomes `null`, and an error occurs when the program tries to dereference `p` at line 5.

```
1  void test(auto_ptr<int> ptr);
2  int main() {                                         // abstracted view
3    int *tmp1;                                 // Declarations only, i.e.
4    auto_ptr<int> p, tmp2;                    // not C++ style semantics
5    tmp1 = new int ;                                   // start of Line 3
6    Use(tmp1);                                // argument in constructor
7    Create(p,tmp1);                                // constructor for p
8    Use(Addr(p));          // start of Line 4: calls copy−constructor
9    Create(tmp2,Addr(p));       // copy−constructor for argument
10   Destroy(p);         // ownership transferred to formal parameter
11   Call(test,tmp2);                                     // function call
12   Use(p);                           // Line 5: Use−after−destroy (error)
13 }
```

**Figure 2: ARC++ representation for Fig. 1.**

To detect this bug, a static analysis requires a model of the implicit copy constructor (line 4), the ownership transfer of the copy constructor and its side-effect on `p`. To model the high-level semantics of such libraries, we propose an abstract representation for C++ statements called Arc++ (Abstract Representation for C++). Fig. 2 shows the Arc++ representations for the program shown in Fig. 1.

**Temporary objects.** One of the peculiarities of the C++ semantics is related to the notion of lifetimes of temporary objects. In C++, temporary objects are often created by the compiler, causing performance and correctness issues that are often hard to find and understand. Temporary objects are unnamed objects created on the stack by the compiler. They are used during reference initialization and during evaluation of expressions including standard type conversions, argument passing, function returns, and evaluation of the throw expression. Performance bottlenecks can arise due to the unnecessary creation and destruction of such temporary objects. Correctness issues can arise due to the complex lifetime semantics of temporary objects often leading to accesses of previously freed/destructed memory.

Generally, the correctness issues related to object lifetimes are hidden during testing due to the fact that stale uses of object storage often occur shortly after destruction of the object. Nevertheless, in an actual deployed production environment, such short-term stale uses can cause hard to find runtime errors and memory corruption, leading to memory faults. Furthermore, such memory corruption can also be potentially exploited by malicious users.

**Low-level memory management.** Furthermore, C++ needs to deal with legacy C semantics, APIs and programs, with low-level memory management issues that are not of concern for Java. One such issue is interaction with the semantics of temporary objects. In a prior effort, we have used model checking for finding interaction bugs involving STL strings and C-style strings [6]. A very short example is shown in Fig. 3: The call to `s.substr(.)` returns a temporary string object. The call to `c_str()` on that temporary object returns an internal pointer into the temporary string object, which represents a properly terminated C string. However, the temporary string object is destroyed after its lifetime, which is right after the call to `c_str()` in this case. Thus, the call to `strlen` in the following line is performed on a C string that has just been deallocated. To deal with such issues, this paper introduces a notion of *lifetime dependency* that captures the dependency of the validity of one object on the validity of another object, and we use a light-weight static analysis instead of model checking, which is more expensive.

```
1 int cutLen(const string &s,size_t i,size_t n) {
2   const char *str = s.substr(i,n).c_str();
3   return strlen(str);
4 }
```

**Figure 3: Stale buffer use from a temporary string.**

### *Our approach.*

This paper presents an abstract representation called Arc++ for modeling C++ objects as well as containers and smart pointers introduced by standard libraries. The main goal of Arc++ is to make object creation, destruction, usage, lifetime, and pointer operations explicit in the abstract model. The Arc++ representation provides the basis for a novel static analysis that we call lifetime dependency analysis to support typestate checking for C++ objects. Arc++ is also used for specifying common bug patterns in the form of typestate automata on these objects. We have developed an automatic framework to generate Arc++ representations from C++ programs and use our novel static analysis to perform typestate checking for many bug patterns.

**Abstract representation.** Each statement in C++ is mapped to a list of abstract instructions in Arc++. For example, Fig. 2 shows the Arc++ representation for the program in Fig. 1. Arc++ captures the ownership transfer of `auto_ptr`'s copy constructor and generates a `Destroy` statement for `p`, enabling static program analysis to detect the bug. Note that besides object creation and destruction, Arc++ also handles C++ references and pointer operations of containers and smart pointers, which have different semantics with respect to pointer aliasing (such as `it = vector.begin()`). Moreover, we utilize the results of the *exceptional flow analysis* [33], enabling developers to compose bug patterns related to exceptions.

**Lifetime dependency analysis.** The lifetime semantics of objects in C++ is quite complicated and can easily lead to correctness or performance issues. One such issue is due to the interaction of low-level memory management with the lifetimes of temporary objects. We use Arc++ to highlight the lifetimes of various objects. However, as we will show for the above example, many times the analysis of such C++ programs can be analyzed precisely by considering a new static analysis that we call *lifetime dependency analysis*.

To handle the lifetime semantics of C++, we consider lifetime dependencies between objects such that the destruction of one of the objects immediately implies the destruction of other *dependent objects*. For example, for the program shown in Fig. 3 we would introduce a lifetime edge for the unnamed temporary object returned from the `substr(.)` method call and the C string pointer `str`. At the time of the temporary object destruction, which occurs after the call to `c_str()`, we follow lifetime dependency edges to other implicitly destructed objects or access paths.

### *Contributions.*

- We present an abstract representation (Arc++ §2) that models C++ objects as well as containers, and smart pointers introduced by standard libraries. Arc++ makes object creation, destruction, usage, lifetime, and pointer operations explicit in the abstract model, providing a basis for static analysis on C++ programs.
- We propose a notion of "lifetime dependency" to track implied destructions between objects (§5). We show that this provides an effective high-level abstraction for many types of issues involving temporary objects and internal buffers. This is used in a novel flow-sensitive, context-sensitive, and path-insensitive static analysis that supports typestate checking for C++ objects (§3 and §4).
- We propose a framework (§6) that (i) automatically generates Arc++ representations from C++ programs, and (ii) performs typestate checking to detect bugs that are specified as typestate automata over Arc++ representations.
- We present five bug patterns using Arc++ and experimental results for checking them on 20 open-source C++ programs of more than 2M LOC. The results show that our framework is very fast and detected 23 bugs and 124 bad programming practices (§7).

## 2. Arc++ ABSTRACTION

This section describes the details of our abstract representation called Arc++. Each statement of a C++ program is mapped to a list of abstract instructions in Arc++ *automatically*. These abstract instructions can be directly used for bug pattern specification and checking. Fig. 4 presents a subset of Arc++.

**Expressions.** Arc++ expressions represent program variables, field accesses, cast operations, return values of function calls, and functional computations without side-effects or control flow. The parameters `e` and `type` refer to the expressions and types in a C++ program. A `Variable` expression represents program variables of primitive or user-defined types. A `Pointer` expression represents C pointers, C++ references, iterators of containers, and the various pointer classes (e.g., `shared_ptr`) in the standard library. Note that an iterator is conceptually treated as a pointer because an iterator may point to some element of the associated container. A `Container` expression is associated with an lvalue or a rvalue expression that represent containers. A container is either a C++ array or one of `Vector`, `List`, `Stack`, `Map`, and `Set` container types in the standard C++ library. The `type` argument in a `Container` expression represents the type of elements stored in the container. A `Call` expression represents the return value of either a global function call or a member function call. Besides the type of the return value, a `Call` expression also provides the information about the receiver object and an argument list.

**Abstract instructions.** Arc++ abstract instructions represent operations on expressions. These include object operations (`Create`, `Destroy` or `Use`), lifetime operations, and others. A `Create` instruction represents a call to the constructor of a class or a memory allocation for a pointer (such as *`new`*/*`malloc`*). A `Destroy` instruction represents a call to the destructor of a class, a memory deallocation (such as *`delete`*/*`free`*), or the reset on pointers (e.g., set pointer to `null`). Note that the implicit calls to copy constructor, destructors, and pointer resets (such as `auto_ptr` in Fig. 1) are made explicit and the corresponding abstract instructions are generated.

**Use operations.** A `Use` instruction represents how the value of an object is accessed. A `Value` operation represents that the underlying expression is a simple variable, a `Dereference` operation signifies that the underlying expression is a pointer dereference, a `Field` operation represents an access to a field of an object, and an `Index` operation represents that the underlying expression is an index into a container object, such as an array or a vector.

**Expressions:**

| | | |
|---|---|---|
| SPtrType | ::= | *AutoPtr* \| *UniquePtr* \| *SharedPtr* |
| PtrType | ::= | *CPtr* \| *Reference* \| *Iterator* \| *SPtrType* |
| ContainerType | ::= | *Array* \| *Vector* \| *List* \| *Stack* \| *Map* \| *Set* |
| Expr | ::= | *Variable*(e, type) |
| | | \| *Pointer* (PtrType, Expr, type) |
| | | \| *Container*(e, ContainerType, type) |
| | | \| *Call* (e, type, Expr option, Expr list) |

**Abstract instructions:**

| | | |
|---|---|---|
| AbsInstr | ::= | *Create*(id, Expr, Expr list) |
| | | \| *Destroy*(id, Expr) |
| | | \| *Use* (id, UseOp) |
| | | \| *Life*(id,LifeOp) |
| | | \| *Other* (id, i) |

**Use operations:**

| | | |
|---|---|---|
| UseOp | ::= | *Value*(Expr) \| *Dereference*(Expr) |
| | | \| *Field* (Expr) \| *Index* (Expr, Expr) |

**Lifetime operations:**

| | | |
|---|---|---|
| LifeOp | ::= | *link*(Expr, Expr) |
| | | \| *invalidateLinked*(Expr) |

**Figure 4: ARC++ Abstraction (Abstract Representation for C++). The terminal symbols `e` and `type` represents the expression and types in the C++ program, respectively.**

**Lifetime operations.** The operation $link(e_1, e_2)$ links the lifetime of $e_1$ with the operations of $e_2$. After the link operation, an $invalidateLinked(e_2)$ operation on $e_2$ causes the invalidation of $e_1$. For example, the $link(it, v)$ operation can be used to represent the fact that after the statement `it = v.begin()`, the lifetime of iterator `it` is linked with the lifetime of the vector `v`. Similarly, $invalidateLinked(v)$ can be used to represent the fact that `v.push_back()` invalidates the iterators that are linked with `v`.

**Dependency labels for member functions.** To generalize the bug detector for objects of user-defined classes, we automatically attach a dependency label to a member function that returns a pointer or a reference to the receiver object's internal buffer. For ease of presentation, we only consider fields of receiver objects as potential candidates.

**Def.** 2.1. *Given a member function `foo` returning a pointer or reference, the call `p = r.foo()` should be treated as an abstract operation that makes `p` dependent on the receiver `r`, if the memory accessible from a returned pointer or reference is destroyed whenever `r` is destroyed or cleared.* ⊠

Based on the above definition, if a field `f` of an object satisfies any of the following conditions, our technique attaches a dependency label to a member function that returns a pointer or a reference and satisfies one of these conditions:

1. If the member function returns a field `f` pointing to dynamically allocated memory whose lifetime is kept by the object (i.e., deleted in the object's destructor).
2. If the member function returns the address or reference of a field `f`, where the type of `f` is either an array or a user-defined class.

**Invalidate operations for member functions.** If a class `C` has a method that returns an internal buffer, then for a $Destroy(o)$ operation associated with an object `o` of class `C`, the $invalidateLinked(o)$ operation is also generated. We also generalize the *invalidateLinked* operation to other user-defined methods. If a method `foo` of a class reallocates any of the internal buffers that are returned by other methods of the class, then the method call `a.foo()` also causes the $invalidateLinked(a)$ to be generated.

# 3. TYPESTATES AND BUG PATTERNS

A bug pattern is specified as a finite state automaton over observable program operations. An observable operation is either a C++ statement or an ARC++ abstract instruction that changes the typestate corresponding to a bug pattern.

**Def. 3.1 (Bug Pattern).** *A bug pattern $\mathcal{P}$ is a tuple $\mathcal{P} = \langle Q, \Sigma, \tau, \mathsf{init}, \mathsf{error}\rangle$, where $Q$ is set of automaton states, $\Sigma$ is the set of observable operations in a program, $\tau \in (Q \times \Sigma \times Q)$ is the transition function mapping a state and an operation to a successor state, $\mathsf{init} \in Q$ is the unique initial state, and $\mathsf{error} \in Q$ is the unique error state.* ⊠

| | | |
|---|---|---|
| Objects | $o$ | $\in$ O |
| Identifiers | $id$ | $\in$ Identifiers |
| Labels | $l$ | $\in$ Labels |
| Fields | $fld$ | $\in$ Fields |
| Operations | $op$ | $\in$ Operations |
| Program | $pgm$ | $::= \overline{proc}$ |
| Procedure | $proc$ | $::= id(\overline{id})\{\overline{stmt}\}$ |
| Access path | $ap$ | $::= id \mid *ap \mid ap.fld$ |
| Statement | $stmt$ | $::= ap := ap \mid ap.op(\overline{ap}) \mid call\ id(\overline{ap}) \mid$ $ap := \&o \mid goto\ l_1\ l_2 \mid l : stmt$ |

**Figure 5: Mini Language. $\overline{x}$ is a list of $x$ terms.**

Our implementation described in §6 operates on C++ programs annoted with ARC++. However, to illustrate our approach, we use the MINI language shown in Fig. 5. We assume a set of identifiers Identifiers, a set of fields Fields, a (possibly infinite) set of dynamically allocated objects O, and a set of observable operations Operations over which the bug patterns are specified. A MINI language program $pgm$ consists of a list of procedures $proc$, and each $proc$ has a unique identifier $id \in$ Identifiers, a list of arguments and statements. In a MINI program, all objects are accessed through an access path, which is either an identifier, a memory dereference, or a field access. In the case of cyclic data structures, we limit the length of access paths by a constant. A MINI language statement $stmt$ is either an assignment from one access path to another, an observable operation, a call, initialization of an access path with the address of an object, non-deterministic goto, or a labeled statement. Let AP be the set of all access paths in the program, and Proc be the set of all procedures in a MINI program. We assume that transitive assignments induced by pointer dereferences and field accesses are present explicitly.

**Def. 3.2 (CFG).** *A procedure $p \in \mathsf{Proc}$ in the MINI language program is represented by a control-flow graph (CFG), which is a directed graph $G_p = \langle N_p, E_p\rangle$, where $N_p$ is a set of CFG nodes and $E_p \subseteq N_p \times N_p$ is a set of CFG edges. The set of nodes $N_p$ consists of a unique **start** node $s_p \in N_p$, a unique **exit** node $e_p \in N_p$, and nodes associated with each statement in the program.*

*Let $N = \cup_{(p \in \mathsf{Proc})} N_p$ be the set of* all *nodes in the program, $E = \cup_{(p \in \mathsf{Proc})} E_p$ be the set of* all *edges in the program,*

*$N_s = \cup_{(p \in \mathsf{Proc})} s_p$ be the set of **start** nodes in the program, and $N_e = \cup_{(p \in \mathsf{Proc})} e_p$ be the set of **exit** nodes in the program. Let $\mathsf{main} \in \mathsf{Proc}$ be the entry function. Finally, for each procedure $p \in \mathsf{Proc}$, let $\mathsf{Caller}_p \subseteq N$ be the set of call nodes that call $p$. Similarly, $\forall n \in N$, $\mathsf{Proc}[n]$ represents the procedure to which the node $n$ belongs to.* ⊠

Given a bug pattern $\mathcal{P}$, the concrete state of node $n \in N$ corresponding to a statement in the MINI language program consists of two parts: (1) an *object map* ($AP \to \mathsf{O}_\perp$) resolving each access to a concrete object, and (2) the *state map* ($O \to Q$) associating an automaton state with each concrete object, where $\mathsf{O}_\perp = \mathsf{O} \cup \{\perp\}$. The universe of all concrete states is represented by $\mathcal{S} = (AP \to \mathsf{O}_\perp) \times (O \to Q)$. Note that the initial state for the start node $s_{\mathsf{main}}$ of procedure $\mathsf{main}$ is $s_0 = \{(\lambda a.\perp) \times (\lambda o.\mathsf{init})\}$, which represents a state where the access paths do not resolve to any concrete objects, and all the objects are in the init state. Intuitively, a MINI language program is considered safe for $\mathcal{P}$ if and only if no concrete object is associated with an error state.

# 4. ACCESS-PATH CLUSTERS ABSTRACT DOMAIN

Since it is computationally intractable to compute the set of all concrete states precisely for a MINI program, we use abstract interpretation [14] to efficiently analyze programs. Abstract interpretation requires an abstract domain $\mathcal{D}$. An abstract domain $\mathcal{D}$ is a lattice $\langle \mathcal{D}, \sqsubseteq, \sqcup \rangle$ along with an abstraction map $\alpha : 2^{\mathcal{S}} \to \mathcal{D}$ and a concretization map $\gamma : \mathcal{D} \to 2^{\mathcal{S}}$ such that $\alpha$ and $\gamma$ form a *Galois Connection* between concrete lattice $2^{\mathcal{S}}$ and the abstract lattice $\mathcal{D}$.

One of the main challenges in defining an abstract domain for programs with pointers lies in incorporating aliasing relationships into the representation of the abstract object so that *strong updates* can be performed for common cases. A strong update *replaces* the current abstract value with a new abstract value, and is performed when the abstract interpreter can determine that the abstract state being updated correponds to a unique concrete state. On the other hand, a *weak update* only *appends* the new abstract value to the current abstract value. A large number of weak updates can adversely affect the precision resulting in false positives.

Our representation for an abstract object state is inspired by the work on aliasing-aware typestate verification developed for Java by Fink et al. [18]. Fink et al. use a notion of *access paths* to objects together with *focus operations* that split heap regions to selectively allow strong updates on certain abstract objects using these access paths. Fink et al. consider *must* and *must-not* access paths, such that successor abstract objects may be split up to gain a certain level of *path-sensitivity* [18]. We briefly present an adaption of their domain for our purposes, and in §5 describe how we extend it to deal with objects whose typestates are inter-dependent (such as LifeOp in §2).

To define an abstract object, we assume that results of a pointer analysis such as Steensgard's [38] or Andersen's [4] are available, and it supports the following operations:

- $\mathsf{mayPtsTo}(a, n)$ is the set of concrete objects pointed-to by the access path $a$ at node $n$. $\mathsf{mayPtsTo}(a)$ is the corresponding flow-insensitive information.
- $\mathsf{alias}(a_1, a_2, n)$ is $\mathsf{true}$ if $a_1$ and $a_2$ are possible aliases at node $n$, and $\mathsf{alias}(a_1, a_2)$ is the flow-insensitive version.

We define several concepts necessary for the abstraction.

**Def. 4.1 (Alias Cluster).** *Let $\mathsf{AP}$ be the set of all access paths in the program. An alias cluster $c \subseteq \mathsf{AP}$ is a set of access paths such that $\forall a_1, a_2 \in c$, $\mathsf{alias}(a_1, a_2)$. An alias cluster $c$ represents the set of objects pointed-to by the access paths in the cluster:*

$$[\![c]\!] = \{o \in \mathsf{O} \mid \exists a \in c, o \in \mathsf{mayPtsTo}(a)\}$$

*The universe of alias clusters $\mathcal{C} \subseteq 2^{\mathsf{AP}}$ is a partitioning of the access paths $\mathsf{AP}$ into alias clusters such that all of the following hold: (1) $\forall c \in \mathcal{C}$, $c$ is an alias cluster, (2) $\cup_{c \in \mathcal{C}}(c) = \mathsf{AP}$, and (3) $\forall c_1, c_2 \in \mathcal{C}, c_1 \cap c_2 = \varnothing \implies [\![c_1]\!] \cap [\![c_2]\!] = \varnothing$. In other words, $\mathcal{C}$ is a set of alias clusters such that each alias cluster $c \in \mathcal{C}$ represents a* distinct *set of concrete objects.* ⊠

**Def. 4.2 (Abstract Object).** *An abstract object $o^\sharp$ is a tuple $\langle c, \mu, \nu \rangle$, where $c \in \mathcal{C}$ is an alias cluster and $\mu, \nu \subseteq c \subseteq \mathsf{AP}$ is a set of access paths. An abstract object $o^\sharp$ characterizes a subset of concrete objects represented by the alias cluster $c$. Specifically, it represents the set of objects that are definitely pointed to by the access paths in $\mu$ (*must *set), but are* not *pointed to by the access paths in $\nu$ (*must-not *set).* ⊠

**Example** 4.1. *Fig. 6 shows three different abstract objects. Let $c = \{a_1, a_2, a_3\}$ be an alias cluster, $[\![c]\!]$ be the corresponding set of abstract objects (denoted by the three circles in Fig. 6). The must and must-not sets in an abstract object partition the objects of an alias cluster as shown in Fig. 6.*

*The abstract object $\langle c, \{a_1\}, \{a_2, a_3\} \rangle$ represents the yellow (lightest) region, which is the set of objects accessible by $a_1$, but* not *by $a_2$ and $a_3$. Similarly, $\langle c, \{a_1, a_2\}, \{a_3\} \rangle$ represents the red (darkest) region, which is the set of objects accessible by* both *$a_1$ and $a_2$, but* not *by $a_3$. Finally, $\langle c, \{a_1, a_3\}, \{a_2\} \rangle$ represents the green region.* ⊠

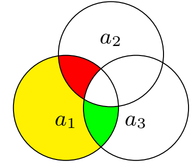

**Figure 6: Access-path clusters domain.**

We define *exclusive accessibility* to cover cases where strong updates can be performed.

**Def. 4.3 (Accessibility).** *An access path $a$ has access to abstract object $o^\sharp = \langle c, \mu, \nu \rangle$, denoted as $a \rhd o^\sharp$, if one of the following conditions hold: (1) $a \in \mu$, or (2) $a \notin \mu \land a \notin \nu$. Otherwise, we say $a$ has no access to $o^\sharp$, denoted as $a \ntriangleright o^\sharp$. Intuitively, $a \rhd o^\sharp$, if the set of objects accessible by $a$ overlap with the set of objects represented by $o^\sharp$, i.e., $[\![\langle c, \{a\}, \varnothing \rangle]\!] \cap [\![o^\sharp]\!] \neq \varnothing$.*

*If $a \in \mu$, the set of objects represented by $o^\sharp$ is entirely contained within the set of objects represented by $\langle c, \{a\}, \varnothing \rangle$. In such cases, we say that $a$ has* exclusive *access to $o^\sharp$, denoted $a \blacktriangleright o^\sharp$.* ⊠

**Example** 4.2. *In Fig. 6, $a_1 \blacktriangleright \langle c, \{a_1\}, \{a_2, a_3\} \rangle$, $a_3 \ntriangleright \langle c, \{a_1, a_2\}, \{a_3\} \rangle$. Finally, $a_2 \rhd \langle c, \{a_1\}, \varnothing \rangle$ because the set of objects pointed to by $a_2$ may overlap with the set of objects pointed to by $a_1$.* ⊠

The notion of accessiblity allows us to determine when a strong or weak update can be performed on an abstract object $o^\sharp$. Consider an automaton operation $a.op()$ at a node $n$ encountered during the analysis. The abstract interpreter has to update the state of abstract objects accessible through $a$ according to the state transitions of $op()$. If $a \blacktriangleright o^\sharp$ at $n$,

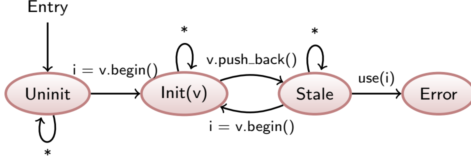


**Figure 7: Iterator pattern.**

then the analysis can perform a strong update, because $o^\sharp$ only represents objects accessible through $a$. On the other hand, if $a \triangleright o^\sharp$ then a weak update has to be performed because it may represent objects not accessible through $a$ as well. Furthermore, if $a \not\triangleright o^\sharp$ the state of $o^\sharp$ is unchanged.

To faciliate strong updates, an abstract object can be refined by using the focus operator, $\mathsf{focus}(\langle c, \mu, \nu \rangle, a) =$

$$\begin{cases} \langle c, \mu \cup \{a\}, \nu \rangle, \langle c, \mu, \nu \cup \{a\} \rangle & \text{if } a \in c \wedge a \notin \nu \wedge a \notin \mu \\ \langle c, \mu, \nu \rangle & \text{otherwise} \end{cases}$$

The focus operation splits a given abstract object $o^\sharp$ into $o^\sharp_1$ containing objects definitely accessible by $a$ and $o^\sharp_2$ containing objects that are not accessible by $a$. Therefore, an operation that is performed through the access path $a$ should update the state of $o^\sharp_1$ and leave the state of $o^\sharp_2$ unchanged.

**Example** 4.3. *Before applying an automaton operation $a.op()$ on an abstract object $o^\sharp$, the* focus *operator can be used to split $o^\sharp$ into $o^\sharp_1$ and $o^\sharp_2$ such that $a \blacktriangleright o^\sharp_1$ and $a \not\triangleright o^\sharp_2$. Therefore, we can perform a strong update on the abstract object $o^\sharp_1$ and leave the state of $o^\sharp_2$ unchanged.* ⊠

For scalable verification, access paths in the must and must-not sets may be discarded, thus merging various abstract objects potentially leading to imprecise (but still sound) verification results. A blur operation discard access paths [18].

The abstract domain $\mathcal{D} = \mathcal{C} \rightarrow 2^{\mathcal{C}^\sharp}$ is a map associating an abstract object with a set of automaton states. The $\sqcup$ and $\sqsubseteq$ on abstract objects and the set of automaton states can be extended pointwise in a straightforward manner to $\mathcal{D}$. $\langle \mathcal{D}, \sqsubseteq, \sqcup \rangle$ forms a lattice.

A *flow-sensitive* abstract map $\eta^\sharp : N \rightarrow \mathcal{D}$ associates each CFG node $n \in N$ with an abstract state map $s^\sharp \in \mathcal{D}$. Given an inductive map $\eta^\sharp$ with respect to a bug pattern $\mathcal{P}$, the Mini language program is safe for the bug pattern $\mathcal{P}$, if and only if $\forall m \in N, a \in \mathsf{AP} : \mathsf{error} \notin \eta^\sharp[m][n]$. To compute the fixpoint, we use the IFDS tabulation solver [37]. Except for the differences discussed in §5, the abstract transfer functions are similar to Fink et al. [18].

After the fixpoint is computed, if none of the nodes in the program have an error state, then the program does not have the given bug pattern. Otherwise, the program may have an execution matching the pattern.

# 5. LIFETIME DEPENDENCY ANALYSIS

Consider an access path $a \in \mathsf{AP}$ and its alias cluster $c \in \mathcal{C}$. Assume that an operation $a.op()$ only affects objects accessible through $a$ or its alias cluster $c$. However, there are bug patterns where an operation $a.op()$ also affects the typestates of objects in other alias clusters. Consider the iterator pattern shown in Fig. 7 that describes a pattern for iterator objects. Note that state `Init(v)` depends upon the container object `v` to which the iterator `i` is initialized. Thus, `Init(v)` in the automaton represents several concrete states, one for each possible container object. When the iterator is

```
1 i1 = v1.begin();     // {v1} ⇝ {i1}
2 i2 = i1;             // {v1} ⇝ {i1, i2}
3 i1 = v2.begin();     // {v1} ⇝ {i2}, {v2} ⇝ {i1}
4 v1.push_back();      // {v1} ⇝ {i2}, {v2} ⇝ {i1}
```

**Figure 8: Dependency information for an example.**

in `Init(v)` and operation `v.push_back()` is encountered, the iterator moves to the `Stale` state.

If the typestate analysis does not track the dependency between the `Init` state and the container `v`, the states of all iterators should be moved to the `Stale` state resulting in many false positives. Consider the example shown in Fig. 8. At line 4, statement `v1.push_back()` would be treated as invalidating both iterators `i1` and `i2` if we did not properly track dependencies. However, only `i2` is actually invalidated at line 4. To deal with this issue, we introduce the concept of dependency between abstract objects.

**Def. 5.1 (Dependency).** *An abstract object $o^\sharp_1$ is dependent on another object $o^\sharp_2$, if the operations on $o^\sharp_2$ affect the typestate of $o^\sharp_1$, and denoted as $o^\sharp_2 \rightsquigarrow o^\sharp_1$.* ⊠

By incorporating the dependency information, we can improve the typestate analysis. Consider Fig. 8, where the comments show the dependency information between iterators (`i1`,`i2`) and containers (`v1, v2`) after the execution of the corresponding statement. The dependency information can be used to invalidate iterator `i2` and leave the state of iterator `i1` unchanged at line 4.

The dependency information can be incorporated into the typestate analysis described in §3 and §4. In addition to the abstract-state maps, the analysis also maintains the set $D \subseteq \mathsf{O}^\sharp \times \mathsf{O}^\sharp$ of dependency edges between objects at every node in the CFG. If $(o^\sharp_2, o^\sharp_1) \in D$ then $o^\sharp_2 \rightsquigarrow o^\sharp_1$.

Consider an automaton operation such as $b = a.op()$ that makes the state of the abstract objects accessible through $b$ to be dependent on the abstract objects accessible through $a$. For such operations, the set of dependency edges $D$ is updated as follows:

$$\begin{aligned} D \ := \ & D \backslash \left\{ o^\sharp_2 \rightsquigarrow o^\sharp_1 \mid a \blacktriangleright o^\sharp_2 \vee b \blacktriangleright o^\sharp_1 \right\} \\ & \cup \left\{ o^\sharp_2 \rightsquigarrow o^\sharp_1 \mid (a \triangleright o^\sharp_2) \wedge (b \triangleright o^\sharp_2) \right\} \end{aligned}$$

First, the existing dependency information for objects for which either $a$ or $b$ has exclusive access is removed and the new dependency information is added between objects for which either $a$ or $b$ has exclusive access. By removing the existing dependency relations for abstract objects with exclusive access, the operation performs a strong update. For all other statements, whenever a new abstract object $o^{\sharp\prime}$ is added through a $\mathsf{focus}(o^\sharp, a)$ or $\mathsf{blur}(o^\sharp, a)$ operator the dependency information of $o^\sharp$ is copied over to $o^{\sharp\prime}$.

**Example** 5.1. *Consider line 3 in Fig. 8, where a new dependency edge between* `v2` *and* `i1` *is added. During the lifetime dependency analysis, the edge* `v1` $\rightsquigarrow$ `i1` *is removed and the edge* `v2` $\rightsquigarrow$ `i1` *is added.* ⊠

**Uses of Lifetime Dependency Analysis.** We are interested in discovering the stale use of dependent objects. Thus, we consider *lifetime dependencies*: An object $o_2$ is lifetime dependent on another object $o_1$, if an operation on $o_1$ or its modification or destruction directly implies the destruction of object $o_2$. These types of lifetime dependencies

**Figure 9: Analysis for the stale string use in Fig. 3**

are especially interesting in the context of a language that relies on low-level memory management.

**Example** 5.2. *Consider the stale buffer access shown in Fig. 3. We illustrate the lifetime dependency analysis in Fig. 9. On the left we show the source code simplifications, which introduce explicit calls to all implicit function calls and temporary object creation as needed. For example, the original statement* `str=s.substr(i,n).c_str();` *is de-sugared into three statements including the copy-construction of a temporary variable called* `tmp` *and finally its destruction after the call to* `c_str()` *is completed.*

*On the right we show the state of the two relevant objects and pointers of interest, namely the C++ string object* `tmp`*, and the C string* `str`*. We assume that the C++ string* `s` *is valid throughout. Assume that for C++ strings, we have typestates* `GoodStr` *to represent a properly initialized string, and* `UninitStr` *to denote an uninitialized or deallocated string. We have similar states for C strings, which are denoted* `GoodCStr` *and* `UninitCStr`*.*

*We show the typestates for the two objects/access path clusters of interest here at the beginning of the sequence of statements. The interesting step is after the call to* `c_str()`*, where a lifetime dependency edge (LTDE) is added denoting the dependency of the abstract object* `(GoodCStr,{str})` *on the abstract object* `(GoodStr,{tmp})`*. Then, the destruction of* `tmp` *in the following step not only destroys* `tmp`*, but also* `str` *via dependence analysis. Thus, we can easily identify the stale use of* `str` *in the following line.* ⊠

*Internal-buffer pattern* refers to bugs, where a deallocated object's internal buffer is accessed. The iterator pattern shown in Fig. 8 is an internal-buffer pattern, because an iterator is a pointer to an internal buffer of some container. When the container is destroyed, its internal buffer is deallocated and the iterator becomes stale.

## 6. IMPLEMENTATION

We have implemented a framework (shown in Fig. 11) that automatically generates Arc++ and uses it for typestate checking to find bugs in C++ programs. A given C++ program is initially translated by our frontend based on EDG [17] into a simplified version of C++ called Cilpp [41]. Cilpp is similar to CIL [30] with suitable additions for C++ constructs such as inheritance and exceptions. The exceptions are handled by the IECFG module [33], which generates an inter-procedural exceptional control flow graph that is similar to the CFG defined in Defn. 3.2, with additional edges related to exceptional flows. The new module Arc++ `Gen.` then automatically generates the Arc++ abstract representation, where each abstract instruction is associated with the control flow information of the IECFGs. With bug patterns provided by users, the framework automatically performs typestate checking on the Arc++ representations and reports detected bugs.

```
type States = Init | Destroyed | Error // Automaton States
let initial = Destroyed // Initial state
let error = Error // Error state
// Transfer function
let transfer (absinstr, inp) =
  match absinstr with
    | Create (id, Pointer(Iterator, e,_), args)
    | Create (id, Container(e, _,_,_), args) →
       // Update state of objects accesses by e to Init state
       updateState(inp, e, Init)
    | Use (id, Dereference(Pointer(Iterator, e, _)) →
       // If e is in Destroyed state, move to Error
       updateState(inp, e, Destroyed, Error)
    | Destroy (id, Pointer(Iterator, e,_), args)
    | Destroy (id, Container(e, _,_,_), args) →
       // Update state of objects accessed by e to Init state
       updateState(inp, e, Error)
    | _ → inp // Anything else, no change
```

**Figure 10: Iterator pattern specified using ARC++.**

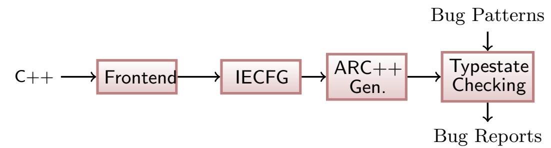


**Figure 11: Automatic ARC++ Framework.**

**Specifying Bug Patterns.** In our framework, the automata for bug patterns are specified in an ML-like language using instructions from Arc++. Fig. 10 shows the bug pattern for detecting invalid use of iterators. First the set of states along with initial and error states are defined. The state transitions for the automaton are provided by the `transfer` function. In `transfer`, Arc++ instructions are used to identify the C++ statement of interest and an appropriate action is performed. `updateState` is an API that describes the state transitions for the typestate analysis.

Given a program and a bug pattern, our framework performs typestate analysis on the program to check the bug pattern. The analysis reports any error transition to the user. Note that the iterator pattern does not specify the dependencies between iterators and containers. This is taken care of during the Arc++ generation, and the lifetime operations *link* and *invalidateLinked* (described in §2) are added to the appropriate C++ statements. For example, *link(it, v)* is generated for `it = v.begin()`, and *invalidateLinked(v)* is generated for `v.push_back()`. The typestate analysis automatically takes these operations into account.

## 7. EVALUATION

To evaluate the effectiveness of the presented techniques, we have implemented some common bug patterns using our Arc++ framework and conducted evaluations on some microbenchmarks (see `http://tinyurl.com/c4vs3x9`) and several open-source C++ projects. We seek to answer the following research questions:

- **RQ1**: Can the bug patterns based on typestates and syntactic checking detect real bugs?
- **RQ2**: Can the bug patterns based on our lifetime dependency analysis detect real bugs?
- **RQ3**: Can our analysis scale to open-source C++ programs of large sizes?

Table 1: Evaluation subjects and analysis time of bug patterns (in seconds).

| Benchmark | # Files | LOC | IECFG | Ptr | ARC++ | Lock | Iterator | c_str() | Aptr | Copy | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Micro-Benchmarks | 15 | 317 | <1 | <1 | <1 | <1 | <1 | <1 | <1 | <1 | <1 |
| TinyXML | 5 | 4,748 | 0.35 | 0.22 | 0.19 | 0.38 | 0.57 | 0.71 | 0.33 | 0.004 | 2.75 |
| motti-3.0.0 | 30 | 7,511 | 0.95 | 0.35 | 0.24 | 0.17 | 0.68 | 0.44 | 0.17 | 0.085 | 3.09 |
| AtomicParsley-0.9.0 | 8 | 8,157 | 0.2 | 0.16 | 0.16 | 0.08 | 0.06 | 0.12 | 0.04 | 0.002 | 0.82 |
| flac-1.2.1 | 12 | 11,395 | 0.75 | 0.61 | 0.54 | 0.72 | 1.14 | 1.11 | 0.64 | 0.016 | 5.53 |
| TUIO_CPP-1.4 | 15 | 13,704 | 0.48 | 0.36 | 0.32 | 0.78 | 2.76 | 0.71 | 0.41 | 0.131 | 5.95 |
| sipp.svn-3.2 | 17 | 17,145 | 1.24 | 0.91 | 0.5 | 0.47 | 1.57 | 1.16 | 0.4 | 0.309 | 6.56 |
| gnuchess-6.0.1 | 66 | 17,467 | 0.96 | 0.19 | 0.14 | 0.17 | 0.16 | 0.29 | 0.13 | 0.004 | 2.04 |
| reacTIVision-1.4 | 45 | 25,677 | 0.47 | 0.25 | 0.23 | 0.28 | 0.63 | 2.23 | 0.26 | 0 | 4.35 |
| ultimatestunts-0751 | 142 | 28,378 | 0.67 | 0.24 | 0.11 | 0.09 | 0.46 | 4.43 | 0.05 | 0.271 | 6.32 |
| gama-1.11 | 73 | 37,080 | 6.81 | 6.77 | 4.73 | 8.13 | 31.3 | 29.32 | 7.38 | 0.052 | 94.49 |
| stella-3.5 | 157 | 46,013 | 2.02 | 0.89 | 0.69 | 0.48 | 1.06 | 1.25 | 0.38 | 0 | 6.77 |
| faust-0.9.46 | 179 | 53,853 | 9.45 | 9.4 | 6.45 | 11.93 | 28.16 | 25.28 | 10.64 | 0.014 | 101.32 |
| rcssserver-15.0.1 | 97 | 62,303 | 17.97 | 15.59 | 4.57 | 12.77 | 21.96 | 19.47 | 14.12 | 0.008 | 106.46 |
| ode-0.12 | 184 | 82,917 | 7.25 | 12.69 | 8.81 | 11.76 | 16.19 | 16.05 | 10.04 | 0.005 | 82.80 |
| amos-3.1.0-rc1 | 239 | 85,891 | 39 | 33.42 | 14.28 | 31.17 | 196.35 | 96.84 | 28.34 | 0.093 | 439.49 |
| cppcheck-1.51 | 118 | 88,314 | 7.58 | 10.09 | 6.51 | 7.19 | 43.38 | 76.85 | 5.74 | 0.026 | 157.37 |
| p7zip_9.20.1 | 352 | 109,261 | 6.67 | 4.48 | 2.92 | 7.03 | 9.67 | 8.36 | 6.22 | 0 | 45.35 |
| cmake-2.8.6 | 479 | 163,277 | 14.3 | 15 | 8.02 | 21.78 | 78.29 | 135.38 | 20.58 | 0.01 | 293.36 |
| Greenstone | 286 | 263,228 | 15.18 | 9.66 | 5.48 | 10.86 | 81.17 | 15.01 | 9.99 | 0.992 | 148.34 |
| scummvm-1.2.1 | 1310 | 1,275,768 | 46.58 | 74.46 | 53.27 | 80.51 | 133.84 | 127.9 | 69.3 | 5.35 | 591.21 |
| Total | 3,829 | 2,402,404 | 178.90 | 195.70 | 118.20 | 206.80 | 649.40 | 562.91 | 185.20 | 7.37 | 2,104.40 |

**Subjects.** We have evaluated the techniques on micro-benchmarks and selected modules of some open-source benchmarks. The micro-benchmarks consist of 16 C++ programs that were handcrafted based on bugs that are encountered in real programs and good coding practice guidelines, such as *Effective C++* [28] and *Dark Side of C++* [40]. These small programs misuse STL libraries and can be used to test the detection capabilities of bug finding tools.

The open-source benchmarks consist of 20 open-source C++ projects. The lines of code (LOC) and the number of files of the projects are shown in Tab. 1. These projects are sampled from various domains, such as games (e.g., GNU Motti and GNUChess), parsers (e.g., TinyXML and AtomicParsley), simulators (e.g., Gama and Stella), and analysis tools (e.g., Cppcheck).

**Evaluation Setup.** For sake of brevity, we present 5 bug patterns that were checked using our ARC++ framework. Some of these bug patterns capture bad programming practices, and the other bug patterns capture invalid memory accesses, resource leaks and misuses of certain APIs:

- **Auto_Ptr Pattern.** This patterns identifies invalid usage of *auto_ptr* as described in §1.
- **Lock Pattern.** This pattern identifies invalid usage of pthread locks, such as double locking and potential deadlocks on exceptions.
- **Overloaded-Operator Pattern**. This pattern identifies bad programming practices of overloading the assignment operator '=' [28].
- **Copy-Constructor Pattern**. This pattern identifies the bad programming practice where a class contains fields that point to dynamically allocated memory, but does not contain a user-defined copy constructor [28].
- **Dependency Patterns.** Dependency patterns include the iterator pattern and the internal-buffer pattern of `c_str()` described in §5.

### *RQ1: Bug Pattern Analysis.*

To address RQ1, we measure the number of bugs detected by the following bug patterns: `auto_ptr`, `pthread` locks, and overloading assignment operator (=). Tab. 2 shows the results of RQ1. Column "Pattern" shows the name of the bug pattern. Columns "Micro" and "OSS" show the number of bugs (*excluding duplicates*) and the number of false positives in brackets found in the benchmarks. We next show some example bugs detected by these patterns and discuss the reasons for the false positives.

**Table 2: Number of real bugs found by patterns (false positives in brackets).**

| Pattern | Micro | OSS |
|---|---|---|
| Overload Op | 3 (0) | 5 (1) |
| Auto_ptr | 2 (0) | 1 (1) |
| Lock | 4 (0) | 2 (2) |
| Copy Ctr | 1 (0) | 118 (21) |
| Iterator | 3 (0) | 15 (25) |
| c_str() | 2 (0) | 6 (6) |
| Total | 15 (0) | 147 (56) |

```
1 class Listener : protected BaseObserver< AudioSender > {
2   Listener( std::auto_ptr< AudioSender > sender )
3     : BaseObserver < AudioSender >( sender ) {}
4   ... };
```

**Figure 12: A bad practice detected by the auto_ptr pattern in RCSS Server.**

**`auto_ptr` pattern.** Fig. 12 shows a bad practice detected by the auto_ptr pattern in the project RCSS Server. The class `Listener` accepts an `auto_ptr sender`, and calls the base class constructor by passing the `auto_ptr sender` by value. Due to the ownership transfer, any subsequent usage of `sender` in the class `Listener` is invalid. In this case, it is error-prone to extend class `Listener` if `sender` is used in any member function of `Listener`. Although we have not found any invalid usage of `sender`, we consider it a bad practice since it is error prone. The other error was a false positive and was caused by the imprecision in the pointer analysis.

**Lock pattern.** Fig. 13 shows the simplified code example of a detected bug in the project TUIO_CPP. Each iteration of the first loop (lines 1-8) starts by locking the `objectList` (line 2), and ends with unlocking the `objectList` (line 7). However, due to the *break* statement at line 5, the loop may end its iterations with the `objectList` locked. If the method `refresh` at line 10 throws an exception, the program

```
1  for( ... ) {
2    lockObjectList(); // lock objectmutex
3    std::list<TuioObject*>::iterator iter;
4    for (iter=objectList.begin(); iter != objectList.end();
           iter++) {...}
5    if(iter==objectList.end()) break; // Jump out of loop
6    ...
7    unlockObjectList(); // unlock objectmutex
8    ...}
9  for (std::list<TuioListener*>::iterator
         listener=listenerList.begin();...)
10   (*listener)->refresh(currentTime);//exception happens
```

**Figure 13: Lock pattern in TUIO_CPP.**

execution will go to the exception handler (not shown here) without unlocking the `objectList`. The exception handler also catches exceptions on paths with `objectList` not being locked, and thus does not unlock the object. This situation will likely lead to a deadlock after the exception is handled. Detecting these kinds of bugs requires the analysis of exceptional control flows, and few existing bug-finding tools [26, 21, 35, 3] have the capabilities to do that. Our lock pattern also detects double locking bugs for TUIO_CPP, and has 3 false positives due to a lack of enough path-sensitivity.

**Overloaded-operator and Copy-Constructor patterns.** In the micro-benchmarks, the overloaded-operator pattern detects 2 bugs, where the operator does not return a reference to the receiver object, and another bug where the assignment operator of a base class is not invoked in the assignment operator of a derived class. The copy-constructor pattern detects a bug where a field of a class is allocated with dynamic memory, but a copy constructor is missing. We use the typestate analysis to check whether a field is allocated with dynamic memory at the end of the constructor.

In the open-source projects, the overloaded-operator pattern detects 5 bugs with 1 false positive, and the copy-constructor pattern detects 57 with 10 false positives. Fig. 14 shows the overloaded-operator bugs found by our framework in the project Gama. Struct `E_3` does not overload the assignment operator and the default generated one returns a reference to the object itself. Thus, objects of `E_3` support *chained assignments*, such as `x = y = z`. However, `E_3` overloads the compound assignment operators (such as '+=' and '-='), and returns *void*. This causes an inconsistency in using `E_3`, and violates the rule that these overloaded operators should return a reference to the object itself.

The overloaded-operator pattern detects 5 bugs while cppcheck [1] detects only 3 of them, because the CHROME object representation used in our framework effectively addresses the challenges posed by inheritance and method overrides. Both our analysis and cppcheck have 1 false positive in RCSS Server. This is caused by the presence of an internal struct `Holder` in class `RCSSCLangLexer`. `Holder` overloads operator '=' for facilitating the lexical processing of different types of tokens, thus making it inappropriate for operator '=' to return the reference to the `Holder` object.

**Answer:** *Our analysis based on the typestates and syntactic checking can find bugs and bad programming practices in open-source C++ projects.*

### *RQ2: Lifetime Dependency Analysis.*

To address RQ2, we measure the number of bugs detected by the bug patterns of iterator and `c_str()`. The number of bugs detected by our framework is reported in Tab. 2. Our framework finds 3 iterator bugs in the micro-benchmarks, and 15 bugs in the open-source benchmarks. These bugs are caused by the use of operations on containers that invalidate the iterators such as `erase`, `push_back`, and `pop_back`. Fig. 15 shows a bug using a stale iterator. Invoking `push_back` at line 3 may cause reallocation of the vector `text`, which will invalidate the iterator `next`. Thus, the usage of `next` at line 5 can result in a stale iterator use. Our analysis links the lifetime of `next` to the vector `text`, invalidates `next` when `text.push_back` is invoked, and correctly detects the invalid usage at line 5.

```
1  struct E_3 {
2    ...
3    void operator+=(const E_3&);
4    void operator-=(const E_3&);
5    void operator*=(double); }
```

**Figure 14: Overloaded operator bug in Gama.**

```
1  UCArray::iterator next = text.begin();
2  ...
3  text.push_back(':'); // may invalidate next
4  ...
5  for (++next; *next != ':'; ++next) {...}
```

**Figure 15: A stale iterator use in Amos-3.1.0.**

In the Greenstone project, the containers are wrapped in custom classes, and thus operations that invalidate iterators happen in some callee of the function in which the operator is used. Even so, our analysis is able to propagate the effects of invalidation up the call stack and detect the bug. However, tools like cppcheck [1] that perform only syntactic checking cannot detect such bugs. In fact, cppcheck detected only 5 out of the 15 iterator bugs reported by our framework.

In the open-source benchmarks, our framework reported 40 bugs for the iterator pattern, of which 15 were real bugs and 25 were false positives. Similarly, our framework reported 12 `c_str()` bugs, of which 6 were real bugs and 6 were false positives. These false positives are caused by lack of path sensitivity, use of blur operations to throw away access paths as discussed in §4, or the imprecision in pointer analysis. For the iterator pattern, some false positives are due to "smart" usage of iterators in the code, as shown in Fig. 16. Note that erasing elements from a `vector` invalidates only iterators pointing after the point of modification. In this example, however, the `vector` is walked in a reverse fashion. In future work, we plan to investigate heuristics to prune such cases.

**Answer:** *Our lifetime dependency analysis discovers a number of iterator and `c_str()` bugs that result in invalid use of stale objects.*

### *RQ3: Scalability Analysis.*

To address RQ3, we measure the analysis time of each bug pattern and the execution time of the modules in our framework. Tab. 1 shows the results of RQ3. Columns "IECFG", "Ptr", and "ARC++" show the analysis time for constructing IECFGs, point-to analysis, and generating ARC++ representations, respectively. Columns "Lock", "Iterator", "c_str()", "Aptr", "Copy" show the analysis time for the corresponding bug patterns. Column "Total" shows the total time of preprocessing and pattern checking for each project. The analysis of the micro-benchmarks and the Overload-Operator pat-

```
1 vector<string>::iterator iterBegin = filenames.begin();
2 for (int i = (int)filenames.size() − 1; i >= 0; i−−) {
3   ...
4   if (matcher.Match(filenames[(unsigned int)i],
        caseSensitive))
5     filenames.erase(iterBegin + i); }
```

**Figure 16: An iterator false positive from cppcheck.**

tern based on syntactic checking is very fast. For most of the projects it takes less than 0.01 second to finish. Thus, we do not show the execution time for them. The copy-constructor pattern could not be checked for projects reacTIVision-1.4, p7zip 9.20.1, and stella-3.5 due to some failures in earlier modules unrelated to the typestate checker. For this reason, the execution time of this pattern for these projects is shown as 0. To reduce false positives, our framework also performs liveness analysis that helps remove invalid aliasing relationships. The execution time of the liveness analysis is similar to pointer analysis and is not shown here.

The results show that for the patterns without dependency analysis (Auto_ptr and Lock), the maximum analysis time is 80.51 s. For the dependency analysis patterns, the maximum analysis time is a bit higher (the Iterator pattern requires 196.35 s), but it is still reasonable. Moreover, the total time required for the pre-processing steps is just 374.60 s, and the total time for the pre-processing and typestate analysis in all the projects is 2104.40 s.

**Answer:** *Our framework completes the analysis of open-source benchmarks of 2M LOCs in a reasonable time.*

*Threats to Validity.*

**Threats to external validity.** We evaluated our program analysis techniques on a large variety of benchmarks. However, we cannot guarantee that the set of benchmark is representative of all domains. To mitigate this limitation, we strived to include a large range of benchmark applications, with varying application domains and of varying sizes. Moreover, we also evaluated our program analysis techniques on micro-benchmarks that are extracted from real bugs and other bug sources, such as Effective C++ [28].

**Threats to internal validity.** One internal validity threat is the correctness of our implementation to preprocess and analyze C++. We rely on the correctness of our CILPP framework for C++, which has been thoroughly tested and is in production use for program analysis within NEC in a tool called Varvel [23, 27]. We have also inspected the newly developed code for correctness.

## 8. RELATED WORK

Our work is quite closely related to pattern-based bug finding tools for Java. FindBugs [21] is a bug pattern detector for Java, which syntactically matches code to suspicious programming practice, in a manner similar to ASTLog [15]. PMD [3] and Jlint [5, 2] perform syntactic checks to analyze Java, and find problems of unused variables, unnecessary object creation, etc. In addition, Jlint detects synchronization problems using data flow analysis. While these tools focus on Java, our bug-finding framework handles more complex C++ semantics, and performs typestate analysis and dependency analysis over ARC++ representations to detect bugs. The state-machine framework of LLVM/Clang [25] provides a possibility for bug pattern checking. However, based on our experience with open-source projects, pointer analysis, object abstraction, and the reasoning about lifetimes and dependencies are critical for effective typestate analysis.

There also exist bug finding approaches for C++. Rose is an open and extensible source-to-source compiler infrastructure [34], and Quinlan et al. [35] propose approaches to search for bug patterns based on Rose's interface to the abstract syntax tree (AST). Cppcheck [1] is a static analysis tool for C++ code, which allows users to specify bug patterns using regular expressions. While these two tools detect bugs by searching for patterns in the AST or matching code using regular expressions, our framework further supports typestate-based bug patterns based on typestate analysis. PR-Miner [26] mines implicit programming rules and detects violations for C code. PR-Miner cannot be applied directly on C++ code, since it does not handle the specific challenges brought by C++, such as exceptional control flows shown in Fig. 13. Coccinelle [32] provides a semantic patch language for specifying desired matches and transformations in C code, which allows automatic bug detection based on the specified semantic patches. The semantic patches used by Coccinelle are quite different from our typestate analysis with dependency analysis. To perform a typestate-like analysis of multiple interacting objects, Naeem and Lhoták propose an abstract representation for objects [29], which is similar to our access-path clusters domain. While they only allow access paths that represent local variables, our domain allows arbitrary access paths, which is required for analyzing C++ programs.

Model checking and theorem proving techniques are also explored to detect bugs. Bandera is a Java verification tool based on model checking and abstraction [13]. ESC/-Java [19] performs formal verification of properties of Java source code, and allows developers to add preconditions, post-conditions, and loop invariants in the form of special comments. Blanc et al. [11] et al. propose an operational model of the behavior guaranteed by the STL standard and apply predicate abstraction to a modified C++ program for verification. Blast [9] provides an observer specification language for users to specify temporal safety properties of C programs, which allows specification of type states and checking of type states. Besides typestate analysis, our framework provides a dependency analysis to support bug patterns that require dependencies among several objects. Moreover, these tools target either Java or C, while we address C++. Finally we also note recent work on theorem proving based approaches for C++ compiler certification in the CompCert project [36].

## 9. CONCLUSIONS

In this paper, we address the static analysis of complex C++ programs with an abstract representation (ARC++) and aliasing-aware typestate analysis techniques. We introduced a notion of lifetime dependency to target complex C++ lifetime semantics by proposing a lifetime dependency analysis. Finally, we show that an implementation of our techniques can find many interesting bugs in open-source projects. For the future, we will be extending our implementation on top of EDG [17] and CILPP [41] to handle programs written using the new C++11 standard. In fact, newly introduced smart pointer types such as `std::unique_ptr` and other changes and library additions will lead to new bug patterns as well.

## 10. REFERENCES


[1] Cppcheck. http://cppcheck.sourceforge.net/.

[2] Jlint. http://artho.com/jlint.

[3] PMD/Java. http://pmd.sourceforge.net.

[4] L. O. Andersen. Binding-time analysis and the taming of C pointers. In *Proc. PEPM*, 1993.

[5] C. Artho. Finding faults in multi-threaded programs. Master's thesis, 2001.

[6] G. Balakrishnan, N. Maeda, S. Sankaranarayanan, F. Ivančić, A. Gupta, and R. Pothengil. Modeling and analyzing the interaction of C and C++ strings. In *Proc. FoVeOOS*, 2011.

[7] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, 2002.

[8] N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *Proc. OOPSLA*, 2008.

[9] D. Beyer, A. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The Blast query language for software verification. In *Proc. SAS*, 2004.

[10] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Proc. OOPSLA*, 2007.

[11] N. Blanc, A. Groce, and D. Kroening. Verifying c++ with stl containers via predicate abstraction. In *Proc. ASE*, 2007.

[12] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, 2008.

[13] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. ICSE*, 2000.

[14] P. Cousot and R. Cousot. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL*, 1977.

[15] R. F. Crew. ASTLog: A language for examining abstract syntax trees. In *Proc. DSL*, 1997.

[16] R. DeLine and M. Fähndrich. Typestates for objects. In *Proc. ECOOP*, 2004.

[17] Edison Design Group. C++ Frontend. www.edg.com.

[18] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *Proc. ISSTA*, 2006.

[19] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI*, 2002.

[20] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. PLDI*, 2005.

[21] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Proc. OOPSLA*, 2004.

[22] R. Huuck, A. Fehnker, S. Seefried, and J. Brauer. Goanna: Syntactic software model checking. In *Proc. ATVA*, 2008.

[23] F. Ivančić, G. Balakrishnan, A. Gupta, S. Sankaranarayanan, N. Maeda, H. Tokuoka, T. Imoto, and Y. Miyazaki. DC2: A framework for scalable, scope-bounded software verification. In *Proc. ASE*, 2011.

[24] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Proc. PLDI*, 2012.

[25] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proc. CGO*, 2004.

[26] Z. Li and Y. Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. ESEC/FSE*, 2005.

[27] N. Maeda, T. Imoto, F. Mitsuhashi, F. Ivančić, G. Balakrishnan, and A. Gupta. VARVEL: A scalable software model checker. In *Proc. ISSRE (Industry Practice Track)*, 2011.

[28] S. Meyers. *Effective C++: 55 Specific Ways To Improve Your Programs And Designs*. Addison-Wesley Professional Computing Series. Addison-Wesley, 2005.

[29] N. A. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. In *Proc. OOPSLA*, 2008.

[30] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. CC*, 2002.

[31] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. ICSE*, 2007.

[32] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Proc. Eurosys*, 2008.

[33] P. Prabhu, N. Maeda, G. Balakrishnan, F. Ivančić, and A. Gupta. Interprocedural exception analysis for C++. In *Proc. ECOOP*, 2011.

[34] D. J. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10, 2000.

[35] D. J. Quinlan, R. W. Vuduc, and G. Misherghi. Techniques for specifying bug patterns. In *Proc. PADTAD*, 2007.

[36] T. Ramananandro, G. D. Reis, and X. Leroy. A mechanized semantics for C++ object construction and destruction, with applications to resource management. In *Proc. POPL*, 2012.

[37] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. POPL*, 1995.

[38] B. Steensgaard. Points-to analysis in almost-linear time. In *Proc. POPL*, 1996.

[39] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *TSE*, 12(1), 1986.

[40] F. von Leitner. Dark side of C++. http://port70.net/ñsz/16_c++.html.

[41] J. Yang, G. Balakrishnan, N. Maeda, F. Ivančić, A. Gupta, N. Sinha, S. Sankaranarayanan, and N. Sharma. Object model construction for inheritance in C++ and its applications to program analysis. In *Proc. CC*, 2012.