Detecting Element Accessing Bugs in C++ Sequence Containers

Zhilin Li
Key Lab. of System Software (CAS) and State Key Lab. of Computer Science, Ins. of Software, CAS
University of Chinese Academy of Sciences
Beijing, China
lizl@ios.ac.cn

Xutong Ma
Key Lab. of System Software (CAS) and State Key Lab. of Computer Science, Ins. of Software, CAS
Beijing, China

Mengze Hu
Key Lab. of System Software (CAS) and State Key Lab. of Computer Science, Ins. of Software, CAS
Beijing, China

Jun Yan*
Key Lab. of System Software (CAS) and State Key Lab. of Computer Science, Ins. of Software, CAS
Tech. Center of Software Eng., Ins. of Software, CAS
University of Chinese Academy of Sciences
Beijing, China
yanjun@ios.ac.cn

ABSTRACT

Sequence Containers (SC) in the C++ Standard Template Library (STL), such as the `vector`, are widely used in large-scale projects for their maintainability and flexibility. However, accessing the elements in an SC is bug-prone, as such operations will not check their boundaries during compilation or execution, which can lead to memory errors, such as buffer overflow problems. And these bugs are difficult to detect with available static analyzers, since the size of SCs and the target of iterators cannot be precisely tracked without a cooperative model for them.

To address this problem, we propose a combined model of SC sizes and iterator targets by tracking them simultaneously through a set of meta-operations extracted from corresponding method calls, and report improper operations according to three bug patterns. We implement the approach as a static analyzer, *Scasa*, on the top of the Clang Static Analyzer (CSA) framework, and evaluate its effectiveness and efficiency against CSA and other state-of-the-art static analyzers on a benchmark composed of 2230 manually created code snippets and eight popular open-source C++ projects with a lot of SC usage. The experimental results reveal that *Scasa* effectively identifies nearly all inherent bugs within the manual code snippets and generates 125 reports for these projects (with a time loss of 5–85%) where 72 reports are marked as correct with a manual revision. And to further confirm these correct reports, we also select some important ones for developers. These results show that accessing elements of SCs is bug-prone, and cooperatively tracking SC sizes and iterator targets can accurately detect these bugs with acceptable overhead.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**.

*Corresponding author.

ASE '24, 27 October – 1 November, 2024, Sacramento, California
2018. ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

KEYWORDS

Sequence Container, Element Accessing Bugs

1 INTRODUCTION

C++ is widely used across critical domains, especially in computer science, where its support for object-oriented programming and generics, along with its efficient execution speed [27], are invaluable. However, early versions of the C++ standard lack built-in implementations for complex structures, requiring programmers to manually implement them. The Standard Template Library (STL) in C++ is introduced to tackle this challenge. STL consolidates the implementation of universal data structures and offers extensive algorithmic support. It has been internationally recognized as a standard since C++98 and has evolved into a vital component of C++ libraries [16].

However, the utilization of the standard template library does not preclude all errors in data structures. There is also the potential for misuse by programmers [7, 14], which can occasionally result in undefined behavior, leading to program outcomes that deviate from expectations or even program crashes. Misuse of standard template libraries often does not manifest as compilation errors, meaning that no errors are reported during compilation, potentially resulting in the oversight of misuse. Among STL containers, sequence containers such as vectors and lists are susceptible to these issues. A shared characteristic of sequence containers is their support for sequential access, allowing users to traverse the entire container through iterator addition and subtraction. Consequently, element accessing bugs are more likely to occur in sequence containers, including *Iterator Out-of-Bound Access* bugs, *Invalid Iterator Access* bugs, and *Empty Container Access* bugs, with such bugs being less common in non-sequential containers. In this paper, we call all these Element Accessing bugs in Sequence Containers EASC bugs. We count the number of projects or commits about the keywords related to EASC bugs on Github in the past decade. The total count of relevant commits since 2013 is depicted in Figure 1. The figure illustrates a substantial number of commits associated with iterator

bugs or sequence container bugs since 2013, with no apparent slowdown in growth rate. Consequently, devising an approach capable of effectively detecting these bugs holds significant importance.

Static analysis represents an effective approach for detecting program defects. However, the experiments, as detailed in Section 4.4, reveal that off-the-shelf tools fail to deliver satisfactory results. Coding style checker CppCheck [3] is lack of model of iterators, which results in failure when determining whether the iterator is out of bounds. Static analyzers Clang Static Analyzer [1] lacks a model of sequence container size, so all checks requiring size are not accurate. Additionally, Facebook Infer [5] even fails to detect such errors altogether.

The deficiencies observed in off-the-shelf analyzers can be categorized into two main challenges: models and checkers. Firstly, sequence containers in the STL are not just standalone containers, they involve a combination of containers, iterators, and STL methods. Consequently, static analysis of sequence containers in STL necessitates not only a pertinent model for containers but also models for the associated iterators and methods. The interconnectedness of these components requires the model to exhibit robust interactivity and responsiveness to relevant changes. However, off-the-shelf analyzers struggle to address this challenge effectively. Secondly, new checkers must be expanded specifically to detect EASC bugs. The absence of comprehensive models leads off-the-shelf analyzers' checkers to adopt relatively conservative judgment criteria for this category of bugs, resulting in a significant number of misses.

In response to these limitations, we introduce the models for sequence containers and checkers designed to check EASC bugs. The approach entails the development of a novel combined model for sequence containers and iterators within STL, aimed at capturing states that conventional models fail to record. We establish a set of meta-operations to represent STL methods and track changes in the state of sequence containers and iterators. These models work in tandem to simulate various behaviors of sequence containers during program execution. Additionally, we define new checkers based on three bug patterns extracted from the C++ standard and famous coding regulations. These checkers check EASC bugs by examining the state of sequence containers and iterators.

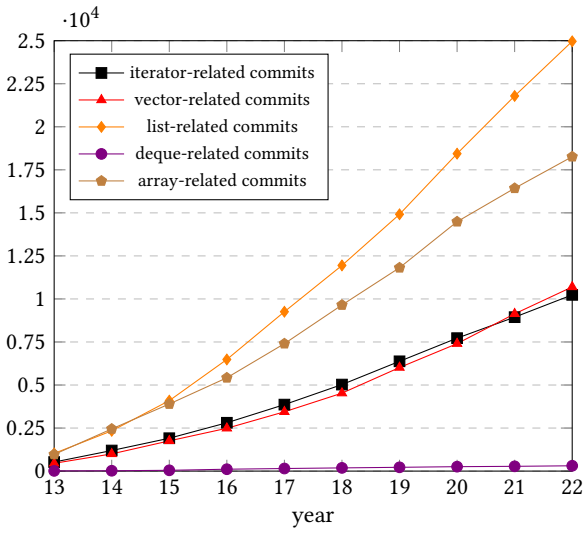

Figure 1: The number of EASC-related commits. Use *iterator*, *vector*, *list*, *deque*, and *array* as keywords to search commits from projects with more than 10,000 stars on GitHub.

We developed an analyzer named Scasa on top of the Clang Static Analyzer [1]. We evaluate Scasa's performance using both a handmade benchmark and eight open-source C++ projects. The experimental results demonstrate the effectiveness of the approach in detecting EASC bugs. The main contributions of our work are outlined as follows:

- We model the sequence containers and their iterators within STL, representing STL methods using a set of meta-operations. This combined model offers improved representations of the states of sequence containers and iterators (Section 3.2, Section 3.3).
- We develop an analyzer equipped with checkers specifically crafted to detect three identified bug patterns, enabling systematic identification of EASC bugs in C++ projects (Section 3.4, Section 4.2).
- Leveraging bug reports generated from our analysis of real-world projects, we offer feedback to developers and uncover previously undetected bugs (Section 4.5).

2 BACKGROUND

In this section, we will introduce the sequence containers in STL, element accessing bugs in C++ sequence containers and the motivating example.

2.1 Sequence Containers in STL

The C++ Standard Template Library (STL) comprises a collection of classes and methods that are integral to the core language. STL containers consist of sequence containers and associative containers, and since the properties and definitions of associative containers differ from those of sequence containers, we will only focus on sequence containers in the subsequent paper. Sequence containers denote a subset of class templates within the STL designed for storing objects. As templates, they offer flexibility to store arbitrary elements and facilitate direct access to them. Depending on the iterator types associated with the sequence containers and the storage format of the elements, sequence containers are further classified into various types: deques, lists, forward_lists, arrays, and vectors. For containers supporting random access, such as vectors, direct access via subscripts akin to arrays is feasible. Containers allowing only sequential access, like lists, enable access through iterators, resembling linked lists. Iterators and methods are also important part of sequence containers.

The iterator serves as a data type representing elements within a container. Within sequence containers, given their support for sequential access, iterators provide a convenient means to access elements throughout the container and traverse its contents. Based on the functionalities of iterator operations, we can categorize them into three main types in Figure 2. Firstly, assignment operations serve as initialization functions. Through these operations, values can be assigned to newly defined iterators using other iterators of

the same type or STL methods. This mechanism also facilitates overwriting the original content within the iterator. Secondly, movement operations enable traversal within the container. For bidirectional iterators present in lists, movements such as `--` and `++` facilitate backward and forward movement, respectively. For random access iterators typical in vectors, additional operations like `+n` and `-n` expedite iterator movement, aiding in a swift traversal of the container. Lastly, invalidation operations can render an iterator invalid when it is deleted or its address is modified. Accessing an invalidated iterator may lead to undefined behaviors. So, caution must be exercised to prevent such occurrences.

STL methods encompass a series of methods designed to streamline operations on sequence containers. Consequently, these methods have the potential to impact both the state of containers and the state of iterators. Regarding container state, STL methods can modify container size. For instance, the `clear` method can directly empty the container, `resize` can modify the container's size, and `insert` can expand the container's size by adding elements. Concerning the iterator state, STL methods can initialize iterators, which is mentioned in the iterator assignment operations discussed previously. For instance, `begin` can assign the iterator to the first element, while `find` can return an iterator within the container that meets specified criteria. Additionally, STL methods can manipulate iterator offsets, for instance, `std::advance` can move the iterator forward. Lastly, STL methods can invalidate iterators. `erase` invalidates the iterator by removing the corresponding element, while `insert` can invalidate iterators after the insertion point, as it may alter iterator addresses.

2.2 Element Accessing Bugs in Sequence Containers

Sequence containers are prone to errors when accessing elements. We call all these Element Accessing bugs in Sequence Containers EASC bugs. According to the documentation of the static analysis tool Cppcheck [2], three bug patterns of EASC bugs can be summarized, and identified, which have the potential to cause undefined behaviors, yet are not adequately detected by off-the-shelf

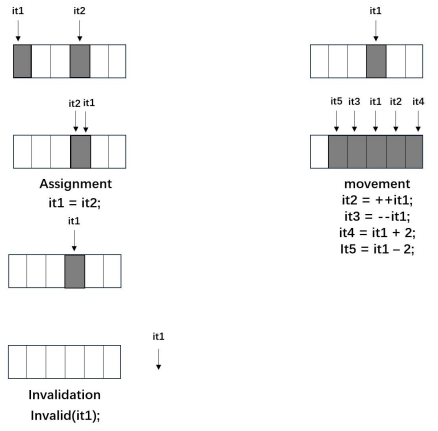

Figure 2: Iterator operations in sequence containers

```
1  int scoreOfParentheses(std::string S){
2    std::vector<int> stack(1, 0);
3    for (char& c : S) {
4      if (c == 'c') {
5        auto it = stack.begin();
6        it++;
7        it++;
8        // Iterator Out-of-Bound Access bug
9      } else if (c == 's'){
10       const auto last = stack.back();
11       stack.pop_back();
12       stack.back() += 1;
13       // Empty Container Access bug
14     } else{
15       auto it = stack.begin();
16       auto it1 = ++ it;
17       stack.erase(it);
18       *it1;
19       // Invalid Iterator Access bug
20     }
21   }
22   return stack.front();
23 }
```

Figure 3: The motivating example

tools. These patterns can be exemplified through simple illustrations provided in Figure 3.

Empty Container Access: Empty containers contain no elements, and any attempt to access elements within them will result in an undefined behavior. Hence, methods attempting to access elements in empty containers, such as `pop_back` and `back`, are reported as *Empty Container Access* bugs.

Iterator Out-of-Bound Access: Containers have bounds, and when using an iterator to access container elements, the iterator must remain within the container's bounds. If an iterator exceeds the container's scope, it may lead to an undefined behavior. Therefore, accessing addresses beyond the container's boundaries with an iterator is reported as an *Iterator Out-of-Bound Access* bug.

Invalid Iterator Access: Sequence containers offer bidirectional iterators for efficient sequential access, and some sequence containers, like vectors, even provide random access iterators. However, when containers undergo insertion or deletion operations, the validity of their iterators may be changed. For instance, after inserting an element, iterators after the insertion point in the vector become invalid. Access attempts on invalid iterators are reported as *Invalid Iterator Access* bugs.

2.3 Motivating Example

The code snippet depicted in Figure 3 comprises three solution code submissions from LeetCode [6]. There are several critical bugs.

Initially, a vector named `stack` is instantiated with a size of one at line 2. Subsequently, upon entering the loop at line 3, if the variable `c` equals char 'c', the code proceeds to the true branch. Here, an iterator is assigned to `stack.begin()` at line 5, followed by two increments at lines 6 and 7. The second increment surpasses the upper bound of the vector, leading to an *Iterator Out-of-bound Access* bug.

In the event that the variable `c` equals char 's' at line 9, invoking `pop_back` on `stack` at line 11 reduces its size to zero. Consequently, calling `back` at line 12 will result in an access bug due to the empty container, reported as an *Empty Container Access* bug.

If the variable `c` does not equal char 'c' or 's', the code may proceed to the else branch at line 15. Here, two iterators, `it` and `it1`, are defined at lines 15 and 16. Upon deletion of `it` at line 17, as the iterator in the vector is a free access iterator, `it1` becomes invalidated since it is behind `it`. Consequently, an access bug will manifest at line 18 due to iterator invalidation, indicating an *Invalid Iterator Access* bug.

While all three bugs result in undefined behaviors, off-the-shelf checkers fail to provide the expected reports. We attempted to analyze the example code using the latest versions of Clang Static Analyzer (CSA) [1], infer [5], as well as CppCheck [3]. However, none of these tools can detect any of the bugs. Based on the empirical research conducted by Kovacs et al. [17], static analysis tools do not excel in analyzing complex data structures like containers in STL, since STL data structures and method behaviors are difficult to model in static analysis leading to numerous false negatives.

3 MODELING SEQUENCE CONTAINERS AND ITERATORS AND CHECKING EASC BUGS

In this section, starting with the workflow, we will introduce the combined model for sequence containers and iterators, abstraction of methods, as well as checkers for the three bug patterns.

3.1 Workflow of Checking EASC bugs

To explain how EASC bugs are checked, we present the workflow of Scasa in Figure 4. The figure represents newly added models, methods, and checkers for detecting EASC bugs.

We systematically analyze each method in the input file according to the topological order of the call graph. This method enables us to traverse program paths and scrutinize relevant statements, focusing particularly on those associated with sequence containers and iterators. To model iterator operations (Section 3.3.1), we employ the iterator modeler, which translates iterator operations into state changes within the iterator models. Similarly, STL methods on sequence containers are abstracted into meta-operations using the Container Operations Modeler (Section 3.3.2). Subsequently, these meta-operations are translated into state changes within the container model (Section 3.2.1) and iterator model (Section 3.2.2). During model application, the states of iterators and containers models are scrutinized by the Checkers for EASC bugs (Section 3.4), and bug reports will be reported whenever EASC bugs are detected.

3.2 The Program State of Sequence Containers and Iterators

For detecting EASC bugs, we need the program states of the sequence containers and iterators, including the sequence container sizes, and the pointee and the validity of iterators. We maintain a set of program states of all the sequence containers and iterators in the program by the container modeler (Section 3.2.1) and the iterator modeler (Section 3.2.2).

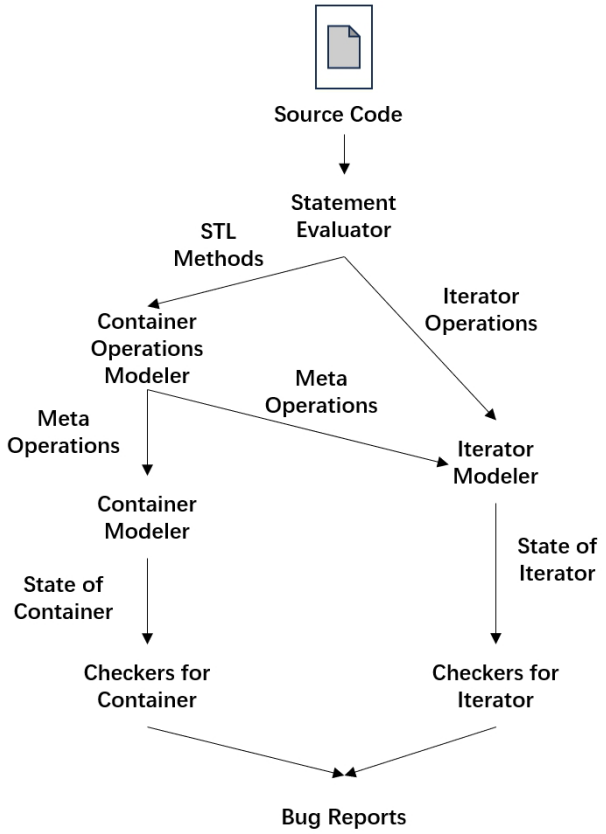

Figure 4: The workflow of modeling and checking EASC bugs

3.2.1 ***Modeling containers with size.*** For detecting EASC bugs, we need sequence container sizes and the positional relationship between iterators and the corresponding containers. However, the EASC bugs targeted in this article primarily involve accessing elements within the container, rather than the values of the elements. Therefore, our approach will not model the values of elements within the container.

Modeling sequence containers can be formally defined as a 3-tuple $ct = \langle id, size, type \rangle$, where

id represents the identifier of a container, serving as a unique symbol to distinguish it from other containers;

$size \in \mathbb{N} \cup SV$ denotes the size of the container. It can either be a non-negative integer or a Symbolic Value (SV). When the size of the container is known, *size* is represented by a non-negative integer. However, if the *size* is unknown, it is represented by a symbolic value;

$type \in \{vector, array, list, forward\_list, deque, other\}$ specifies the category of the container, which can be one of the predefined types such as *vector*, *array*, *list*, *forward_list*, *deque*, or *other* (containers not classified as sequence containers). Since our approach focuses solely on sequence containers, we use *type* to filter containers in STL. Upon creation of a container, we track its type by examining the Abstract Syntax Tree (AST) of the current program statement

and store it. Subsequently, during container operations, we check the type of the container. If it is a sequence container, we proceed with analyzing its operations, otherwise, we exclude the operations of the container.

During static analysis, when the size of a container is required, we will check whether the container exists within the current program's container set. If it does, the corresponding size of the container is retrieved directly. However, if the container is passed from an external source or is a global variable, a symbolic value is generated to represent its size. Combining the size and type information with a new identifier creates a 3-tuple for the container, which is then added to the program state set. Subsequently, the generated symbolic value is returned for further analysis.

In the motivating example, the program initializes an integer vector named *id*, containing a single element, 0. Consequently, the state of the container $\langle id, 1, vector\rangle$ is established, indicating that the sequence container *id* is a vector with a size of 1. Upon execution of the `pop_back` method at line 11, reducing the container's size by 1, the container's state is accordingly updated to $\langle id, 0, vector\rangle$. Similarly, when the `erase` method is invoked at line 17, also reducing the container's size by 1, the container's state is updated to $\langle id, 0, vector\rangle$. By utilizing the 3-tuple representation of the container, the checker can effectively examine its size and determine whether it is empty.

3.2.2 ***Modeling Iterators with Pointee and Validity.*** Among the bug patterns summarized, the state of an iterator focuses on two properties: the pointee of the iterator, including the position and the container of the iterator, and the validity of the iterator. Since the position of the iterator in the corresponding container can be represented by its offset from the beginning position of the container, the iterator model can be formally defined as a 4-tuple $iter = \langle id, ct, offset, valid\rangle$ where

id is the identifier of an iterator, *id* is a symbol used to distinguish one iterator from another;

$ct \in CT \cup \{unknown\}$ represents the container to which the iterator belongs and CT Represents the set of all container 3-tuples in the program. If the container is known, it corresponds to a specific container in the container set. If the container is unknown, its *ct* is *unknown*;

$offset \in \mathbb{N} \cup SV$ represents the position of the iterator, which is a non-negative integer or a symbolic value. It is a non-negative number when the position of the iterator is determined, and a symbolic value when the position of the iterator is uncertain;

$valid \in \{true, false\}$ is a Boolean value which represents the validity of the iterator.

For the convenience of operational semantic description in Section 3.3.1 and Section 3.3.2, if the iterator of a sequence container is now in the invalid range of an invalid operation, We call the iterator Invalid Sequence Container Iterator (InSCIt).

3.3 The State Transition of Sequence Containers and Iterators

Statements in the program may result in the state transition of sequence containers or iterators. We need to recognize these statements to analyze the state transition and refresh the program state. So we will introduce iterator operation modeler (Section 3.3.1) and container operation modeler (Section 3.3.2) in the section.

3.3.1 ***Iterator Operations Modeler to Record the Offset and Validity Changes.*** Iterator operations can affect the state of the iterators. So we can define three kinds of iterator operations. Their operational semantics are presented in Table 1.

For the `Assign` operation, three scenarios can occur. Firstly, if the operation involves assigning through another iterator, the state of the target iterator mirrors that of the source iterator. Consequently, we replicate the state of the source iterator onto the target iterator. Secondly, if the operation entails assignment through an STL method and the offset of the returned iterator from the method is known, such as `begin` and `end`. In such cases, we can duplicate the state of the returned iterator onto the target iterator. Lastly, if the operation involves assignment through an STL method but the offset of the returned iterator from the method is unknown, like `find` and `find_if`. For such STL methods, we assume that the offset of the returned iterator is 0, representing one of the potential return values.

For the movement operation of iterators, it can be categorized into two analogous situations based on the direction of movement. If the iterator moves forward, the `increase` operation is invoked, otherwise, the `decrease` operation is invoked. Given that the container to which the iterator belongs is already known, we can utilize the offset to denote the position of the iterator. Consequently, when the movement operation transpires, the *offset* of the iterator will also undergo modification.

For the `invalid` operation, we first locate the container to which the iterators that require invalidation belong based on *ct*. Within the same container, we utilize the offset instead of the address. Subsequently, if the offset of the iterator falls within the invalid range, the iterator is InSCIt and we will invalidate it.

There is an exception, when *ct* of an iterator is *unknown*, we will assume the first use of the iterator is always correct. When the iterator first comes into contact with an iterator of a known container or directly with a container, such as a method call or an iterator assignment, we assign that container to the iterator's *ct*.

In the motivating example, at line 5, the program initializes an iterator *id* for the vector `stack`. The state of the iterator is $\langle id, \texttt{stack}, 0, true\rangle$, indicating that *id* is a valid iterator for the vector `stack`, with an offset of 0. Through two increment operations at lines 6 and 7, the state of the iterator changes to $\langle id, \texttt{stack}, 2, true\rangle$. At lines 15 and 16, two iterators are created. The state of iterator `it` is $\langle id1, \texttt{stack}, 0, true\rangle$, while the state of iterator `it1` is $\langle id2, \texttt{stack}, 1, true\rangle$. Then, at line 17, as `it` is deleted, both `it` and `it1` are rendered invalid because the iterators in the vector are free to access iterators. Consequently, the state of `it` becomes $\langle id1, \texttt{stack}, 0, false\rangle$, and the state of `it1` becomes $\langle id2, \texttt{stack}, 1, false\rangle$. The iterator model effectively captures the iterator's pointee and validity, facilitating the checker's assessment of whether the iterator is out of bounds or invalid.

3.3.2 ***Modeling Container Methods with meta-operations.*** Because we mainly focus on container size, iterator validity, and iterator offset changes. Many STL methods exhibit similar behavior. Therefore, we abstract STL methods as six types of meta-operations in Table 3, and their operational semantics are presented in Table 2.

`Construct`. Corresponds to the constructor of the STL sequence container. Upon container creation, we update the container state by obtaining its size from the constructor and modifying the *size* attribute accordingly.

`Resize` Encompasses all STL methods that directly modify the container size, including `resize` and `clear`. Here, *size* refers to the count of elements within the container. Therefore, methods like reserve() are not modified with `Resize` operations, but resize() are modified with. Different methods or different container types may lead to different InSCIts, for example, the InSCIts of the method clear of vector are all iterators of the container. Consequently, we invalidate all container iterators and update the *size* attribute accordingly.

`Add`. Encompasses methods that augment the size of the STL sequence container, such as `emplace` and `insert`. We track the size change induced by these methods, which can be either 1, the difference in offsets of the iterators, or the size of a container. Additionally, we identify the InSCIts resulting from these operations. For instance, in a vector, the InSCIts after an insertion include all iterators after the insertion position, whereas, in a list, there is no iterator to be invalid. Subsequently, we invoke the iterator operation `Invalid` to invalidate InSCIts and update the container's *size* attribute accordingly.

`Sub`. Analogous to `Add`, `Sub` encompasses methods that reduce the size of the STL sequence container, such as `erase` and `pop_back`. We ascertain the size change and InSCIts resulting from these methods and adjust the state of both iterators and the container accordingly.

`Swap`. Pertains to STL methods that interchange containers. Given that swap methods impact both the parameter and the caller, they are treated as a distinct meta-operation. We retrieve the sizes of both containers and exchange their respective *size* attributes and InSCIts are also invalid by `Invalid` operation.

`Other`. Represents STL methods that do not affect the container size, iterator offset, or validity. Although these methods serve different purposes, they are not considered in the model. Some methods that return an iterator are addressed within the iterator model. For example, `begin` returns the head iterator of the container, generating an iterator with an offset of 0, used as the return value of `begin`. However, for methods like `find`, where information about the container elements is unavailable, determining the iterator's offset is impractical. To mitigate false positives, we select a possible return value, the head iterator, for such methods.

In the motivating example, at line 10, the program executes the method `pop_back`, resulting in a change in the container's size. This action corresponds to the meta-operation *Sub*, with a size decrement of 1 and an invalid range encompassing the last element in the container and any subsequent iterators. Since there are no iterators within this invalid range, no iterators will be invalidated. Then, the *size* attribute of the container state transitions from 1 to 0. At line 15, the program utilizes the method `erase`, which also impacts the container's size. Here, the meta-operation *Sub* is identified, with a size change of 1 and an invalid range is `it` and subsequent iterators. Both `it` and `it1` fall within this invalid range, the *valid* attributes of both `it1` and `it` will transition from *true* to *false*. Additionally, the *size* attribute in the container state will transition from 1 to 0.

3.4 Checking EASC Bugs

The checkers are used to detect the three bug patterns and Algorithm 1 shows the process.

Empty Container Access. To identify *Empty Container Access* bugs, we examine access methods requiring a non-empty container, such as `back`, or `Sub` meta-operations. If the container size is 0 when invoking such methods, or if it is less than the size change corresponding to the *Sub* operation, we report an *Empty Container Access* bug. In the motivating example, at line 11, where the container's state is$\langle id, 0, vector\rangle$, the program invokes the method `back`. The *Empty Container Access* checker observes that the container's size is 0, triggering the reporting of an *Empty Container Access* bug.

Iterator Out-of-Bound Access. Detecting *Iterator Out-of-Bound Access* bugs involves checking iterator movements (`Increase` or `Decrease` operations). For an `Increase` operation, we check if the new iterator offset exceeds the container size. Conversely, for a `Decrease` operation, we confirm if the iterator offset minus the variable is less than 0. In both cases, we report an *Iterator Out-of-Bound* Access bug. In the motivating example, at line 6, the iterator's state is represented as a 4-tuple, $\langle id, \texttt{stack}, 1, true\rangle$, while the container's state is a 3-tuple $\langle id, 1, vector\rangle$. At line 7, the *Iterator Out-of-Bound Access* checker detects that the offset of the iterator `it`, plus the variable 1, exceeds the size of the container, leading to an *Iterator Out-of-Bound Access* bug.

Table 1: operational semantics of Iterator Operations. (sc: sequence container, τ: *offset*, ϕ: *valid*.)

Type	Expression	Operational Semantics
`Assign`	`Assign`(it_1, it_2)	$\frac{it_1=\langle id,ct,\tau,\phi\rangle}{it_1=\langle id,it_2.ct,it_2.\tau,it_2.\phi\rangle}$
`Increase`	`Increase`(it,n)	$\frac{it=\langle id,ct,\tau,\phi\rangle}{it=\langle id,ct,\tau+n,\phi\rangle}$
`Decrease`	`Decrease`(it,n)	$\frac{it=\langle id,ct,\tau,\phi\rangle}{it=\langle id,ct,\tau-n,\phi\rangle}$
`Invalid`	`Invalid`(InSCIt)	$\frac{InSCIt=\langle id,ct,\tau,\phi\rangle}{InSCIt=\langle id,ct,\tau,false\rangle}$
`Bind`	`Bind`(it,sc)	$\frac{it=\langle id,unknown,\tau,valid\rangle}{it=\langle id,sc,\tau,\phi\rangle}$

Table 2: operational semantics of meta operations. (Δ: size change, sc: sequence container, θ: *size*, τ: *type*.)

Type	Expression	Operational Semantics
`Construct`	`Construct`(sc,Δ)	$\frac{sc=\langle id,\theta,\tau\rangle}{sc=\langle id,\Delta,\tau\rangle}$
`Resize`	`Resize`(sc,Δ,InSCIt)	$\frac{sc=\langle id,\theta,\tau\rangle}{sc=\langle id,\Delta,\tau\rangle;\ \texttt{Invalid}(InSCIt)}$
`Add`	`Add`(sc,Δ,InSCIt)	$\frac{sc=\langle id,\theta,\tau\rangle}{\texttt{Resize}(sc,\theta+\Delta,InSCIt)}$
`Sub`	`Sub`(sc,Δ,InSCIt)	$\frac{sc=\langle id,\theta,\tau\rangle}{\texttt{Resize}(sc,\theta-\Delta,InSCIt)}$
`Swap`	`Swap`(sc_1, sc_2,InSCIt)	$\frac{sc_1=\langle id_1,\theta_1,\tau\rangle;\ sc_2=\langle id_2,\theta_2,\tau\rangle}{sc_1=\langle id,\theta_2,\tau\rangle;\ sc_2=\langle id,\theta_1,\tau\rangle;\ \texttt{Invalid}(InSCIt)}$

Table 3: Meta-Operations

Meta-Operation	description
Construct	Construct the container
Resize	Resize container size
Add	Increase container size
Sub	Decrease container size
Swap	Swap container sizes
Other	Independent of container size

Invalid Iterator Access. To catch *Invalid Iterator Access* bugs, we inspect iterator validity during access. If the *validity* of the iterator is *false* when accessed, we report an Invalid Iterator Access bug. In the motivating example, at line 16, as the program accesses `it1`, the *Invalid Iterator Access* checker scrutinizes the state of `it1`. Because the *validity* of `it1` is *false*, an *Invalid Iterator Access* bug is promptly reported.

4 EVALUATION

To evaluate the effectiveness of our model and the ability to reveal EASC bugs, we carry out three groups of experiments to answer the following three research questions.

- **RQ1**: The coverage of AST nodes. Can Scasa more completely cover and analyze the AST nodes of the sequence container transition method in the projects?
- **RQ2**: Accuracy of Scasa on handmade benchmark. Can Scasa detect EASC bugs caused by different operations on handmade benchmark?
- **RQ3**: Effectiveness of Scasa on open-source projects. How many real-world EASC bugs can be detected by Scasa?

The first research question evaluates the analysis ability of *Scasa*. We count the number of AST nodes of STL methods that change the state of sequence containers and iterators analyzed by Scasa in eight open-source C++ projects and compare it with Clang Static Analyser (CSA). The comparison is presented in Section 4.3. We also use a handmade benchmark and eight open-source C++ projects to present the effectiveness of *Scasa* and compare the ability to detect EASC bugs with the off-the-shelf open-source static analysis tools. The comparison is presented in Section 4.4 and Section 4.5.

Algorithm 1 Checking EASC Bugs. (Path is an execution path composed of AST nodes obtained through static analysis)

```
for all n in Path do
  if Equal(n,Sub) then
    ct = GetContainer(n);
    change = GetParam(n);
    if ct.size − change < 0 then
      BugReport(Empty Container Access Bug);
  if Equal(n,Increase) then
    it = GetIterator(n);
    ct = it.ct;
    change = GetParam(n);
    if it.offset + change > ct.size then
      BugReport(Iterator Out-of-Bound Access Bug);
  if Equal(n,Decrease) then
    it = GetIterator(n);
    change = GetParam(n);
    if it.offset - change < 0 then
      BugReport(Iterator Out-of-Bound Access Bug);
  if IsAccessOperation(n) then
    it = GetIterator(n);
    ct = it.ct;
    if ct.size == 0 then
      BugReport(Empty Container Access Bug);
    if !it.valid then
      BugReport(Invalid Iterator Access Bug);
    if it.offset < 0 || it.offset >= ct.size then
      BugReport(Iterator Out-of-Bound Access Bug);
```

4.1 Setup of Experiments

1) Environment and Tools: We conduct all experiments on a Linux server equipped with two Intel® Xeon® E5-2680 v4 CPUs with a total of 56 threads and 256 GB of memory. The evaluation compares Scasa against the three off-the-shelf latest versions of tools: Clang Static Analyzer (CSA) version 18, CppCheck version 2.13.0 [3] and infer version 1.1.0 [4].

2) Benchmark: The benchmark comprises two components: handmade snippets (benchmark hm) and open-source projects (benchmark osp). Benchmark hm in Table 4 consists of 2230 handmade code snippets written in C++11 standard and designed to assess

the model's capability in representing sequence containers and detecting inserted EACS bugs. Due to the absence of relevant test code snippets, We enumerate all methods in sequence containers that involve container state changes, make a wrong case for each of the methods, and combine them with 10 control flows in the Juliet Test Suite [9]. The snippets are categorized into three types based on the types of errors: 690 snippets for Invalid Iterator Access bugs, 1100 for Empty Container Access bugs, and 440 for Iterator Out-of-Bound Access bugs.

Benchmark osp comprises eight C++ open-source projects sourced from GitHub, as depicted in Table 5. These projects are written in C++11 or later versions of the C++ standard and boast widespread usage and substantial scale on GitHub. The corresponding number of stars and lines of C++ code are provided in the second and third columns, respectively. For each project, we utilize the latest available version during the experiments. The fourth column shows the extensive usage of sequence containers in these projects' development processes, increasing the likelihood of encountering EASC bugs. Unlike the handmade benchmark, these real-world projects feature extensive codebases with intricate functionalities. Therefore, we select these projects for analysis.

Table 4: Statistics of benchmark hm. (ECA: *Empty Container Access* bug, IOA: *Iterator Out-of-Bound Acess* bug, IIA: *Invalid Iterator Access* bug)

Type	ECA	IOA	IIA
vector	160	230	160
list	180	310	50
forward_list	170	280	40
deque	170	260	190
array	10	20	0
Total	690	1100	440

Table 5: Information of benchmark osp. Tol nodes are all AST nodes that contain the methods that can change the state of sequence container.

Project	Commit	Stars	KLoc	Tol nodes
Aria2	`8cb271b`	33.5k	93.25	547
Assimp	`7774f18`	10.2k	212.84	2,258
Barony	`0110f8d`	466	369.29	1,340
Bitcoin	`05b6a17`	75.6k	168.98	2,069
Cataclysm-DDA	`03175a6`	9.7k	467.27	5,134
Jsoncons	`c642f38`	666	38.91	1,384
Leveldb	`068d5ee`	35k	65.76	352
Rgbds	`8066d46`	1.3k	20.36	150

4.2 Implementation of Scasa

Following the methodology outlined in the preceding section, we apply our model within the Clang Static Analyzer (CSA) [1] of clang-15. However, the reports generated by the Scasa not only encompass the anticipated EASC bug reports but also include three additional types of reports: Non-EASC Bug Reports, Non-STL EASC Bug Reports, and EASC bug reports with assertions in the path. Since the first two types of bug reports are irrelevant to Scasa's objectives and paths containing assertions are unable to be processed at the moment, we implemented measures during static analysis to mitigate their inclusion. Specifically, when reporting bugs, we scrutinize the Abstract Syntax Tree (AST) of the sequence container. If the type of the container does not belong to the namespace std, the bug is not reported. Additionally, upon completing the static analysis, we conduct a secondary screening of the reports. By categorizing the bug types and analyzing the execution paths, we can filter out the aforementioned types of bug reports. These strategies ensure that the final reports exclude the aforementioned three types of bug reports.

4.3 Evaluation on the coverage of AST Nodes Between CSA and Scasa

Table 6: The numbers of covered and analyzed AST nodes of sequence container state transition methods. (Total: all AST nodes that contain the methods that can change the sequence container state, Covered: AST nodes that are covered by static analysis tools, Analyzed: AST nodes where the methods are modeled and the container state of static analysis tools is changed.)

Project	Total	Scasa		CSA	
		Covered	Analyzed	Covered	Analyzed
Aria2	547	487	487	524	480
Assimp	2,258	1,651	1,650	1,266	1,230
Barony	1,340	930	930	830	718
Bitcoin	2,069	1,568	1,567	1,512	1,441
Cataclysm-DDA	5,134	3,734	3,733	3,513	3,265
Jsoncons	1,384	896	896	836	802
Leveldb	352	142	142	149	148
Rgbds	150	109	109	106	100
Total	13,234	9,530	9,527	8,745	8,193

As shown in Table 6, Scasa can cover more AST nodes of sequence container state transition methods most time, and based on this, Scasa can also analyze more AST nodes and reflect the state transition methods in the container and iterator models. This improvement can be attributed to two main factors.

First, the container operation modeler is more comprehensive and can simulate a broader range of STL methods compared to CSA. For example, `insert` is overloaded into four types in actual use. However, during the analysis process of CSA, due to the lack of size in the container model, only two of the four types of `insert` statements in the program can be recognized. In contrast, With the help of the meta operation and the container model, the four types of `insert` statements can be classified into one type of meta operation so that they can be processed uniformly, and more `insert` statements in the program can be identified. Second, the combined model of the container and the iterator better track the state of containers and iterators through the iterator operation modeler and container operation modeler. Consequently, Scasa handles sequence container and iterator state transitions more effectively, avoiding many situations where CSA might skip analysis due to incomplete

container or iterator states. As a result, Scasa offers a more precise description of sequence container behavior within projects, leading to improved bug detection.

Anwser to RQ1: Scasa covers 5% more AST nodes and analyzes 10% more AST nodes than CSA, which shows that Scasa has better coverage and analysis capabilities for the state transition of sequence containers.

4.4 Evaluation on Accuracy of Scasa on Benchmark hm

We use static analysis tools that support the C++11 standard, CSA, Cppcheck, and infer to perform testing together with Scasa. We will evaluate the effectiveness of the tools based on precision and recall.

According to Table 7, Scasa demonstrates superior performance than CSA, Cppcheck, and infer, infer even can not detect sequence container bugs. Furthermore, it is evident that off-the-shelf tools employ conservative detection strategies for EASC bugs, resulting in significantly higher precision than recall. This conservative approach can lead to a substantial number of false negatives in real-world projects. However, our model, owing to its precise analysis of container and iterator states, can more accurately scrutinize sequence container behavior, thereby mitigating false negatives. Consequently, Scasa exhibits a significantly higher recall rate compared to other tools in the experiment.

Anwser to RQ2: Scasa performs much better than other state-of-the-art static analysis tools on the benchmark hm, which shows that Scasa can detect bugs caused by a variety of STL methods.

Table 7: The analysis results on the benchmark hm. We assume the correct bug report to be true positive. The reason why the total number of reports of Scasa is more than others is that there are 30 snippets where Scasa reports a bug, but the bug location is wrong, those snippets are both FP and FN.

Tool	Scasa	CSA	cppcheck	infer
TP	2080	320	1210	0
TN	0	0	0	0
FP	30	0	0	0
FN	150	1910	1020	2030
Precision	98.57%	100.00%	100.00%	N/A
Recall	93.27%	14.34%	54.26%	N/A
F1	95.84%	25.08%	70.34%	N/A

4.5 Evaluation on Effectiveness of Scasa on benchmark osp

Since the static analysis report includes the error path and the constraints that the path needs to satisfy, to evaluate false positives and accuracy, we meticulously scrutinize all reports from the three static analysis tools and invited two of the authors who have experiences of C++ development to check the feasibility of these reports and make joint judgments. Paths deemed feasible were considered correct reports and the outcomes are tabulated in Table 8.

The table reveals that Scasa not only generates a greater number of bug reports compared to off-the-shelf tools but also exhibits significantly higher precision. To further validate the accuracy of the reports, We select 15 typical bug reports from 72 reports to the original developers for feedback. And up to now, there are 4 reports verified, and 1 report fixed before we report.

Moreover, we conduct an analysis of the occurrence of the three different bug patterns in the correct bug reports identified by Scasa. The findings are presented in Table 9.

The prevalence of *Iterator Out-of-Bound Access* bugs is conspicuous. In contrast to *Empty Container Access* bugs and *Iterator Invalid Access* bugs, where bugs stem solely from container or iterator operations respectively, *Iterator Out-of-Bound Access* bugs implicate both the iterator's pointee and the container's size. These two factors are interrelated, making it easier for developers to overlook their constraints during development, thereby leading to out-of-bounds scenarios. Hence, the abundance of *Iterator Out-of-Bound Access* bugs is reflective of real-world circumstances.

Table 8: Evaluation on benchmark osp.

Project	Scasa	CSA	Cppcheck
Aria2	2/4	0/1	0/0
Assimp	17/27	6/10	0/0
Barony	3/10	0/3	0/0
Bitcoin	4/11	0/3	0/0
Cataclysm-DDA	44/70	1/30	1/4
Jsoncons	0/0	0/0	0/0
Leveldb	1/2	0/3	0/0
Rgbds	1/1	0/1	0/0
Total	72/125	7/51	1/4

Table 9: The number of bug patterns in the correct reports.

Project	IIA	ECA	IOA
Aria2	0	0	2
Assimp	0	0	17
Barony	0	0	3
Bitcoin	0	0	4
Cataclysm-DDA	0	5	39
Jsoncons	0	0	0
Leveldb	0	0	1
Rgbds	0	0	1
Total	0	5	67

Table 10: Time and memory cost on benchmark osp.

Project	CSA		Scasa	
	Time (Sec.)	Memory (MB)	Time (Sec.)	Memory (MB)
Aria2	172.21	380.07	180.51 (4.47%+)	403.37
Assimp	412.38	504.40	613.08 (48.66%+)	487.11
Barony	261.68	790.89	405.78 (55.06%+)	777.14
Bitcoin	832.99	856.30	965.31 (15.88%+)	771.94
Cataclysm-DDA	1,928.24	1,595.78	2,823.93 (46.45%+)	1,477.76
Jsoncons	509.08	903.02	806.4 (58.40%+)	774.94
Leveldb	556.82	385.75	999.59 (79.51%+)	381.68
Rgbds	36.51	421.64	66.96 (83.40%+)	394.67

Finally, as Scasa is developed based on CSA, we also assess the changes in time and memory costs compared to CSA. By averaging three executions, we collect data on these metrics, as summarized in Table 10. The results reveal that Scasa exhibits an increased run-time compared to CSA because more statements need to be analyzed, and more operations need to be checked. This rise in execution time is expected, especially given our focus on projects with numerous STL sequence container operations, necessitating extensive sequence container analysis in Scasa. However, Scasa consumes less memory than CSA. This can be attributed to Scasa's ability to detect EASC bugs, leading to the termination of certain paths earlier compared to CSA, thereby reducing memory usage.

Anwser to RQ3: Scasa generates 125 reports for benchmark osp (with a time loss of 5–85%) where 72 reports are marked as correct and 5 reports are verified in 15 typical reports we select for developers, which shows that Scasa can detect real-world EASC bugs that state-of-the-art static analysis tools cannot detect.

5 THREAT TO VALIDITY

The main threats to the validity of the results lie in the following three aspects.

•*Non-sequential container*: In the STL, besides sequence containers, there are also non-sequential containers like map and set. These containers differ in properties, definitions, and functionalities from sequence containers, making their integration into our current model impractical. Additionally, commonly used non-STL containers such as string cannot be accurately recognized and processed in the model either. Consequently, if these non-sequential containers affect the state of sequence containers, the model may fail to generate accurate bug reports. Developing a new model for these Non-sequential containers is a future focus for us.

•*Benchmark Selection*: The handmade benchmark prioritizes functional aspects over program logic structures, which explains the considerable disparity in results between CSA and Cppcheck in the handmade benchmark versus real-world projects. Consequently, Scasa's performance on real-world projects may not match that of the handmade benchmark. Moreover, the selected projects may not encompass all language features, potentially leading to false negatives or positives when analyzing more complex projects.

•*CSA Analysis Engine*: CSA, being a static analysis tool, is not without its limitations, as discussed in Section 4.2. These include challenges with assertion handling, loss of global and class variable information, and method skipping for efficiency reasons, all of which can impact result accuracy and precision. Future versions of CSA are expected to incorporate numerous improvements and bug fixes, potentially altering the outcomes of Scasa.

•*Versions of C++*: In this paper, we focus on the STL methods under the C++11 standard. However, subsequent versions of C++ have introduced many new STL methods, such as list::erase_if in C++20 and vector::insert_range in C++23. Current model is unable to recognize these newly introduced methods, leading to inaccurate bug reports. Nevertheless, the impact of these new methods on the state of containers and iterators still essentially falls within one of the six meta-operations. Therefore, these new methods can be similarly abstracted as meta-operations, enabling the model to adapt to newer versions of C++.

6 RELATED WORK

Our work is mainly related to modeling and checking sequence container usage and memory-related bugs. In this section, we will mainly present the existing research in these aspects.

•*Analysis on sequence container usage.* Analysis of sequence container usage can be approached through both static and dynamic analysis techniques. In static analysis, D'evai et al. [10] utilized formal verification methods to analyze STL containers by formally specifying its methods. Pataki et al. [23] proposed employing iterator positions to represent container sizes and verify their validity. Additionally, Pataki et al. [25] and Horvath et al. [15] argued for an extension of the STL's iterator traits in order to emit warnings. They focus primarily on the intrinsic properties of STL containers or iterators, while overlooking the interactions between STL containers and iterators. This limitation hinders their ability to describe the state changes of STL containers and iterators. Scasa overcomes this limitation by utilizing a composite model. In dynamic analysis, Pataki et al. [26] introduced a novel iterator adaptor type and tag, along with safe containers capable of tracking iterator validity at runtime. This approach enables programmers to verify the preconditions of STL algorithms dynamically. Furthermore, Pataki et al. [24] proposed an extension to the iterator framework that performs runtime checks on iterator validity. Scasa as a static analysis tool can be performed at earlier stages of program development. This allows for the timely detection and reporting of potential issues, thereby reducing the cost of fixing them.

•*Analysis of memory-related bugs.* Memory-related bugs encompass a wide range of issues, some of which closely resemble bugs related to element access in sequence containers. Various studies have addressed these bugs using diverse approaches: Balakrishnan et al. [8] introduced memory models for C++ programs that are aware of the heap. They employed standard verification techniques and model checking to verify the properties of these models, thereby identifying potential bugs. Dillig et al. [11] developed a sound, precise, and fully automatic technique for static reasoning about container contents. This method allows tools to perform detailed program analysis based on an advanced understanding of container contents. Xu et al. [31] introduced Melton, a prototype tool that utilizes path-sensitive symbolic execution to track memory actions across different program paths governed by path conditions. SMOKE [12] employs a staged approach to balance precision and scalability in analysis. This method addresses the paradox where highly precise analysis limits scalability and imprecise analysis harms precision or recall. PCA [19] performs inter-procedural points-to and data-flow analyses, supporting the detection of memory leaks. Trabish et al. [29] proposed a bounded symbolic-size model that allows object sizes to have a range of values limited by a user-specified bound. This model improves performance and coverage in symbolic execution, leading to the discovery of previously unknown bugs. Ma et al. [22] proposed a static approach to detecting memory-related bugs by tracking heap memory management within smart pointers. In contrast, Scasa employs a meta-operation to abstract STL methods, which allows Scasa to more accurately track the state change of sequence containers and successfully detect potential memory errors in sequence containers.

•*Analysis on pointer.* Iterators, despite not being pointers, exhibit behavior akin to pointers, enabling the application of pointer analysis techniques to iterator analysis. Krainz et al. [18] introduced an incremental pointer analysis method utilizing diff graphs to depict how a function accesses and alters memory. Li et al. [20] proposed the Scaler framework, which efficiently estimates the amount of points-to information required to analyze each function with various context sensitivity variants. Yan et al. [32] presented a pointer-analysis-based static analysis for detecting Use-After-Free (UAF) bugs in multi-MLOC C source code, offering efficiency and effectiveness. Trabish et al. [30] proposed a novel integration of pointer analysis with dynamic analysis, including dynamic symbolic execution, via past-sensitive pointer analysis. Fegade et al. [13] employed semantic models to simplify complex pointer implementations, resulting in increased precision without significant performance overhead. Liu et al. [21] developed an incremental context-sensitive pointer analysis algorithm capable of scaling to large, complex Java programs, and amenable to efficient parallelization. Tan et al. [28] proposed selective context-sensitivity approaches to scale context-sensitive pointer analysis for large and complex Java programs. In contrast, Scasa not only analyzes the similar part of iterator with pointers but also considers the impact of STL methods on iterators, as well as how container types influence iterator validity. These allow Scasa to more accurately track the state of iterators.

7 CONCLUSION AND FUTURE WORK

In this paper, we introduced a combined model of sequence containers and iterators, along with a set of meta-operations derived from STL methods abstraction to manipulate container and iterator states. Our focus was on detecting Element Accessing bugs in Sequence Containers (EASC bugs), for which we identified three distinct bug patterns. We then implemented checkers of these bug patterns and evaluated our model and checkers using both a handmade benchmark and eight open-source C++ projects. The experimental results demonstrate that the approach can accurately detect EASC bugs.

In the future, we envision several avenues for expanding this work. As the first tool specifically designed for precisely identifying EASC bugs, we focused on three bug patterns. However, as the C++ standard evolves, additional bug patterns related to sequence containers may emerge, warranting further investigation and inclusion in the analysis. Then, while the study primarily addressed sequence containers within the C++ Standard Template Library (STL), there is scope for extending the analysis to other STL containers as well as custom containers found in large libraries, provided that precise detection mechanisms are developed for these container types.

ACKNOWLEDGMENTS

To Robert, for the bagels and explaining CMYK and color spaces.

REFERENCES

[1] [n. d.]. Clang Static Analyzer — clang-analyzer.llvm.org. https://clang-analyzer.llvm.org.
[2] [n. d.]. Cppcheck - A tool for static C/C++ code analysis — cppcheck.sourceforge.io. https://cppcheck.sourceforge.io/.
[3] [n. d.]. GitHub - danmar/cppcheck: static analysis of C/C++ code — github.com. https://github.com/danmar/cppcheck.
[4] [n. d.]. GitHub - facebook/infer: A static analyzer for Java, C, C++, and Objective-C — github.com. https://github.com/facebook/infer.
[5] [n. d.]. Infer Static Analyzer | Infer | Infer — fbinfer.com. https://fbinfer.com.
[6] [n. d.]. LeetCode. https://leetcode.com/
[7] Bence Babati and Norbert Pataki. 2019. Static analysis of functors' mathematical properties in C++ source code. In *AIP Conference Proceedings*, Vol. 2116. AIP Publishing.
[8] Gogul Balakrishnan, Naoto Maeda, Sriram Sankaranarayanan, Franjo Ivančić, Aarti Gupta, and Rakesh Pothengil. 2011. Modeling and analyzing the interaction of C and C++ strings. In *International Conference on Formal Verification of Object-Oriented Software*. Springer, 67–85.
[9] Paul E Black and Paul E Black. 2018. *Juliet 1.3 test suite: Changes from 1.2.* US Department of Commerce, National Institute of Standards and Technology ....
[10] Gergely Dévai and Norbert Pataki. 2009. A tool for formally specifying the C++ Standard Template Library. In *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, Vol. 31. 147–166.
[11] Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Precise reasoning for programs using containers. *ACM SIGPLAN Notices* 46, 1 (2011), 187–200.
[12] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 72–82.
[13] Pratik Fegade and Christian Wimmer. 2020. Scalable pointer analysis of data structures using semantic models. In *Proceedings of the 29th International Conference on Compiler Construction*. 39–50.
[14] Gábor Horváth and Norbert Pataki. 2015. Clang matchers for verified usage of the C++ Standard Template Library. In *Annales Mathematicae et Informaticae*, Vol. 44. 99–109.
[15] Gábor Horváth, Attila Páter-Részeg, and Norbert Pataki. 2017. Detecting Misusages of the C++ Standard Template Library. In *Proceedings of the 10th International Conference on Applied Informatics (ICAI 2017)*. 129–136.
[16] Nicolai M Josuttis. 2012. *The C++ standard library: a tutorial and reference*. Addison-Wesley.
[17] Réka Kovács, Gábor Horváth, and Zoltán Porkoláb. 2019. Detecting C++ lifetime errors with symbolic execution. In *Proceedings of the 9th Balkan Conference on Informatics*. 1–6.
[18] Jakob Krainz and Michael Philippsen. 2017. Diff graphs for a fast incremental pointer analysis. In *Proceedings of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. 1–10.
[19] Wen Li, Haipeng Cai, Yulei Sui, and David Manz. 2020. PCA: memory leak detection using partial call-path analysis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1621–1625.
[20] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 129–140.
[21] Bozhen Liu and Jeff Huang. 2022. SHARP: fast incremental context-sensitive pointer analysis for Java. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–28.
[22] Xutong Ma, Jiwei Yan, Wei Wang, Jun Yan, Jian Zhang, and Zongyan Qiu. 2021. Detecting memory-related bugs by tracking heap memory management of C++ smart pointers. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 880–891.
[23] Norbert Pataki. 2010. C++ Standard Template Library by Ranges. In *Proc. of the 8th International Conference on Applied Informatics (ICAI 2010) Vol*, Vol. 2. 367–374.
[24] Norbert Pataki. 2012. Safe iterator framework for the C++ standard template library. *Acta Electrotechnica et Informatica* 12, 1 (2012), 17.
[25] Norbert Pataki and Zoltán Porkoláb. 2011. Extension of iterator traits in the C++ Standard Template Library. In *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 911–914.
[26] Norbert Pataki, Zalán Szűgyi, and Gergely Dévai. 2011. Measuring the overhead of c++ standard template library safe variants. *Electronic Notes in Theoretical Computer Science* 264, 5 (2011), 71–83.
[27] Lutz Prechelt. 1999. Technical opinion: comparing Java vs. C/C++ efficiency differences to interpersonal differences. *Commun. ACM* 42, 10 (1999), 109–112.
[28] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.
[29] David Trabish, Shachar Itzhaky, and Noam Rinetzky. 2021. A bounded symbolic-size model for symbolic execution. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1190–1201.
[30] David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. 2020. Past-sensitive pointer analysis for symbolic execution. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on*

the Foundations of Software Engineering. 197–208.

[31] Zhenbo Xu, Jian Zhang, and Zhongxing Xu. 2015. Melton: a practical and precise memory leak detection tool for C programs. *Frontiers of Computer Science* 9 (2015), 34–54.

[32] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*. 327–337.