Detecting Memory-Related Bugs by Tracking Heap Memory Management of C++ Smart Pointers

Xutong Ma[1,3], Jiwei Yan[2,3], Wei Wang[1,3], Jun Yan[1,2,3,§] Jian Zhang[1,3,§] and Zongyan Qiu[4]
[1]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[2]Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
[3]University of Chinese Academy of Sciences
[4]School of Mathematical Sciences, Peking University
Emails: {maxt, yanjw, wangwei19, yanjun, zj}@ios.ac.cn, qzy@math.pku.edu.cn

***Abstract*—The smart pointer mechanism, which is improved in the continuous versions of the C++ standards over the last decade, is designed to prevent memory-leak bugs by automatically deallocating the managed memory blocks. However, not all kinds of memory errors can be immunized by adopting this mechanism. For example, dereferencing a null smart pointer will lead to a software failure. Due to the lack of specialized support for smart pointers, the off-the-shelf C++ static analyzers cannot effectively reveal these bugs.**

In this paper, we propose a static approach to detecting memory-related bugs by tracking the heap memory management of smart pointers. The behaviors of smart pointers are modeled during their lifetime to trace the state transitions of managed memory blocks. And the specially designed checkers are used to check the state changes according to five collected bug patterns. To evaluate the effectiveness of our approach, we implement it on the top of the Clang Static Analyzer. A set of handmade code snippets, as well as nine popular open-source C++ projects, are used to compare our tool against four other analyzers. The results show that our approach can successfully discover nearly all the built-in bugs. And 442 out of 648 reports generated from the open-source projects are true positives after manual reviewing, where the bugs of dereferencing null smart pointers are most frequently reported. To further confirm our reports, we design patches for Aria2, Restbed, MySQL and LLVM, in which seven pull requests covering 76 bug reports have been merged by the developers up to now. The results indicate that pointers should always be carefully used even after migrated to smart pointers and static analysis upon specialized models can effectively detect such bugs.

***Index Terms*—C++ Smart Pointer, Memory Errors**

I. Introduction

Because of the inconveniences of the C/C++ language on the way to manually manage heap memory, generations of developers and maintainers have spent decades fighting against memory errors, such as memory leak, null pointer dereference, and so on. Fortunately, with the evolution of C++ standards (the continuous versions since C++11), developers now are equipped with the *Smart Pointer* mechanism to help them automatically manage heap memories.

Figure 1 presents the trend of smart pointer usages crossing the last decade. The lines represent the statistics on GitHub when searching with keywords of *unique_ptr*, *shared_ptr*, *weak_ptr* and *smart pointer*. The entries in the figure indicate

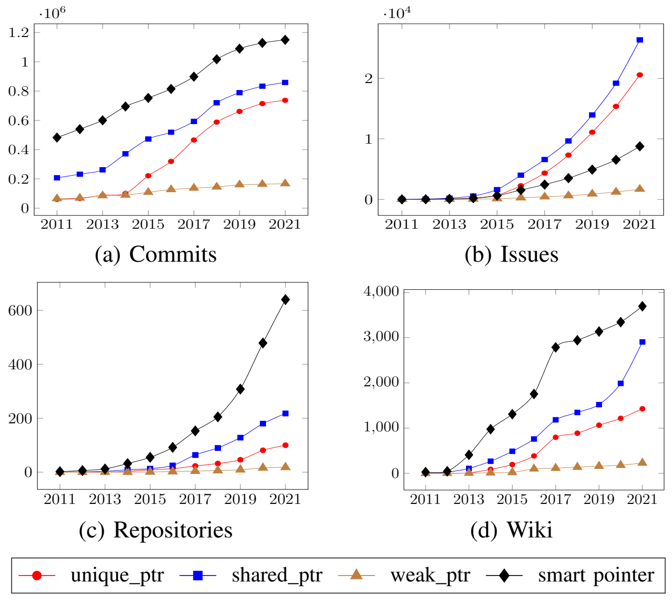

Fig. 1: Trend of smart pointer usages on GitHub

the number of the entities which explicitly mention these keywords. According to the results, more and more projects begin to use smart pointers to manage heap memory approximately since 2013 and 2014.

The C++ smart pointers are designed to prevent memory-leak bugs. The memory blocks assigned to smart pointers can be automatically deallocated when the smart pointers go out of scope. Besides, as the calls to destructors are scheduled by the compiler, smart pointers can also prevent use-after-free bugs caused by improperly scheduled manual deallocations [1].

However, using smart pointers cannot immunize all kinds of memory errors. The undefined behaviors of smart pointer methods presented in the *CPP Reference* [2] can lead to memory errors such as dereferencing null smart pointers and so on. In addition, improper usages of smart pointers, which are prohibited by the *C++ Core Guidelines* [3] and widely used coding regulations [4, 5], can also lead to memory leaks and efficiency issues [6, 7]. In this paper, we call all these memory-related bugs caused by smart pointers the *Misuses of Smart Pointers* (MisSP).

Static analysis is an effective way to check program defects.

§Corresponding authors

However, according to our experiments on a handmade benchmark presented in Section V-B1, the off-the-shelf tools cannot provide us with satisfying results. On one hand, some coding style checkers like *CppCheck* [8] and *SPrinter* [6] can only detect coding style problems and a few intra-procedural MisSPs in some simple functions. On the other hand, static analyzers like *Clang Static Analyzer* [9] and *Facebook Infer* [10] will miss a lot of real bugs according to our experiments.

The shortcomings of the off-the-shelf analyzers can be summarized into two aspects: model and checker. On one hand, it is a straightforward approach to model smart pointers by directly analyzing their implementations in the C++ Standard Library. However, as the most commonly used implementations (GCC libstdc++, Clang libc++ and MSVC Standard Library) are all very complex, it is difficult and inefficient to model the behaviors in this way. On the other hand, new checkers should also be specially designed to reveal the MisSPs that cannot be reported by memory error checkers.

Facing the shortcomings, we propose our models of smart pointers and checkers on MisSPs, which can be seen as a smart-pointer extension of the typestate analysis on heap memory blocks [11, 12]. We extend the states of heap memory, which are widely used to check memory errors [13, 14], to cooperate with the model of smart pointers. And a set of meta-operations are defined to model the method calls and record the states of smart pointer objects. Besides, new checkers are specifically designed based on five error patterns extracted from the C++ standard and famous coding regulations. And MisSPs are checked with these checkers according to the states of heap memory and smart pointers.

We implement our analyzer, *Spelton*, on the top of the *Clang Static Analyzer* [9] and evaluate it with a handmade benchmark and nine open-source C++ projects. The experimental results indicate that our approach can effectively detect MisSPs. Our main contributions are listed as follows:

- We model the behavior of smart pointers via a set of meta-operations. And the states of heap memory blocks are also extended to be managed by smart pointers.
- An analyzer with checkers specially designed for five extracted bug patterns is implemented to systematically scan for MisSPs on C++ projects;
- Our discoveries and suggestions on using smart pointers are also presented based on our statistics and bug reports.

II. Background

In this section, we will introduce the usages of smart pointers, the basic concepts that smart pointers are based on, as well as the different kinds of smart pointers defined in the C++ standard.

A. Usage of C++ Smart Pointers

To roughly understand how smart pointers work, we present the comparison of managing memory manually and using smart pointers. The following two functions implement the same functionality line by line. They first allocate a `Dog` object, then calls its method `bark` and deallocates it at last. The left snippet manages the object manually, whereas the one on the right-hand side uses smart pointers.

```
void ManualBark() {          void AutoBark() {
  Dog *dog = new Dog;          unique_ptr<Dog> dog(new Dog);
  dog->bark();                 dog->bark();
  delete dog;                  // No manual deallocations.
}                            }
```

The comparison shows their two main differences. First, a smart pointer object instead of a raw pointer is used to point to the allocated memory. And second, manual deallocation for the managed memory is not needed when using smart pointers.

B. RAII and Memory Ownership

1) RAII based Resource Management: C++ smart pointers are designed based on a common programming idiom in Object-Oriented Programming called *Resource Acquisition Is Initialization* (RAII) or *Scope-Bound Resource Management* (SBRM) [15]. The names explain this idiom from its two main characteristics. First, the resource is allocated during the initialization of a manager object. And second, their lifetimes are bound to the corresponding scope via the variable of the manager object. Therefore, the resource can be properly deallocated when it is no longer accessible.

The smart pointers take the idiom to manage the memory blocks assigned to it. More specifically, the manually allocated objects and arrays will be automatically deallocated during the destruction of the smart pointer objects that they are assigned to. And a smart pointer object will be destructed when it goes out of scope as expected (*e.g.* the function normally returns) or accidentally (*e.g.* an exception is thrown from this scope). As the destruction is scheduled by compilers, the idiom can significantly help to prevent memory errors caused by improperly scheduled manual deallocations, such as double-free and use-after-free bugs [1].

2) Ownership of Heap Memory Blocks: Different from other RAII-based manager classes, such as the file IO stream, whose resources can only be used through the manager object, the allocated memory can also be used via a raw pointer. To take advantage of both smart pointers and raw pointers, it is suggested by some coding guidelines [3, 4, 5] to use the memory block through a raw pointer, and leave the smart pointers managing the deallocation of memory blocks only. These suggestions indicate that smart pointers are the *owners* of their managed memory blocks.

The *ownership* of a heap memory block represents the responsibility of deallocation [16]. If an object has a field pointing to a heap memory block and deallocates the block in its destructor, such as smart pointers, we call these kinds of objects *owners*. While for other blocks that are manually deallocated via a raw pointer, their ownership is held by the programmer. Ownership can be either transferred between owners or shared by a group of owners. When ownership is *uniquely* held by one owner, it can be transferred to other owners. And this unique owner is responsible to deallocate the memory. While if the ownership is *shared*, it cannot be

TABLE I: Smart pointers in C++ standard and their features

Name	Standard	Ownership	Dereference	Array
`auto_ptr`	C++ 98	*unique*	Yes	No
`unique_ptr`	C++ 11	*unique*	Yes	Yes
`shared_ptr`	C++ 11	*shared*	Yes	Yes
`weak_ptr`	C++ 11	N/A	No	Yes

transferred but can only be released by one of these owners, and the last owner reserved will deallocate the memory.

C. Smart Pointer Implementations in C++ Standard

We summarize all kinds of smart pointer classes of the C++ standard in Table I. The entries presented in the table include the class *Name*, the *Standard* in which the class is introduced, the kind of *Ownership* management, whether users can *Dereference* the smart pointer, and whether managing dynamic *Array* (allocated with operator `new[]`) is supported.

In C++ 98 standard, the auto pointer (`auto_ptr`), the first smart pointer class, is introduced. However, as a consequence of its ill-formed copy semantics, it is not widely used. And it is replaced by the unique pointer (`unique_ptr`) in the C++11 standard and fully removed in the C++17 standard.

Then in C++11 standard, the concept of smart pointers is further perfected with three new smart pointers. The unique pointer is designed as a *unique* owner, in contrast with the shared pointer (`shared_ptr`), which is used as a *shared* owner. Besides, to prevent possible memory leaks caused by circular referencing of shared pointers [17], the weak pointer (`weak_ptr`) is also introduced in the C++11 standard. Different from the two kinds of owners, the weak pointers are not owners and only reference the ownership managed by a group of shared pointers. Therefore, they cannot be dereferenced. To access the memory, a weak pointer needs to be cast to a shared pointer.

Since the auto pointers are deprecated and become seldom used, we mainly focus on the weak pointers and two *owner pointers* (the unique and shared pointer) in this paper.

III. Misuses of Smart Pointers

In this section, we will introduce the *Misuses of Smart Pointers* (MisSP). A motivating example will first show the real-world MisSPs, then each pattern that we extracted is explained with a simple example.

A. Motivating Example

The code presented in Figure 2 is composed of three real-world MisSPs found in three different projects [18, 19, 20]. Among them, bug 1 and 2 are found and reported by us, and bug 3 is found from the commit history and can also be reported by our tool. There is one class and two functions in the code. Class `Request` on line 1 has three methods, and the other two functions, `HandleRequest` (line 6) and `Entry` (line 11), work with a `Request` object managed by a unique pointer.

When using an instance of class `Request`, users can add a buffer to the instance via method `addBuffer` and handle the request with method `HandleWithBuf`, or users can call function `HandleWithoutBuf` directly without adding a buffer. The buffer added with method `addBuffer` will not be deallocated in the class, and the function returns `true` if the buffer is successfully added.

```
 1  struct Request {
 2    bool addBuffer(char *b);
 3    static void HandleWithBuf(Request &R);
 4    static void HandleWithoutBuf(Request &R);
 5  };
 6  void HandleRequest(unique_ptr<Request> r,
 7                     bool hasBuffer) {
 8    if (r && hasBuffer) Request::HandleWithBuf(*r);
 9    else Request::HandleWithoutBuf(*r); // Bug 1
10  }
11  void Entry(unique_ptr<Request> R) {
12    unique_ptr<char> b(new char[size]); // Bug 2
13    HandleRequest(move(R),
14                  R->addBuffer(b.get())); // Bug 3
15  }
```

Fig. 2: The motivating example

Function `HandleRequest` and `Entry` implement an example of handling requests. The execution of the example starts from function `Entry`. A buffer is allocated and assigned to a unique pointer. Then the buffer is lent to the argument instance of class `Request`. Then the request is moved to function `HandleRequest` to be handled.

The three bugs are commented in the example. On line 8, the nullity of unique pointer `r` is checked. And it can be inferred that if pointer `r` is null, the operator `*` call on the else branch (line 9) will dereference a null smart pointer (Bug 1). On line 12, a buffer is allocated with operator `new[]` and is then assigned to a unique pointer of type `char`. This indicates that operator `delete` will be used during deallocation, which cannot be matched with the allocator operator (Bug 2). And on line 13, the unique pointer `R` is moved to the parameter `r` of function `HandleRequest`. Then the operator `->` will dereference a null smart pointer on line 14 (Bug 3).

Although the three bugs can all trigger program crashes, the off-the-shelf memory error checkers cannot provide us with desired reports. We tried to scan the example code with the latest version of two static analyzers (*Clang Static Analyzer* (CSA) [9] and *Infer* [10]) and two coding style checkers (*CppCheck* [8] and *SPrinter* [6]). Among these four tools, CSA can only report a use-after-moved bug on line 14, which seems less important than dereferencing a null smart pointer; whereas *SPrinter* can correctly report bug 2 on line 12, but the report will be suppressed if there is an intermediate raw pointer forwarding the address. And the other two tools fail to report any of the bugs.

B. Patterns of MisSPs

The bug patterns of *Misuses of Smart Pointers* (MisSP) are extracted from the undefined behaviors in the C++ standard (according to the introductions on the *CPP Reference* [2]) as well as the rules in the *C++ Core Guidelines* [3] and the *Smart Pointer guidelines* of the *Chromium* project [5]. Five patterns are finally selected, as they are likely to cause a crash or have

```
void DereferenceNull() {
  unique_ptr<int> p;
  *p = 5;
}
```

(a) Dereference Null (DN)

```
void BadAssignment() {
  int a = 5;
  unique_ptr<int> p(&a);
}
```

(b) Bad Assignment (BA)

```
void TypeMismatch() {
  unique_ptr<int> op;
  op.reset(new int[5]);
  unique_ptr<int[]> ap;
  ap.reset(new int(5));
}
```

(c) Type Mismatch (TM)

```
struct T
  { shared_ptr<T> t; };
void CircularReference() {
  shared_ptr<T> t(new T);
  t->t = t;
}
```

(d) Circular Reference (CR)

```
int use(shared_ptr<int> p2) {
  return p2 ? *p2 + 5 : 0;
}
int UniqueShared() {
  shared_ptr<int> p1(new int(5));
  return use(p1) + 5;
}
```

(e) Unique Shared (US)

Fig. 3: Examples of Misuses of Smart Pointers

efficiency issues that have not been paid enough attention to. We will further introduce the reason why these patterns are selected and what the other patterns are in Section VI. These five kinds of MisSPs can be illustrated with the simple examples presented in Figure 3.

Dereference Null (DN). The term *dereference null* is short for dereferencing a null smart pointer. Similar to raw pointers, dereferencing a null smart pointer will also lead to a crash. For the two kinds of *owner smart pointers*, *i.e.* the unique and the shared pointers, they can be dereferenced via the overridden operator *arrow* (`->`), *star* ($\star$) and *subscription* (`[]`). Therefore, we report a *dereference null* bug if these operators are called with a null smart pointer.

Bad Assignment (BA). The memory managed by owner smart pointers will always be deallocated. Therefore, when a smart pointer points to a memory block that should not be deallocated, such as stack memory and so on, we will report the assignment as a *bad assignment* bug.

Type Mismatch (TM). The type argument of the smart pointer will determine whether operator `delete` or `delete[]` will be used to deallocate the memory. If it does not match with the allocator of the managed memory, we will report a *type mismatch* bug on the assignment.

Circular Reference (CR). All reference-counting-based memory management mechanisms suffer the circular referencing problem, so are the shared pointers. However, a garbage collector is not available for shared pointers to reclaim the memory in the cyclic structures. When a ring of memory blocks is connected with smart pointers, we will report a *circular reference* bug.

Unique Shared (US). Shared pointers consume more memory and time than unique pointers [7] during execution. Besides, shared pointers are infectious. When a shared pointer is used, all related pointers should also be replaced with shared pointers. Since there would be many shared pointers used, it is usually hard to infer which shared pointer actually deallocates the managed memory [5]. Therefore, as a consideration of program efficiency and maintainability, unique pointers should be used if the ownership is not semantically shared [3].

In the example, the parameter `p2` is only dereferenced without sharing the ownership to other objects or containers. And pointer `p1` is used to automatically deallocate the memory and satisfy the interface of calling function `use`. In this case, we will report a *unique shared* bug for pointers `p1` and `p2`. The solution is to change pointer `p1` to a unique pointer to manage the memory block and replace `p2` with a raw pointer to dereference the memory.

IV. Approach of Checking MisSPs

In this section, starting with the workflow, we will introduce our extension of memory block states, models of smart pointers, as well as checkers for the bug patterns. And a case study on the motivating example will be presented at the end of this section to concretely illustrate our approach.

A. Overview of Workflow

To explain how MisSPs are checked, we present the workflow of our approach in Figure 4. The components surrounded with dashed lines represent the ones that are newly added to model and check smart pointers.

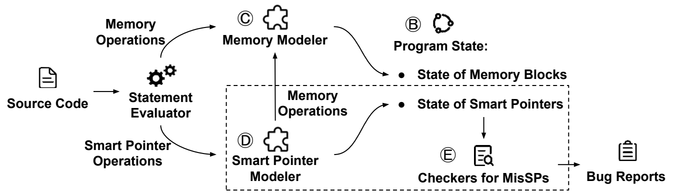

Fig. 4: Workflow of checking MisSPs

The *Statement Evaluator* will parse the input file and analyze each function according to the topological order of the call graph. It is used to explore program paths and evaluate the encountered statements that are not related to heap memory and smart pointers. For the operations on raw pointers and memory blocks, they are modeled by the *Memory Modeler* (Section IV-C), which is a typestate analysis on the heap memory blocks. And the *Smart Pointer Modeler* (Section IV-D) extends the typestate analysis with new states and operations of heap memory and smart pointers.

The modelers will modify the *Program State* (Section IV-B) according to the operations executed. And if a smart pointer operation needs to modify the state of its managed memory block, the *Memory Modeler* will be invoked by the *Smart Pointer Modeler* to handle the corresponding memory operations. When applying the model, the states of memory blocks and smart pointers will be checked by the *Checkers for MisSPs* (Section IV-E) and bug reports will be generated if any MisSPs are found.

B. Program State

The program states record the states of memory blocks and smart pointers. We use *Mem* to represent the set of all valid heap memory blocks, which is composed of the allocated memory set AM, the set of captured memory from external contexts EM and the set of null pointer values Λ. For simplicity, we use λ to refer to the global null pointer constant. And Set Ptr is used to denote all of the smart pointer objects among the entire program, together with set Var to represent all variable names in the code.

In the C++ language, three pairs of allocators and deallocators are used to manage heap memory: managing *raw* blocks with function `malloc`, `free` and *etc.*, managing dynamic *objects* with the operator `new` and `delete`, and managing dynamic *arrays* with the operator `new[]` and `delete[]`.

We use enumerate value Raw, $Object$ and $Array$ to represent the type of the allocators and deallocators that are used in these three kinds of memory management approaches respectively. And for simplicity, the set $\{Raw, Object, Array\}$ is represented as ADT. Besides, value Any is used as the type of the allocator of a memory block captured from the external context, which indicates that we do not know how it is allocated, and it can be reclaimed with all kinds of deallocators. And there are no deallocators of type Any.

With the definitions of the above sets, we define the program state as a triplet $\mathcal{S} = \langle M, P, Q \rangle$ where

- $M \subseteq Mem$ is the set of monitored memory blocks that are being analyzed in the current context;
- $P \subseteq Ptr$ indicates the set of all traced smart pointers;
- $Q : M \to \mathcal{P}(P)$ represents the map from a monitored memory block to the set of all its owner smart pointers.

For a monitored memory block $m \in M$, it is defined as a triplet $m = \langle id, at, st \rangle$ where

- $id \in \mathbb{N}$ is a natural number used as the identifier of the memory block;
- $at \in ADT \cup \{Any\}$ represents its allocator type;
- $st \in MemSt$ denotes the state of the memory block, where the set of all states is $MemSt = \{Allocated, Captured, Escaped, Usable, Null, Freed, Bad\}$.

And we define a traced smart pointer $p \in P$ as a triplet $p = \langle vn, dt, tm \rangle$ where

- $vn \in Var$ is its variable name used as a unique identifier;
- $dt \in ADT$ represents the type of its pending deallocator;
- $tm \in Mem$ indicates the memory block it points to.

When a monitored non-null memory block $m \in M \setminus \Lambda$ is assigned to a traced owner pointer $p \in P$, we have $p.tm = m$. And it is required that pointer p should be added to block m's owner set $Q[m]$ and removed after unassigned.

In the following subsections, we will introduce the operations on memory blocks and smart pointers, as well as the transitions of the program state.

C. Modeling State Transitions of Memory Blocks

The state transitions of memory blocks are modeled with the *Memory Modeler*. For a memory block m, its identifier $m.id$ and allocator type $m.at$ are determined when it is created, and will not be changed then. However, the operations performed on memory blocks will change its state $m.st$. To support checking C++ code with smart pointers, we extend the state

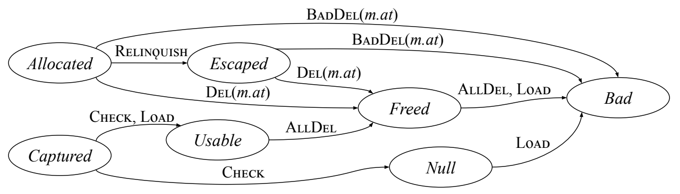

Fig. 5: State transitions of memory blocks

transitions of heap memory, which are widely used to check memory errors [13, 14].

The operations on a memory block are composed of three kinds of DEALLOCATE operations of the *ADT* types, as well as the CHECK, LOAD and RELINQUISH operations. To distinguish different kinds of DEALLOCATE operations, for a deallocator of type t, function DEL(t) is used to represent the corresponding DEALLOCATE operation. And for a monitored memory block m, we define another function BADDEL$(m.at)$ to represent the set of DEALLOCATE operations with mismatched deallocators, where

$$\textsc{BadDel}(m.at) = \begin{cases} \emptyset & m.at = Any \\ \{\textsc{Del}(t) \mid \forall t \in ADT \setminus \{m.at\}\} & m.at \in ADT \end{cases}$$

Figure 5 presents the state transitions of a memory block m, where ALLDEL represents all three kinds of DEALLOCATE operations. The operations that will not change the states are omitted for simplicity.

For an allocated memory block $m_a \in AM$, its initial state is *Allocated.* And the type of the corresponding allocator will be stored as its allocator type $m_a.at$. If m_a is then managed by a group of shared pointers, and one of the pointers is sent to an uninterpreted function or inserted into an STL container, operation RELINQUISH will change its state to *Escaped.*

For a memory block m_e captured via an external pointer ($m_e \in EM$), its initial state is *Captured* and its allocator type is set to *Any*. On one hand, if m_e is directly dereferenced via operation LOAD, we will modify its state to *Usable*. On the other hand, if its nullity is checked with operation CHECK, the state will be changed to *Usable* on the then branch whereas *Null* on the else branch. The *Usable* state indicates that memory block m_e is confirmed to be non-null on the then branch. While the *Null* state on the else branch represents m_e is a null constant, and block m_e will also be moved from the set of external memory EM to the null constant set Λ.

All the *Allocated*, *Escaped* and *Usable* memory blocks can be deallocated. The only exception is the *Escaped* memory blocks whose deallocation via shared pointer destructors will be omitted, as we cannot know whether there are any other owners left. When a memory block m is reclaimed with the DEALLOCATE operations that are not in the invalid deallocator set BADDEL$(m.at)$, its state can be changed to *Freed.*

Apart from the transitions mentioned above, there are also edges representing memory errors, which will change the state to *Bad*. These errors are *mismatched memory management routines* (*Allocated* and *Escaped* to *Bad*), *use after free* (*Freed* to *Bad*) and *null pointer dereference* (*Null* to *Bad*).

D. Modeling Operations on Smart Pointers

The *Smart Pointer Modeler* manages the state transitions of smart pointer objects. The states mainly focus on assignments and ownership management. When a new smart pointer p is defined or an external one is referred to for the first time, we will create an object for it. Its variable name $p.vn$ and deallocator type $p.dt$ are directly set according to its declaration and will not be changed during its lifetime. Its value $p.tm$ will be assigned to the null pointer constant μ before it is further used via other operations.

As the owner pointers and weak pointers have differences in ownership management, we will separately introduce their operations in the next two subsections.

1) Owner Smart Pointer Operations: To model the assignment of owner pointers, we define two meta-operations, SET and UNSET, to assign or unassign a smart pointer. Their operational semantics are presented below.

$$[\textsc{Set}]\ \frac{p \in P, m \in M, p.tm \in \Lambda, p \notin Q[m]}{\textsc{Set}(p,m) : p.tm := m, Q[m] := Q[m] \cup \{p\}} \tag{1}$$

$$[\textsc{Unset}]\ \frac{p \in P, m \in M, p.tm = m, p \in Q[m]}{\textsc{Unset}(p) : p.tm := \mu, Q[m] := Q[m] \setminus \{p\}} \tag{2}$$

The SET operation assigns a memory block m to an empty smart pointer p whereas the UNSET operation clears a previous assignment. When assigning with operation $\textsc{Set}(p, m)$, the value of the assigned pointer $p.tm$ will be modified to the assigned memory block m, and p will be added to m's owner set $Q[m]$; vice versa for operation $\textsc{Unset}(p)$. After operation UNSET, the pointer will be reassigned to null.

With the definition of two meta-operations and operations on memory blocks, we can now define the operations on owner pointers. The operational semantics of these operations are presented below. Among these operations, apart from the SHARE operation which can only be used on a shared pointer, the others can be applied to both kinds of owner pointers. For simplicity, we will omit the preconditions of requiring the mentioned pointers to be traced ($p \in P$).

$$[\textsc{Clear}]\ \frac{|Q(p.tm)| = 1}{\textsc{Clear}(p) : \textsc{Del}(p.dt)(p.tm); \textsc{Unset}(p)} \tag{3}$$

$$\frac{|Q(p.tm)| \neq 1}{\textsc{Clear}(p) : \textsc{Unset}(p)} \tag{4}$$

$$[\textsc{Assign}]\ \frac{m \in M, m.st \notin \{\mathit{Freed}, \mathit{Bad}\}, Q[m] = \emptyset, \quad m.at = \mathit{Any} \vee (m.at \neq \mathit{Any} \wedge m.at = p.dt)}{\textsc{Assign}(p,m) : \textsc{Clear}(p); \textsc{Set}(p,m)} \tag{5}$$

$$[\textsc{Share}]\ \textsc{Share}(p_1,p_2) : \textsc{Clear}(p_2); \textsc{Set}(p_2, p_1.tm) \tag{6}$$

$$[\textsc{Move}]\ \textsc{Move}(p_1,p_2) : \textsc{Clear}(p_2); \textsc{Set}(p_2, p_1.tm); \textsc{Unset}(p_1) \tag{7}$$

$$[\textsc{Access}]\ \frac{p.tm \notin \Lambda}{\textsc{Access}(p) : \textsc{Load}(p.tm)} \tag{8}$$

$$[\textsc{Inspect}]\ \textsc{Inspect}(p) : \textsc{Check}(p.tm) \tag{9}$$

Operation CLEAR is used to break the original assignment, which is used when destructing or reassigning a smart pointer. For a unique pointer and the last live shared pointer, definition (3) is used to deallocate the managed memory block and set the pointer to null. Whereas when clearing an empty smart pointer or one of a group of shared pointers, definition (4) can unassign the pointer and leave the memory unchanged.

In contrast with operation CLEAR, three operations are used to assign an owner pointer. Operation ASSIGN is used to reassign a pointer. The SHARE operation is used to share the ownership with another shared pointer. And operation MOVE can transfer the ownership from one smart pointer to another. When assigning an owner pointer, the previous assignment will be canceled with operation CLEAR, and then the new value will be assigned with operation SET. And for the MOVE operation, the source pointer will also be CLEAR-ed.

The other two operations, ACCESS and INSPECT, are just wrapped memory block operations. They are used to LOAD or CHECK a memory block respectively.

The preconditions of ASSIGN and ACCESS operation are used to check the MisSPs. We will further introduce them in Section IV-E.

2) Weak Pointer Operations: For the weak pointers, as they do not manage the ownership and cannot be dereferenced, they only have two operations: CATCH and LOCK. The corresponding operational semantics are presented below.

$$[\textsc{catch}]\ \textsc{Catch}(p_w,p_s) : p_w.tm := p_s.tm \tag{10}$$

$$[\textsc{lock}]\ \frac{p_w.tm = m \wedge m.st = \mathit{Captured}}{\textsc{Lock}(p_s,p_w) : \textsc{Check}(m); \textsc{Lock}(p_w,p_s)} \tag{11}$$

$$\frac{p_w.tm = m \wedge m.st \in \{\mathit{Allocated}, \mathit{Escaped}, \mathit{Usable}\}}{\textsc{Lock}(p_w,p_s) : \textsc{Share}(p_w,p_s)} \tag{12}$$

$$\frac{p_w.tm = m \wedge m.st \in \{\mathit{Null}, \mathit{Freed}, \mathit{Bad}\}}{\textsc{Lock}(p_w,p_s) : \textsc{Clear}(p_s)} \tag{13}$$

The CATCH operation is used to reference the ownership of a group of shared pointers. As the weak pointers are not owners of the managed memory block m, we will only reset the value without adding the pointer to the owner set of the pointee memory block $Q[m]$.

And the LOCK operation is used to create a new shared pointer from the referenced ownership. If a weak pointer pointing to a memory block m with a *Captured* state, operation CHECK will be first used to determine its nullity, and a shared pointer p_s will be created with a recursive call on both the *Usable* and *Null* branches. When the state of m is *Allocated*, *Escaped* or *Usable*, the created shared pointer p_s will share the ownership of m. And if m cannot be dereferenced, *i.e. Freed*, *Null* or *Bad*, a null shared pointer will be created. A weak pointer is *expired* when its memory block is deallocated.

E. Checkers for MisSPs

The checkers are specially designed for the five patterns. The states of related memory blocks and smart pointers, as well as the owner sets Q will be checked by the checkers.

The precondition of operation ACCESS ($p.tm \notin \Lambda$) requires the dereferenced smart pointer should not point to a null constant, which is used to check the *dereference null* bugs. Besides, to detect such bugs caused by expired weak pointers, we suppose the first shared pointer created by operation LOCK on an external weak pointer is always the last remaining owner. And when the managed memory is therefore deallocated, the

TABLE II: State transitions of the motivating example

Line	Statement	Operation	Monitored Memory Block Set (M)	Traced Smart Pointer Set (P)	Map of Owner Sets (Q)
11	BEGIN	-	$\emptyset$	$\emptyset$	$\emptyset$
12	`new char[]`	Allocate Array	$\{\langle m_1, Array, Allocated\rangle\}$	$\emptyset$	$\{m_1 : \emptyset\}$
12	`b(`m_1`)`	$\textsc{Assign}(\mathtt{b}, m_1)$	$\{\langle m_1, Array, Allocated\rangle\}$	$\{\langle \mathtt{b}, Object, m_1\rangle\}$	$\{m_1 : \{\mathtt{b}\}\}$
13	`R`	$\mathcal{N} : \textsc{Move}(\mathtt{R}, \mathtt{r})$	$\{\langle m_1, Array, Allocated\rangle, \langle m_2, Any, Captured\rangle\}$	$\{\langle \mathtt{b}, Object, m_1\rangle, \langle \mathtt{R}, Object, m_2\rangle\}$	$\{m_1 : \{\mathtt{b}\}, m_2 : \{\mathtt{R}\}\}$
13	`move(R)`	$\textsc{Move}(\mathtt{R}, \mathtt{r})$	$\{\langle m_1, Array, Allocated\rangle, \langle m_2, Any, Captured\rangle\}$	$\{\langle \mathtt{b}, Object, m_1\rangle, \langle \mathtt{R}, Object, \mu\rangle, \langle \mathtt{r}, Object, m_2\rangle\}$	$\{m_1 : \{\mathtt{b}\}, m_2 : \{\mathtt{r}\}\}$
6	BEGIN	-	$\emptyset$	$\emptyset$	$\emptyset$
8	`r`	$\mathcal{N} : \textsc{Inspect}(\mathtt{r})$	$\{\langle m_1, Any, Captured\rangle\}$	$\{\langle \mathtt{r}, Object, m_1\rangle\}$	$\{m_1 : \{\mathtt{r}\}\}$
8	`if (r)`	$\textsc{Inspect}(\mathtt{r}) : \mathrm{T}$	$\{\langle m_1, Any, Usable\rangle\}$	$\{\langle \mathtt{r}, Object, m_1\rangle\}$	$\{m_1 : \{\mathtt{r}\}\}$
8	`if (r)`	$\textsc{Inspect}(\mathtt{r}) : \mathrm{F}$	$\{\langle m_1, Any, Null\rangle\}$	$\{\langle \mathtt{r}, Object, m_1\rangle\}$	$\emptyset$

weak pointer will get expired and the future calls to operation LOCK will generate null pointers only.

For operation ASSIGN, two kinds of bugs will be checked on the assigned memory block m. The first three clauses of its preconditions require that block m should have been monitored and can be deallocated ($m \in M$), and not in deallocated or error state ($m.st \notin \{\mathit{Freed}, \mathit{Bad}\}$), or managed by other owners ($Q[m] = \emptyset$). If block m violates any of the conditions, it will trigger a *bad assignment* bug.

And the last clause requires that the memory block m is either a captured external one ($m.at = Any$), or is allocated in the current context ($m.at \neq Any$) and the deallocator type of the assigned smart pointer p can match its allocator type ($m.at = p.dt$). A *type mismatch* bug will be reported if the condition is unsatisfiable.

A *circular reference* bug is defined as a ring of memory blocks linked with smart pointers. We use the owner set Q of each monitored memory block to create the point-to graph. If a cyclic structure can be found in the graph starting from the newly assigned memory block, it means the assignment triggers a *circular reference* bug. The *circular reference* checker is invoked every time after operation ASSIGN.

To check the *unique shared* bugs, the owner set of a memory block should be tracked when it is assigned to a shared pointer. When calling operation ASSIGN with a shared pointer p, it will be recorded as the first owner of the assigned memory block. And when operation CLEAR on pointer p deallocates its managed memory (definition 3), it means that the ownership is uniquely held by the first owner p and should be replaced with a unique pointer. Besides, when the first owner p_1 is the only remaining owner of its managed memory block, and operation MOVE transfers its ownership to another shared pointer p_2, pointer p_2 will become the new first owner.

F. Case Study on Motivating Example

In this subsection, we use the motivating example in Figure 2 to concretely illustrate how our checkers work. The program state transitions are presented in Table II. The suffixes *T* and *F* of operation INSPECT represent the state on the `true` or `false` branch respectively, and prefix $\mathcal{N}$ denotes the changed state before an operation.

The analysis starts from the top-level function among the call graph, *i.e.* function `Entry`. At the beginning of the function (line 11), the state is empty. The first statement in the function is the `new[]` operator call on line 12. It will create a memory block object of $\langle m_1, Array, Allocated\rangle$. As the block is not assigned to a smart pointer, its owner set $Q(m_1)$ is left empty.

Next, the memory block m_1 will be assigned to smart pointer `b` via operation $\textsc{Assign}(\mathtt{b}, m_1)$ on line 12. Before that, pointer `b` will be created as $\langle \mathtt{b}, \mathit{Object}, \mu\rangle$. Then in the operation, the checkers found that the last clause of the precondition, $m_1.at = Any \vee (m_1.at \neq Any \wedge m_1.at = \mathtt{b}.dt)$ is violated, where $\mathtt{b}.dt = \mathit{Object}$ and $m_1.at = Array$. Therefore, a *type mismatch* bug will be reported here. As the error will not make the program crash, the analysis will continue. After the assignment, the pointer `b` will be changed to $\langle \mathtt{b}, \mathit{Object}, m_1\rangle$, and it is added to the owner set of memory block m_1.

After that, we will evaluate the arguments for the call to function `HandleRequest` on line 13. As the evaluation order of the arguments is undefined in the C++ standard, we will first evaluate the call to function `std::move` and then the arrow operator call of smart pointer `R`.

The function call to `std::move` on line 13 will trigger operation $\textsc{Move}(\mathtt{R}, \mathtt{r})$, which moves the ownership from the parameter `R` of function `Entry` to the parameter `r` of function `HandleRequest`. After that, pointer `R` will be cleared, and pointer `r` will take the ownership of memory block m_2.

Finally, on the next line, operation $\textsc{Access}(\mathtt{R})$ in the arrow operator call of pointer `R` will trigger a *dereference null* bug, as it breaks the precondition of the ACCESS operation. And the analysis on this path is therefore terminated here.

As there are no remaining paths for function `Entry`, the analysis will restart from function `HandleRequest` (line 6). When checking the nullity of smart pointer `r`, a memory block m with state *Captured* will be recovered to pointer `r`. Then operation $\textsc{Inspect}(\mathtt{r})$ will fork the path and change the state of m to *Usable* or *Null* on the `true` or `false` branch respectively. And a *dereference null* problem will be then reported on the false branch with operation $\textsc{Access}(\mathtt{r})$ when calling operator `*` of pointer `r`.

V. EVALUATION

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

- **RQ 1**: *Effectiveness.* How many handmade and real-world MisSPs can be detected by our tool?

- **RQ 2**: *Usability.* How much time and memory will be consumed? What are the reasons for false alarms?
- **RQ 3**: *Discoveries.* What can we know about MisSPs from our statistics?

The first research question evaluates the effectiveness of our tool. We use a handmade benchmark and nine open-source C++ projects to present the effectiveness of our tool, and compare the ability to detect MisSPs with the off-the-shelf open-source static analysis tools. The comparison is presented in Section V-B1 and V-B2. To evaluate the usability, we also measure the time and memory cost of our tool on the open-source projects. And the composition of false positives and their reasons are also discussed in Section V-C. The third research question is used to present the discoveries during checking MisSPs. The corresponding empirical conclusions and hints on using smart pointers are presented in Section V-D.

A. Setup of Experiments

1) Implementation: We implement our tool, *Spelton*, on top of the symbolic execution engine of the *Clang Static Analyzer* (CSA) [9] 9.0.0 with all its original checkers disabled. The original CSA has very limited support of *Cross Translation Unit* (CTU) analysis, *i.e.* inlining inter-file function calls. To help *Spelton* to find the desired inter-file function definitions and import them to the current analysis context, we also improved the features and fixed the bugs in CSA 9.0.0. Besides, to speed up the analysis process and take full advantage of the system resources, *GNU Make* [21] is used to concurrently generate CTU function indexes as well as analysis the code for each source file. Therefore, our *Spelton* can be considered as a standalone MisSP finder tool.

When analyzing an input file, the analysis engine of the CSA will handle the static symbolic execution process, and its *Checker* mechanism is used to model the smart pointer method calls and check for MisSPs. When a smart pointer method is called, instead of inlining the callee, we will apply the corresponding operations of smart pointers and memory blocks, and check MisSPs based on our error patterns.

2) Environment and Tools: We set up all of our experiments on a Linux server with two Intel® Xeon® E5-2680 v4 CPUs of 56 threads and 256 GB of memory in total. We evaluate our tool against four off-the-shelf tools of the latest version including the *Clang Static Analyzer* (CSA) 11.0.1 [9], the *Infer Static Analyzer* (Infer) v1.0 [10], the *CppCheck* v2.3 [8] and the *SPrinter* v1.1 [6].

3) Benchmark Composition: Our benchmark is composed of two parts: handmade snippets and open-source projects.

The first part is the 912 handmade code snippets which are used to test the ability to model the behaviors of smart pointers and check the inserted MisSPs. Table III presents the detailed statistics of the handmade benchmark. The number of *Files*, inserted bugs (*Pos.*) and corresponding fixes (*Neg.*) are presented for each category.

The first group of 486 snippets is automatically generated by imitating the *Juliet Test Suite* v1.3 (JTS) [22]. We manually created 16 bug templates by replacing the invoked methods in

TABLE III: Statistics of handmade snippets. The imitated ones are grouped with bug types and the mutated ones are categorized with their original CWE IDs.

Types	Imitated Snippets					Mutated Snippets				Total
	DN	*BA*	*TM*	*CR*	*US*	C415	C476	C590	C762	
Files	306	36	72	54	18	36	72	238	80	912
Pos.	342	36	72	54	18	104	72	238	82	1,018
Neg.	495	66	132	99	33	132	132	343	224	1,656

TABLE IV: Information of the open-source project instances. Statistics of smart pointer utilization are presented as follows: the numbers of smart pointers of class members (#F), local variables (#L) and reference variables (#R), as well as method calls with ACCESS operations (#A) among all calls (#All).

Project	Commit	KLoc	#F	#L	#R	#A / #All
Aquila	`d5e3bd`	15.92	0	3	4	9 / 9
Aria2	`9d0a48`	125.19	1,991	344	807	5,479 / 6,773
Celero	`0d7b24`	8.37	5	7	105	176 / 191
Evpp	`867645`	60.13	72	65	94	690 / 933
Osrm	`15f0ca`	746.33	11	22	83	384 / 567
Restbed	`03f1f2`	22.86	104	64	242	734 / 877
Spdlog	`3dedb5`	32.55	1	0	43	26 / 26
MySQL	`ee4455`	3,633.72	230	234	996	2,803 / 5,088
LLVM	`bbd4eb`	5,842.38	1,089	980	3,734	15,788 / 20,824

figure 3 with the ones of the same functionality, and adding tests on our model of smart pointers. A part of them is designed for both unique and shared pointers, while others are for shared pointers only. The test code snippets are generated by applying the bug templates on 18 control flow templates extracted from the JTS.

The remaining 426 snippets are semi-automatically mutated from four types of memory errors in the JTS. For each type of memory errors selected, we first manually design how to mutate the bug to one of the patterns we check. Then, we carry out the mutation on the original snippets and merge the similar ones where only their data types of the pointers are different. And finally, the snippets are manually reviewed to remove the invalid ones that cannot be analyzed by CSA.

The second part of our benchmark is composed of nine C++ open-source projects from *GitHub*, as shown in Table IV. The selected projects are all written in C++11 or newer versions of C++ standards whose usability is important in their application fields: backend services, performance benchmarks, network libraries and compilers. For each project, we use their latest versions when carrying out the experiments. The corresponding commit hashes and kilo code lines of code are presented in the second and third columns. Besides, to present their utilization of smart pointers, we also provide the corresponding statistics in the last four columns.

B. Evaluation on Effectiveness

We evaluate the effectiveness of *Spelton* against other four off-the-shelf tools on the handmade benchmark as well as the open-source C++ projects.

1) Effectiveness on Handmade Benchmark: In this experiment, the precision and recall of all five tools are evaluated.

TABLE V: Effectiveness on handmade benchmark

Tool	*Spelton*	*CSA*	*Infer*	*CppCheck*	*SPrinter*
TP	1016	72	0	172	206
TN	1656	1656	1656	1656	1656
FP	0	0	0	4	10
FN	2	946	1018	846	812
Precision	100.00%	100.00%	N/A	97.73%	95.37%
Recall	99.80%	7.07%	N/A	16.90%	20.24%
F1	99.90%	13.21%	N/A	28.81%	33.39%

The purpose of this experiment is to use these straightforward bugs to check their ability to reveal the MisSPs.

Table V presents the statistics of these measures. Among the reports of these tools, the unrelated true reports are removed before counting, while the false ones are recorded as false positives. The first four rows present the number of True or False, Positives or Negatives. And the last three rows show the precision, recall, and F1-measure values.

According to the table, *Spelton* can find almost all bugs with no false alarms. The only two false negatives are caused by improperly handled `switch-case` statements. While other tools can only detect a part of bugs with false positives.

2) Effectiveness on Open-Source Projects: To evaluate the effectiveness on real-world C++ code, we run these tools on nine popular open-source C++ projects. Compared with the handmade benchmark, these real-world projects have massive code (8–5,842 Kloc each project) and complex features. The results are presented in the left two groups of Table VI, where the empty cells indicate no bugs are reported.

The first group of columns provides how many bugs are reported by the other four off-the-shelf tools. As there are too many reports to review, we use the information of smart pointer variables presented in Table IV to automatically find the reports related to smart pointers. The filter selects 151 reports (shown with the numbers in parentheses) , and only one report of CSA and two reports of *SPrinter* are true bugs with manual review. The bugs missed by *Spelton* are those memory errors triggered with raw pointers cooperating with smart pointers. We will have a discussion on such memory errors in Section VI.

For the results of *Spelton* in the second group , the number of reports is categorized with their types of bug patterns, and the results are presented with the ratio of true positives to all reports. According to the table, 442 out of 648 reports generated by our tool are recognized as real bugs after a manual review. And the most frequently reported bugs are the *dereference null* bugs.

To further confirm the true positives generated by *Spelton*, 14 patches are designed and submitted to the developers, where related bugs are fixed together within one patch. By now, seven patches have been merged, and five patches have not been reviewed yet. Besides, one has been fixed before we submit it; and one has been confirmed by the developers as a true bug, but they are not intended to fix such kinds of bugs. Among the merged patches, bugs in 76 reports are fixed.

C. Evaluation on Usability

The usability of *Spelton* is evaluated with the cost of time and memory. Besides, the reasons for our false positives in the open-source projects are also discussed in this Section.

1) Evaluation on Time and Memory Cost: The total time and memory costs of *Spelton* are presented in the last two groups of Table VI. The time cost is measured in seconds, while for the memory cost, we measure the peak memory cost collected from the process status. For best performance, we optimize *Spelton* with `-O3` option. The *Max* cost is measured for each input file, and the *Average* value is computed for every kilo line of source code.

According to the measurements, a project with the average size of all evaluated projects (1,165.27 Kloc) can be analyzed with 37,421.16 seconds and 8,913.68 MB of memory sequentially, or with approximately 1.3 hours and 69.64 GB of memory under an eight-thread concurrency.

2) Discussion on False Positives: More than a half (115 out of 208) of the false positives are triggered by the code that is not analyzed. They can be separated into three groups.

The first group is the *uninterpreted functions*, which indicates *Spelton* cannot find the definition of a callee function. The reasons include 1) the function is a third-party library function whose definition is not provided, 2) it is a virtual function call on an object with an unknown type, and 3) a failed CTU indexing or importing of the invoked external functions. There are 22 reports clustered to this group.

The second group is the *implicit constraints*. These constraints hold among the entire program, but fail to be collected in the current analysis context. For example an iterator class in LLVM, the nullity of its unique pointer field represents whether it is the `end` iterator or not. When analyzing the code, as we cannot collect this constraint in the current context, *Spelton* will assume that an iterator with such a null pointer is not the `end` iterator. It will therefore generate a *dereference null* report at the dereference site of the iterator. And 88 false reports can be categorized into this group.

The last group is the *disabled assertions*, where the `assert` macros in the code are replaced with a void expression when the code under analysis is compiled with assertions disabled. As the conditions in the assertions are removed during parsing, *Spelton* cannot add the corresponding constraints in the assertions, and therefore lead to false positives. There are 5 false reports in this group.

Besides, 78 *unique shared* reports considered false ones may still be true bugs or helpful to developers. In *Spelton*, we check the *unique shared* bugs for every unique path. Therefore, we will report the paths that do not share the ownership, when there are other paths in this function sharing the ownership. Since we cannot make sure whether all the involved shared pointers can be replaced with unique and raw pointers. For the sake of conservation, we mark them as false positives.

There are 3 false positives of type *bad assignment* triggered via the feature of *user-customized deleters*. This feature allows users to determine what is to be done in the DEL operation. Therefore, it will still be correct when a non-heap memory

TABLE VI: Evaluation on open-source C++ projects

Projects	Reports of Compared Tools				Reports of Spelton					Time Cost (Sec)		Memory Cost (kB)	
	CSA	Infer	CppCheck	SPrinter	DN	BA	TM	CR	US	Max	Average	Max	Average
Aquila		8	17						7/7	10.64	22.11	1,059.40	1,772.88
Aria2	4(1)	12		2(2)	6/7				15/28	162.87	32.68	4,327.29	1,682.89
Celero		3			3/3			0/1	0/2	21.12	16.17	1,591.04	1,233.15
Evpp	47(9)	14(6)	12	1				3/3	11/15	228.47	61.27	3,952.66	1,475.53
Osrm	14	12	20		0/1				6/6	401.43	8.69	5,226.96	259.47
Restbed	2	254(18)	34		1/1	0/2			2/4	237.73	26.76	1,931.03	463.75
Spdlog		1	3						3/3	15.10	1.91	312.88	52.67
MySQL	1637(14)	1668(2)	crash	89(37)	51/59	0/4	2/2		6/14	593.48	14.94	7,530.18	400.40
LLVM	657(35)	5567(18)	326	59(9)	261/364	3/3		0/3	62/116	3,086.51	45.72	20,308.85	1,220.18
Total	2361(59)	7539(44)	412	151(48)	322/435	3/9	2/2	3/7	112/195	-	289.02	-	7,833.03

block is assigned to a smart pointer with a deleter that does not deallocate the memory. As this feature is seldom used, we do not model this feature in *Spelton*.

The remaining false reports include 7 reports for we consider the reported site may be written on purpose by the developers, and 5 reports that are too complex to be checked and marked as false positives directly.

Although these reports may be difficult to be filtered automatically, it is easy for developers to filter or suppress them via automatic code transformation with the knowledge of the project [23].

D. Discoveries about Smart Pointers

According to the bug reports on the open-source projects, we have three discoveries. First, all of the real *bad assignment* and *type mismatch* bugs are caused by not using function `std::make_unique` and `std::make_shared` to create a new smart pointer. Another three commits found in the history of project *Osrm* and *LLVM* also agree with this conclusion. Therefore, it is strongly suggested to use these two makers to allocate memory and create smart pointers.

Second, developers should pay more attention to shared pointers. Although the *unique shared* bugs will not lead to crashes or resource depletion, it will strongly affect the efficiency of the program [7]. Besides, shared pointers may also trigger *circular reference* bugs. Therefore, developers had better have a double-check on whether the ownership needs to be shared before using shared pointers.

Third, smart pointers cannot immunize all kinds of memory errors, especially the null pointer dereference bugs. The C++ standard smart pointers are designed to prevent dangling pointers with null pointer values. And null pointer values are always used as the return values of abnormal circumstances. Therefore, we believe these are the reasons why the *dereference null* bugs are reported most.

VI. Threat to Validity

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

• *Error Pattern Selection.* In this paper, we only address the bugs that can be directly triggered by the method calls of three kinds of smart pointers. However, there are another two kinds of smart pointer related memory errors that can be triggered with raw pointers: 1) the memory leak bugs due to not deallocating a memory block after its ownership has been dropped with the `release` method of unique pointers, and 2) the dangling raw pointer bugs after a smart pointer deallocates its managed memory block.

Although smart pointers add new deallocation sites to the program, these two kinds of bugs are still pure raw pointer bugs, since smart pointers are designed to avoid being dangling. As they have been well studied, we prefer checking these bugs that cannot be directly triggered by smart pointers with the corresponding state-of-the-art approaches. And our model of smart pointers can be integrated into them to perfect the results.

• *Benchmark Selection.* The validity of our benchmark may be subjected to the threat that our handmade code snippets may not be complete enough to test all the circumstances. Therefore, the performance of *Spelton* on real-world projects cannot be as good as the handmade benchmark. Besides, the projects selected may not cover all language features either, there could be false negatives or positives when analyzing other complex projects.

• *CSA Analysis Engine.* When developing *Spelton*, the latest released version available was 9.0.0, which is 12.0.0 now. There are many improvements and bug fixes during this period. Some of our solutions and strategies during developing the tool can be replaced with newer and better optimizations. And the bugs of the old version may also affect the precision.

VII. Related Work

Our work is mainly related to modeling and checking smart pointers and memory-related bugs. In this section, we will mainly present the existing researches in these aspects.

• *Analysis on Smart Pointers.* Currently, there is only a little research about the C++ smart pointers on different kinds of aspects. Babati *et al.* [7] researched the performance of smart pointers under different compiler configurations, and concluded that using shared pointers is very resource-consuming. And Henriques *et al.* [24] compared the performance of different memory deallocation methods in C++, C# and Java with smart pointers of C++ and garbage collector of C# and Java. Their conclusion is the garbage collector of C# outperforms the one of Java and the smart pointers of C++. Donchev *et al.* [25] and Raj *et al.* [26] explored teaching smart pointer usages to newbies and experienced programmers respectively. Their researches introduced their experience from their courses. *SMARTOR* [27] is a tool that helps developers to replace raw-pointer-based memory management with smart pointers. Apart from C++ smart pointers, the smart pointers of the *Boost Libraries* [28] are also involved in this tool. And *SPrinter* [6] is a linter-liked tool that mainly focuses on checking coding-style problems of smart pointer usages. Some of the bug types of *SPrinter* are also supported by our tool in this paper.

• *Ownership of Memory Blocks*. The concept of ownership of memory blocks has been widely used to analyze memory errors. Svoboda *et al.* [16] introduced the *Pointer Ownership Model* to represent a similar concept of smart pointers. Heine *et al.* [29] utilized the ownership model to check for memory leak and double-free problems. They introduced a unique pointer liked strategy to use raw pointers in C programs by limiting the ownership of a memory block only managed by one owner raw pointer. And they developed a tool to check the violations of this model. And Aldrich *et al.* [30] developed a tool to annotate variables in a Java program to help programmers understanding the data flow. They introduced a static ownership model to annotate the variables, which requires the type of ownership of a variable cannot be changed.

• *Pointer Analysis*. Pointer analysis is also an important part of our work, and pointer analysis of smart pointers has many similarities with raw pointer analysis. Trabish *et al.* [31] introduced a symbolic execution aided pointer analysis method. The method uses the program state during symbolic execution to help a query on a static pointer analysis. And the results of the query are used to improve the precision of the symbolic execution. By reusing the information available, Krainz *et al.* [32] employed a diff-graph-based method to quickly analyze points-to relations incrementally. The method uses the diff-graph to represent each function, and re-computes the graph for a function if it is changed. In this way, the previous analysis results of an unchanged function can be reused. And Grech *et al.* [33] utilized a dynamic-snapshot-based static analysis method to analyze for Java programs. They use dynamic analysis to make heap snapshots, and the static analysis uses the information from the snapshots to analyze the pointers. A scalability-focused algorithm is presented by Li *et al.* [34]. They estimate the amount of information of points-to relations that will be used when analyzing a function, and select a proper sensitivity degree for each function to archive high scalability together with enough precision. And Thiessen *et al.* [35] presented a method of combing demand-driven local reasoning analysis and object sensitivity for analyzing pointers. The results indicate the method is more efficient for context-sensitive pointer analysis.

• *Typestate Analysis on Heap Memory*. Checking heap memory usages for memory errors is a typical utilization of the typestate analysis approach. *Melton* [13] and *Smoke* [14] are both tools for checking memory leaks. They model the state of memory blocks to detect the ones that go out-of-scope without being deallocated. Yan *et al.* [36] checked use after free problems by using Spatio-Temporal Context to infer the potential usages after deallocation. They summarize the program with a k-level context-sensitive pointer analysis, and diagnose use-after-free bugs by checking the intersection of the pointee set of deallocation with the set of dereferences. And *TsmartGP* [37] is a tool for checking memory errors according to pointer analysis. It uses a flow-, context-, field- and quasi path-sensitive pointer analysis to record the value of pointers and check errors with them.

Besides, fixing memory errors related to heap memory management will also use a typestate-like analysis approach to model the state of heap memory blocks. *MemFix* [1] tries to fix trivial memory leaks by rescheduling the calls to function `free`. They model the state transitions of heap memory blocks, and use the state to find a better place to deallocate the memory for all paths after they have removed all `free`s in a function. *SAVER* [38] attempts to fix heap memory errors according to given bug reports. They use a value-flow graph to model the operations on a memory block, and fix the given bug by inserting and replacing the memory operations. Finally, a typestate-like analysis is used to verify the patches generated.

VIII. Conclusion and Future Work

In this paper, we proposed an approach to modeling the state of C++ smart pointers together with the managed heap memory, and we defined a group of operations on them modifying the states. To diagnose misuses of smart pointers (MisSP), we extracted five error patterns for the operations on smart pointers and implemented checkers for the patterns. We evaluated our model and checkers on a handmade benchmark and nine open-source C++ projects. The experimental results indicate our approach can precisely detect the MisSPs.

In the future, this work can be extended in three aspects. First, as the first tool focusing on precisely checking MisSPs, we check five error patterns. More error patterns of MisSPs can be added with the evolution of the C++ standard. Second, the model can be applied to other analysis algorithms other than static symbolic execution to make the check more precise and efficient. And last, this paper only focuses on smart pointers of the C++ standard. Other kinds of smart-pointer-like manager classes or pointer-liked classes also need to be checked if their implementations cannot be precisely analyzed.

Acknowledgment

The authors would like to thank the anonymous reviewers for their suggestions. This work is supported in part by the Key Research Program of Frontier Sciences, Chinese Academy of Sciences under grant No. QYZDJ-SSW-JSC036, and the 973 Program of China (2014CB340701).

References

[1] J. Lee, S. Hong, and H. Oh, "Memfix: static analysis-based repair of memory deallocation errors for C," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 95–106.

[2] "CppReference." [Online]. Available: https://en.cppreference.com/

[3] B. Stroustrup and H. Sutter, "C++ core guidelines," *Web. Last accessed February*, 2018.

[4] E. Geisseler and P. Meier, “Gslatorptr-C++ core guidelines pointer checker and support library refactorings,” Ph.D. dissertation, HSR Hochschule für Technik Rapperswil, 2016.

[5] The Chromium Projects, “Smart Pointer Guidelines.” [Online]. Available: https://www.chromium.org/developers/smart-pointer-guidelines

[6] X. Ma, J. Yan, Y. Li, J. Yan, and J. Zhang, “SPrinter: a static checker for finding smart pointer errors in C++ programs,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1122–1125.

[7] B. Babati and N. Pataki, “Comprehensive performance analysis of C++ smart pointers,” *Pollack Periodica*, vol. 12, no. 3, pp. 157–166, 2017.

[8] “CppCheck.” [Online]. Available: https://github.com/danmar/cppcheck

[9] “Clang Static Analyzer (CSA).” [Online]. Available: https://clang-analyzer.llvm.org

[10] “Infer Static Analyzer.” [Online]. Available: https://fbinfer.com

[11] R. E. Strom and S. Yemini, “Typestate: A programming language concept for enhancing software reliability,” *IEEE Transactions on Software Engineering*, no. 1, pp. 157–171, 1986.

[12] R. DeLine and M. Fähndrich, “Typestates for objects,” in *European Conference on Object-Oriented Programming*. Springer, 2004, pp. 465–490.

[13] Z. Xu, J. Zhang, and Z. Xu, “Melton: a practical and precise memory leak detection tool for C programs,” *Frontiers of Computer Science*, vol. 9, no. 1, pp. 34–54, 2015.

[14] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang, “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, 2019, pp. 72–82.

[15] “RAII - CppReference.” [Online]. Available: https://en.cppreference.com/w/cpp/language/raii

[16] D. Svoboda and L. Wrage, “Pointer ownership model,” in *2014 47th Hawaii International Conference on System Sciences*. IEEE, 2014, pp. 5090–5099.

[17] S. Beyer, “Efficient cycle collection in a hybrid garbage collector with reference counting and mark-and-sweep,” Ph.D. dissertation, Wien, 2020.

[18] “Restbed pull request #106.” [Online]. Available: https://github.com/Corvusoft/restbed/pull/448

[19] “Mysql bug #101767.” [Online]. Available: https://bugs.mysql.com/bug.php?id=101767

[20] “Aria2 pull request #106.” [Online]. Available: https://github.com/aria2/aria2/pull/106/

[21] “GNU Make.” [Online]. Available: https://www.gnu.org/software/make

[22] P. E. Black and P. E. Black, *Juliet 1.3 test suite: changes from 1.2*. US Department of Commerce, National Institute of Standards and Technology, 2018.

[23] R. van Tonder and C. L. Goues, “Tailoring programs for static analysis via program transformation,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 824–834.

[24] L. Henriques and J. Bernardino, “Performance of memory deallocation in C++, C# and Java,” 2018.

[25] I. Donchev *et al.*, “Experience in teaching C++11 within the undergraduate informatics curriculum,” *Inf. in Education*, vol. 12, no. 1, pp. 59–79, 2013.

[26] A. G. S. Raj, V. Naik, J. M. Patel, and R. Halverson, “How to teach” modern C++” to someone who already knows programming?” in *Proceedings of the 20th Australasian Computing Education Conference*, 2018, pp. 97–104.

[27] A. Fröhlich and C. Mollekopf, “Smartor-dress naked C++ pointers to smart pointers,” Ph.D. dissertation, HSR Hochschule für Technik Rapperswil, 2013.

[28] “Boost C++ Libraries.” [Online]. Available: https://www.boost.org/

[29] D. L. Heine and M. S. Lam, “A practical flow-sensitive and context-sensitive C and C++ memory leak detector,” in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003, pp. 168–181.

[30] J. Aldrich, V. Kostadinov, and C. Chambers, “Alias annotations for program understanding,” *ACM SIGPLAN Notices*, vol. 37, no. 11, pp. 311–330, 2002.

[31] D. Trabish, T. Kapus, N. Rinetzky, and C. Cadar, “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*, 2020, pp. 197–208.

[32] J. Krainz and M. Philippsen, “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*, 2017, pp. 1–10.

[33] N. Grech, G. Fourtounis, A. Francalanza, and Y. Smaragdakis, “Shooting from the heap: Ultra-scalable static analysis with heap snapshots,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 198–208.

[34] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, “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*, 2018, pp. 129–140.

[35] R. Thiessen and O. Lhoták, “Context transformations for pointer analysis,” *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 263–277, 2017.

[36] H. Yan, Y. Sui, S. Chen, and J. Xue, “Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 327–337.

[37] Y. Wang, G. Chen, M. Zhou, M. Gu, and J. Sun, “TsmartGP: a tool for finding memory defects with pointer analysis,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1170–1173.

[38] S. Hong, J. Lee, J. Lee, and H. Oh, “Saver: scalable, precise, and safe memory-error repair,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 271–283.