SPrinter: A Static Checker for Finding Smart Pointer Errors in C++ Programs

Xutong Ma[1,3,†], Jiwei Yan[2,†], Yaqi Li[2,3,‡], Jun Yan[1,2,3,†,§] and Jian Zhang[1,3,†,§]
[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
Email: †{maxt, yanjw, yanjun, zj}@ios.ac.cn, ‡liyaqi17@otcaix.iscas.ac.cn

***Abstract*—Smart pointers are widely used to prevent memory errors in modern C++ code. However, improper usage of smart pointers may also lead to common memory errors, which makes the code not as safe as expected. To avoid smart pointer errors as early as possible, we present a coding style checker to detect possible bad smart pointer usages during compile time, and notify programmers about bug-prone behaviors. The evaluation indicates that the currently available state-of-the-art static code checkers can only detect 25 out of 116 manually inserted errors, while our tool can detect all these errors. And we also found 521 bugs among 8 open source projects with only 4 false positives.**

***Keywords*-C++ Smart Pointer; Memory Error; Coding Styles**

I. Introduction

In the programming languages without garbage collection, such as C/C++, Pascal, *etc.*, it is the programmer's job to manually deallocate the memory that is dynamically allocated in the program, which is always verbose and bug-prone. As a result, the improper manual deallocations will lead to lots of common known *memory errors* (ME), such as *memory leak* (ML), *use after free* (UaF), *double free* (DF), and so on.

To prevent these MEs, *smart pointers* (SP) are introduced into C++ to automatically deallocate the memory pieces that are no longer used. Although they are very helpful to reduce MEs, improper usages can also introduce new MEs, which are sometimes even harder to detect. We call them *smart pointer errors* (SPE).

There are mainly three factors leading to SPEs. First, the concept of the SP is unfriendly to beginners. It is difficult to explain the differences between the four kinds of standard C++ SPs [1]. Second, the API usage of the SP classes is also confusing. We have found more than 5,500 questions about SPs on *StackOverflow* by now, and the number is still increasing. Third, since SPs are used to avoid MEs, programmers may be credulous to their code when SPs are used, which will make related errors more likely to be ignored. Therefore, an SPE checker is required to detect the bug-prone and unsafe usages during compilation, so as to avoid serious MEs ahead of time.

We have also found similar requirements in the industry. The earliest functionality requirement of checking SPEs was proposed in 2012 [2]. However, there are still very limited choices of SPE checkers available in practice by now. On one hand, some C++ specific static analyzers like *CSA* [3] can detect a small part of SPEs through their ME checkers. However, their bug reports will mark only the MEs (*e.g.* the second deallocation site in a DF error), rather than the misused smart pointer APIs, which makes the reports inconvenient to use. On the other hand, other tools like *Clang-Tidy* [4] and *Cppcheck* [5] can only detect deprecated or out of date usages that will not directly lead to MEs.

In this paper, we present *SPrinter*[1], a static coding style checker for detecting SPEs and avoiding possible MEs. We extracted 10 error patterns by manually reviewing about 800 smart pointer related issue reports on *GitHub* and questions on *StackOverflow*, and *SPrinter* is implemented based on these extracted patterns.

We evaluate *SPrinter* on the benchmark composed with 32 manually designed instances and 8 open-source C++ projects. According to the experiments, *SPrinter* can find all 116 inserted errors in the 32 instances, while *Cppcheck* can only find 25 of them. For the open-source projects, *SPrinter* reported 521 errors, and 517 of them are true positives.

II. Smart Pointers

Smart pointers are designed according to the principle of *Resource Acquisition Is Initialization* (RAII), which indicates that the life cycle of the allocated memory is bound with a smart pointer object. A smart pointer object acquires the memory when it is constructed, keeps the memory valid to access in its scope, and deallocates the memory when it is destructed. The ownership to the memory can be held by one or more smart pointers, and it can be therefore categorized into two types, the *exclusive* and the *shared*.

The *exclusive* ownership indicates that the memory is uniquely occupied by one smart pointer object (for *unique* and *auto* pointers). Therefore, the ownership can only be moved from one smart pointer object to another, and the original smart pointer object will reference the null pointer, which is an invalid address, after the ownership is transferred.

For the *shared* ownership, the memory can be shared between multiple smart pointers (for *shared* and *weak* pointers). To manage those owners, the *reference counting model* is

§Corresponding authors

[1]Available at: `https://github.com/Snape3058/SPrinter`.

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00117

TABLE I: Detailed Error Patterns Checked in SPrinter

Patterns	Errors
1. Unconscious ownership transfer.	
1.1. Using auto pointers in STL containers.[1]	UaF
1.2. Declaring private auto pointer fields with default copy constructor and assignment operator.	UaF
2. Leaked ownership.	
2.1. Unused return values of `release` method.[2]	ML, DF
2.2. Using `release` method as an observer.	ML, DF
2.3. Deallocating the return value of the `get` method.	DF
3. Forked ownership.	
3.1. Using raw pointers to initialize smart pointers.	DF, FMnH
3.2. Multiple initializations with the same raw pointer.[3]	DF, FMnH
4. Invalid memory ownership.	
4.1. Using non-heap memory to initialize a smart pointer.	FMnH
4.2. Array and non-array type mismatch.	NDTM
4.3. Using weak pointers without checking for validity.	UaF

[1] Fully available in `Cppcheck`.
[2] Fully available in `Clang-Tidy`.
[3] Checked in `Cppcheck`, but the diagnostic information is confusing.

used to decide when the memory should be deallocated. And when the last owner pointer is being destructed, the memory will be finally deallocated. To prevent the ML errors caused by circular referencing, the *weak* pointers are introduced to reference the memory without sharing the ownership (the reference number will not be modified), and they should be cast to temporary *shared* pointers before being used.

To be compatible with the functions that will deallocate its argument pointer in it (the sink functions), the ownership can also be released to another owner. Then the programmers should make sure the ownership is properly transferred and the new owner will finally deallocate the memory.

Violating the principle of RAII can make the program bug-prone. Common violations, including *unconscious ownership transfer*, *leaked ownership*, *forked ownership*, and *invalid memory ownership*, can lead to different kinds of MEs. Therefore, the classes using SPs should be well designed and the SPs should also be carefully used. The corresponding error patterns will be described in detail in the next section.

III. Smart Pointer Error Patterns

In this section, we demonstrate the 10 error patterns with some detailed examples. The patterns are categorized with the type of common RAII violations mentioned in the previous section. Table I describes the SPE patterns and corresponding memory errors in detail. In addition to the errors (ML, DF and UaF) mentioned in Section I, the *freed memory not on heap* (FMnH), and *new and delete operator type mismatch* (NDTM) errors can also be checked with *SPrinter*. Since the ownership transfer will make the original SP references a null pointer, we also call the *null pointer dereference* error caused by a transferred smart pointer a UaF error for simplicity.

A. Unconscious Ownership Transfer

The first group of patterns are categorized as unconscious ownership transfer operations. Since the ownership transfers of auto pointers are carried out with copy assignments, it may be incorrectly and thoughtlessly used. Therefore, every auto pointer transfer should be carefully checked.

```
1  class Group1 {
2   public:
3    void foo(auto_ptr<int> ap) { x = *ap; }
4   private:
5    int x;
6    auto_ptr<Group1> self;
7  };
8
9  void bar() {
10   vector<auto_ptr<Group1>> obj(2);
11   *obj[1] = *obj[2];
12   auto_ptr<int> ap(new int(3));
13   obj[1]->foo(ap);
14   *ap = 3;
15 }
```

Fig. 1: Example of unconscious ownership transfer.

Figure 1 shows an example of the errors in this group. There are two diagnostic notices generated for the example based on pattern 1.1 (line 10), pattern 1.2 (line 6).

When auto pointers are used as the element type of an STL container (line 10), the assignment operations between the container elements may lead to undesired ownership transfers, which can lead to unanticipated deallocations of the managed memory. Therefore, it is dangerous to put an auto pointer inside an STL container, and this usage should be warned.

Similarly, if a private auto pointer field is declared inside a class without copy constructors and copy assignment operators defined (line 6), the copy operation between two objects of this class (line 11) will make the ownership transferred rather than the value copied. Since the syntax of C++ allows such kind of operations, the programmers may implicitly transfer the ownership by accident.

To help the programmers to debug implicit ownership transfer, we add a debug checker to present all ownership transfers of auto pointers (line 13). As presented in the example, the ownership is transferred from variable `ap` to the argument of function `Group1::foo`, then the variable `ap` will be assigned `nullptr` implicitly, and the dereference on line 14 will trigger a UaF error. With this debug checker, the programmer can be explicitly notified of all implicit transfers and correct the bug conveniently.

B. Leaked Ownership

The `release` method is used to release the ownership to other owners, while the `get` method is an observer to access the managed memory. Because of their similarity, programmers may be confused with these two methods [6]. If the released ownership is not taken by another owner, the memory will be probably leaked. And if the memory is deallocated through the observer, it will probably be accessed or deallocated again after becoming invalid.

However, when checking the real-world code based on the patterns, we found another kind of ownership leak with these two methods. In the example shown in Figure 2, class `Container` takes ownership from outside with method `take` (line 4), and the managed memory pieces are deallocated in the destructor of the class (line 7). Since the `take` method cannot take the ownership until the parameter pointer is stored in vector `v`, one safe way (no memory leak

```
1  class Container {
2   public:
3    // Acquire ownership:
4    void take(int *p) { v.push_back(p); }
5    // Destruct memory:
6    ~Container() {
7      for (auto &i : v) delete i;
8    }
9   private:
10   vector<int *> v;
11 } C;
12
13 int foo() {
14   unique_ptr<int> p = make_unique<int>(42);
15   // Transfer ownership from p to C.v:
16   C.take(p.get());
17   // Release ownership of p:
18   return *p.release();
19 }
```

Fig. 2: Example of leaking ownership to raw pointer container.

possibility) to transfer the ownership is to separate it into two steps: 1) the `Container` class accesses the memory through the observer of the smart pointer (line 16), and then 2) the ownership is released from the smart pointer after the ownership is successfully transferred (line 18).

Generally speaking, there are no memory errors during this process. However, the design of the class `Container` violates the principle of RAII, which makes the ownership leaked through the `get` method, rather than released directly to another owner. If the user forgets to release the ownership, or smart pointer `p` is a shared pointer, it will probably lead to a DF error.

C. Forked Ownership

Figure 3 presents an example of forking the ownership to two shared pointers. Since different smart pointer objects have their independent destructors, one address can only be used to initialize one smart pointer object. Otherwise, it will trigger a DF error. For shared pointers, the ownership is shared by copy assignments, rather than creating another shared pointer with the same address. And this feature is also confusing to beginners. One simple way to initialize different smart pointers with the same address is to assign the address to a raw pointer variable. To prevent this kind of error, we reported the smart pointers initialized with raw pointers (line 3 and line 4).

D. Invalid Memory Ownership

The invalid memory ownership describes three kinds of invalidity: the ownership from non-heap memory, mismatched-type memory, and expired memory. Figure 4 presents a detailed example.

Non-allocated memory should not be deallocated, hence they should not be managed by smart pointers (pattern 4.1). However, we still found some questions asking about whether

```
1 void foo() {
2   int *p = new int(42);
3   shared_ptr<int> sp1(p);
4   shared_ptr<int> sp2(p);
5 }
```

Fig. 3: Example of forked ownership.

```
1 void foo() {
2   int v = 42;
3   unique_ptr<int> up(&v);
4   auto_ptr<int> ap(new int[42]);
5   weak_ptr<int> wp;
6   { shared_ptr<int> tp(new int); wp = tp; }
7   *wp.lock() = 0;
8 }
```

Fig. 4: Example of invalid memory ownership.

they can use smart pointers managing non-allocated memory [7]. Therefore, the `&` operator computed addresses, array addresses, and literal addresses will be warned when they are used to initialize a smart pointer object. An example is shown on line 3.

Besides, a new array operator (`new[]`) should not be paired with a delete object operator (`delete`), and vice versa for `new` and `delete[]`. A checker is added to detect the type mismatch of new operators and the template argument of smart pointers (line 4, pattern 4.2), which determines the type of delete operator used in its destructor. This check is also a functionality requirement for *CSA* proposed in 2012 [2].

Since the weak pointers do not share the ownership, they should be cast to a temporary shared pointer (with the `lock` method) before being used. However, if the referenced memory gets expired (already been deallocated), which may frequently happen under multi-thread environments, the cast will fail, and dereferencing such temporary shared pointers will trigger a UaF error (pattern 4.3). To avoid such errors, we check all the dereference sites of the shared pointers who are returned by the `lock` method, and report an error if the shared pointer is not checked (line 7).

IV. Implementation

We implement our checkers on top of the *Clang-Tidy* framework, which is a widely used tool and can be integrated into various IDEs and editors. When a source file is being checked, it will be first compiled to an Abstract Syntax Tree (AST). Then the registered *AST matchers* (analogy with regular expressions for strings) are used to filter the AST sub-trees that are concerned in this check. Finally, the matched sub-trees are checked by the related *AST inspectors*.

We implement our AST matchers according to the above error patterns, and the matched sub-trees are further checked to ignore invalid or unnecessary matches. Since it is difficult to compute the real life cycle of the allocated memory, SPrinter does not provide any fix suggestions. As checkers in *Clang-Tidy* are managed with modules based on their categories, we also create a new module called *smart pointer safety* to manage all our matchers and inspectors. The usage of SPrinter is the same as *Clang-Tidy*. The SPEs can be checked by simply enabling our *smart pointer safety* module.

V. Evaluation

To evaluate the effectiveness of *SPrinter*, we ran it against *Cppcheck* (version 1.86) on 32 manually designed benchmark instances with 116 inserted errors. And we also tested it on 8 open-source projects, whose reports are manually reviewed. To avoid the impacts of other checkers in *Clang-Tidy*, we enabled

TABLE II: Results of Manually Designed Instances

Pattern ID	All Errors	SPrinter	Cppcheck
1.1	12	12	12
1.2	5	5	
2.1	16	16	
2.2	6	6	
2.3	6	6	
3.1	16	16	
3.2	16	16	11
4.1	21	21	
4.2	10	10	2
4.3	8	8	
Total	116	116	25

only our *smart pointer safety* module during the evaluation. As the total execution time is dominated by the compilation process, there is no need to evaluate the checking overhead. The command line output of running SPrinter on the code in Figure 3 is presented below:

```
4 warnings generated.
temp.cpp:2:8: warning: Raw pointer used for
    initiating multiple smart pointers.
  int *p = new int(42);
temp.cpp:3:23: warning: Initiating smart
    pointer with a raw pointer.
  shared_ptr<int> sp1(p);
temp.cpp:2:8: note: Referenced raw pointer:
  int *p = new int(42);
 ...
```

Table II presents the statistics of reports generated by *SPrinter* and *Cppcheck* on our manually designed benchmark instances. The columns indicate the ID for each pattern, the number of all the bad usages inserted in the instances, and the number of reports generated by *SPrinter* and *Cppcheck*. According to the table, our *SPrinter* can find all the inserted errors, while *Cppcheck* can find only a few of them.

Table III presents the statistics of reports for selected open-source projects generated by *SPrinter*. The columns indicate the problems we found in different projects. According to the table, *SPrinter* can find 521 problems among different patterns in all the selected projects. To ensure the correctness of the reports, we manually inspected every report generated by SPrinter. And we found that only the four reports under pattern 3.2 are false positives, and the reports under pattern 3.1 do describe true problems, but will not lead to fatal errors. According to the table, there are three patterns (1.1, 2.3 and 4.2) that cannot be matched in open-source projects. As these patterns will trigger program crashes immediately after they are executed, there is almost no chance to find them in a fully tested project.

VI. Related Works

The concept of RAII is similar to the pointer ownership model (POM) [8], which has been widely used to check or prevent memory errors of C programs in a lot of works. D. L. Heine *et al.* [9] introduced an ownership model that limits the ownership occupied by only one owner to prevent ML and DF errors, and developed a tool to check the potential violation of the ownership model. In AliasJava [10], J. Aldrich *et al.* proposed a memory ownership model with detailed ownership description, and the ownership type cannot be changed. This model is used to annotate the variables in a Java program, and help the programmers to understand its data flows. M. Maalej *et al.* [11] introduced an extension to the Ada language to provide a new feature similar to C++ smart pointers. Besides, B. Bence *et al.* [12] also analyzed the overheads of using smart pointers, which is a very useful suggestion for the programmers of performance-sensitive projects.

TABLE III: Results of Open-Source Projects

Ru. \ BI	CC	CH	CS	GS	LP	MS	OT	VC	Total
1.1									0
1.2			10						10
2.1	1	6		8	6	7			28
2.2	1				2				3
2.3									0
3.1		11	2	8	43	7		6	77
3.2					2			2	4
4.1					5				5
4.2									0
4.3							36	358	394
Total	2	17	12	16	58	14	36	366	521

The abbreviations indicate the following projects: centreon-clib (CC), ClickHouse (CH), cppcms (CS), geos (GS), LLVM-Project (LP), MySQL-Server (MS), opendht (OT) and vmpc (VC).

VII. Conclusion and Future Work

In this paper, we present a static coding style checker for C++ smart pointer errors caused by API misuses of the smart pointer classes. As a systematic tool focusing on SPEs, SPrinter can really help programmers to detect smart pointer errors and prevent potential memory errors. In the future, we will continue working on checking smart pointer errors with precise static analysis methods.

Acknowledgment

This work is supported in part by the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (CAS), Grant No. *QYZDJ-SSW-JSC036*.

References

[1] 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.
[2] D. Gribenko, "Static analyzer: add smart pointer checker," http://lists.llvm.org/pipermail/cfe-dev/2012-January/019446.html, 2012.
[3] "Clang Static Analyzer (CSA)," https://clang-analyzer.llvm.org/.
[4] "Clang-Tidy," http://clang.llvm.org/extra/clang-tidy/.
[5] "Cppcheck," https://github.com/danmar/cppcheck.
[6] "What if I delete the pointer that the smart pointer is managing?" https://stackoverflow.com/q/30294604, 2015.
[7] "C++ create shared_ptr to stack object," https://stackoverflow.com/q/38855343, 2016.
[8] D. Svoboda *et al.*, "Pointer ownership model," in *47th HICSS*, 2014, pp. 5090–5099.
[9] D. L. Heine *et al.*, "A practical flow-sensitive and context-sensitive C and C++ memory leak detector," in *PLDI*, 2003, pp. 168–181.
[10] J. Aldrich *et al.*, "Alias annotations for program understanding," *SIGPLAN Not.*, vol. 37, no. 11, pp. 311–330, 2002.
[11] M. Maalej *et al.*, "Safe dynamic memory management in Ada and Spark," in *Ada-Europe*, 2018, pp. 37–52.
[12] B. Babati *et al.*, "Comprehensive performance analysis of C++ smart pointers," *Pollack Per.*, vol. 12, no. 3, pp. 157–166, 2017.