Detection of Vulnerability Guard Elimination by Compiler Optimization Based on Binary Code Comparison

Yuka AZUMA Nagisa ISHIURA
School of Science and Technology, Kwansei Gakuin University
2-1 Gakuen, Sanda, Hyogo, 669–1337, Japan

Abstract—It is known that guards against vulnerabilities in C programs might be eliminated by compiler optimization if they are not written properly. This paper proposes a method to detect such flaws in software by binary code comparison. Given a source code, a pair of binary codes are generated, one with standard optimization and the other with problematic optimization suppressed. Since simple comparison of the binary codes end up with an unacceptable amount of false positives, call instructions in each function are collated to detect discrepancies. In a preliminary experiment on 7 programs, our method successfully detected 2 instances of guard losses with only one false positive.

I. Introduction

Cyber attacks targeting software vulnerability are critical threats to our today's society. To keep software free from such flaws, collective efforts are being made. Security critical codes are heavily audited and formal techniques may be applied in some cases.

However, even if thorough countermeasures are taken in source code level, there is no guarantee that the security is kept all the way down to the machine code. It is pointed out that compiler optimization, even if it is correctly implemented, can violate security guarantees incorporated in source code [1]. Examples of security violations induced by compiler optimization are elimination of codes to delete secret information, elimination of codes to guard vulnerabilities, destruction of code arrangement against side channel attacks, etc.

In this paper, we focus on guard elimination by compiler optimization, where carelessly written guards to protect vulnerable code sections may be deleted by optimization utilizing undefined behavior in the C language standard. We propose a method to detect the loss of guards by comparing a pair of binary codes, with and without problematic optimization. We also propose a code comparison method focusing on call instructions, which yields fewer false positives. A preliminary experiment demonstrates that guard extinction is detected by our method with very few false positives.

II. Elimination of Guards by Compiler Optimization

To protect the memory space from attacks, operations that may destroy the memory integrity must be guarded. For example, in Fig. 1 (a), dynamic memory allocation is attempted in line 6, for which the value of `size` must be checked if it is in the valid range. The code in line 5 is a guard which is intended to reject the case where overflow occurs on `size`.

Who wrote this code might expect that the value of `size` would rap around on overflow. However, the C language specification stipulates that signed overflow triggers undefined behavior, where any result is valid for the program, including continuing computation with random values or aborting the computation. C compilers, such as GCC and LLVM, make full use of the undefined behavior in a source program to generate a better code. When overflow does not occur in line 4 of the example, the comparison in the guard is always false and hence the guard is not necessary. When overflow occurs, any behavior is valid for this program, since it is undefined behavior. Taking the both cases in consideration, compilers can delete the guard in line 5, so that the memory allocation in line 6 will be exposed without protect[1].

SOF.c
```
1: #define BASE 1024
2:   ...
3: char *alloc_buff (int n){
4:   int size = BASE + n;
5:   if (size < n) error();
6:   char* p = malloc(size);
7:   ...
8: }
```
(a) Signed overflow (SOF)

NPD.c
```
 1: typedef struct{
 2:   int a[A_SIZE];
 3:   int x;
 4: } str_t;
 5: ...
 6: int sub(str_t p){
 7:   ...
 8:   int y = p -> x;
 9:   if (p==NULL) error();
10:   p -> a[k] = z;
11: }
12: ...
```
(b) NULL pointer dereference (NPD)

Fig. 1. Guard elimination by compiler optimization.

Fig. 1 (b) shows another example. The guard in line 9 rejects accesses to a struct by a NULL pointer. However, in this code, the guard was accidentally misplaced; it should have been placed before line 8. When `p` is not NULL, the condition of the guard in line 9 is always false, and the guard is unnecessary. When `p` is NULL, the dereference of `p` in line 8 triggers undefined behavior, then any behavior is valid for this program. Taking both cases into account, compilers may delete the guard in line 9. Most compilers do not generated the code to abort program on NULL pointer dereference only; if the effective is valid, the access may be granted. This led to a vulnerability in Linux kernel [2].

As a first aid fix to this problem, GCC has implemented compiler options to suppress problematic optimization. For example, `-fwrapv` option generates codes that precisely wrap round on integer overflow. However, these options are not always recommended because they slow down the codes substantially. Moreover, not all the compilers do not support all the options. GCC also has some options to warn on code elimination utilizing undefined behavior. For example, `-Wstrict-overflow` option turns on warning when code elimination utilizing signed overflow is performed. However, this option does not always work and there is no option for optimization regarding null pointer dereference. Static code analysis may detect this kind of flaw, but it may yield many irrelevant false positives.

[1]The guard would not be eliminated if the predicate were written as `(INT_MAX-BASE<n)`. Unsigned integers cause no problem because they do not overflow but wrap round.

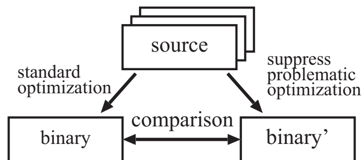

Fig. 2. Flow of the proposed method.

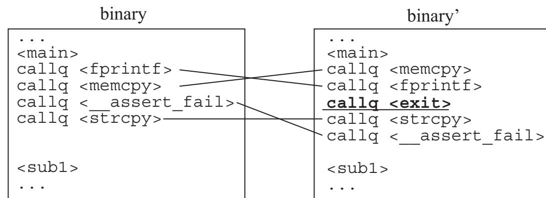

Fig. 3. Code comparison focusing on call instructions.

III. Detection of Guard Elimination by Compiler Optimization

This paper proposes a method to detect guard elimination by compiler optimization. We assume that a set of source codes for building a binary executable code is given. Then it is tested if any guard is lost during compilation.

Fig. 2 illustrates the flow of the proposed method. A pair of binary codes are generated from given source codes. One is with standard optimizing options. The other is without optimization that is known to delete guards. By comparing the binary codes (the assembly codes obtained by disassembling), loss of guards can be detected. For example, in the case of GCC, optimization tasks utilizing undefined behavior regarding signed overflow and null pointer dereference are suppressed by compiler options `-fwrapv` and `-fno-delete-null-pointer-checks`, respectively.

However, these suppressors change resulting codes more largely than we expect, all over the whole code. This is because many optimization passes work in combination and dependent on each other. As a result, a simple code differencer produces an impractical amount of false positives.

To solve this problem, we compare a pair of binaries focusing on subroutine calls. This scheme works because the guards almost always have calls error handling routines for failed assertions. In our method, call instructions in each subroutine are collated. Fig. 3 illustrates the procedure. The codes are dissected into subroutines and then call instructions (`callq`) in each function are matched by the operands. Reordering of the instructions are allowed within the subroutine. If a lonely call is found, it is reported as a potential guard loss.

IV. Preliminary Experiment

A guard loss detector based on the proposed method is implemented in Perl5. It runs on Linux (Ubuntu 18.04.2).

The result of preliminary experiment is summarized in Table I. The compiler used was GCC-7.4.0 and the default optimizing option was `-O2`. "SOF" and "NPD" are results when `-fwrapv` and `-fno-delete-null-pointer-checks` were applied which suppressed optimization regarding signed overflow and null pointer dereference, respectively. "#diff" shows the number of reported code differences and "#detect" the number of actual guard losses which were confirmed by human inspection. Two guard losses regarding signed overflow were detected with only one false positives.

TABLE I
Experimental result.

program	#line (×1000)	SOF		NPD	
		#diff	#detect	#diff	#detect
echo	4.9	0	0	0	0
mcc	1.2	1	1	0	0
libpng	4.6	1	0	0	0
zlib	13.4	0	0	0	0
libvorbis	19.2	0	0	0	0
pngquant	0.7	0	0	0	0
pngquant'	0.7	1	1	0	0

```
 1:...
 2:if ( x->type == token_STAR) {
 3:  int count = 0;
 4:  while (x -> type == token_STAR ) {
 5:   count ++;
 6:    ...
 7:  }
 8:  if (count == 1) { ...  }
 9:  else if (count > 1) { ...  }
10:  else { assert (0); }
11: ...
```

Fig. 4. Code fragment of mcc.

Fig. 4 shows a code fragment of mcc. It was found that `assert` statement in line 10 was optimized away. Note that no warning was issued even with `-Wstrict-overflow` option, so that the programmer had had no chance to know the guard loss. The difference reported on "libpng" was a false positive. This was due to the fact that `-fwrapv` option somehow suppressed function inlining of a certain function which made the number of call instructions different. Program "quant'" was a clone of "quant" but one of the guards was intentionally rewritten so that the predicate was dependent on wrap round on signed overflow. The loss of guard from this code flaw was successfully detected.

Our method just detects that guards are eliminated, which do not directly lead to vulnerabilities. However, we consider it still important to detect and remove potential vulnerabilities.

V. Conclusion

This paper has proposed a method of detecting guard elimination by compiler optimization. Loss of guards have been successfully detected in a preliminary experiment.

Currently, localization and confirmation of guard losses are done by manual inspection, which should be automated to some extent. Moreover, experiments on much more programs should be done. We will be working on these issues.

Acknowledgments—Authors would like to express their appreciation to the members of Ishiura Lab. of Kwansei Gakuin Univ. for their cooperation.

References

[1] Vijay D'Silva, et al.: "The Correctness-Security Gap in Compiler Optimization," in *Proc. IEEE Security and Privacy Workshops*, pp. 73–87 (May 2015).

[2] Cve-2009-1897, `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1897` (June 2009).

[3] "Bug 30475: assert(int+100>int) optimized away," `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=30475` (Jan. 2007).