

Is a False Positive really False Positive?

Hong Jun Choi, Hyuk Lee, Jin-Young Choi

School of Cybersecurity, Korea University, Seoul, 02841, Republic of Korea

hjcho1@korea.ac.kr, fmlab_hlee@korea.ac.kr, narnia@korea.ac.kr

Abstract— As the number of devices with software increases, software reliability and security has become more critical. To improve reliability and security, developers and test engineers use static analysis tools to find defects early in the development process. However, it takes a lot of time and effort to determine whether alarms from performing static analysis are true or false positive. In this paper, we argue that all integer overflow generated by static analysis tools are weaknesses and should eventually be corrected. To show that our argument is reasonable, we explain static analysis results for binary search program code and CWE:190 example code in terms of reliability and security. It is unnecessary to identify whether the integer overflow generated by static analysis tools is true or false positive.

Keywords— Software Reliability, Software Security, Static Analysis, Integer Overflow, True Positive, False Positive

I. INTRODUCTION

As software spreads to various industries, software dependence and complexity increase, and the probability of software defects is also increasing. According to the National Institute of Standards and Technology (NIST) report, it is cost-effective to detect and repair software weaknesses early in development [1]. The report also explains that if developers and test engineers find defects in coding or unit tests, they can fix it at the cost of ten times less than the prices found in the integration or component system tests.

To find weaknesses early and develop software at a low cost, many developers and test engineers use static analysis tools. The static analysis tool checks the program code without executing the program. Static analysis can effectively diagnose weaknesses such as overflow, Division by zero, and Out of bounds array access, which are challenging to detect with dynamic analysis [2]. However, the disadvantage is that it is difficult to determine if an alarm generated by the static analysis tool is a true positive or a false positive.

The most common and influential weaknesses in the past two years are in the 2020 Common Weakness Enumeration (CWE) Top 25 [3]. We performed a static analysis for a binary search program code that can clearly show the difference between reliability and security and the part of CWE-190: Integer Overflow, which ranks 11th in the CWE Top 25 [4]. Also, we analyzed the static analysis results in terms of software reliability and security.

In this paper, We argue that all integer overflow generated by static analysis tools are weaknesses. To support our argument, we discuss a binary search program and Juliet's test case example [5] in terms of reliability and security. By

analyzing static analysis results of them, we confirm that a false positive is a real positive. The analysis result of Juliet's false positive example code shows especially our argument better.

II. RELATED WORK

There have long been studies comparing the relationship between reliability and security in many respects. Related studies show that reliability and security have a subset relationship. However, we argue that two properties do not overlap in static analysis.

A. Reliability and Security - Convergence or Divergence [6]

- **Reliability:** The probability of the parts, components, products or systems to accomplish their functions which they were.
- **Security:** The capacity to protect the data of an organization against unauthorized access or against modifying data in order to ensure availability, confidentiality, integrity and non-repudiation.

In [6], it shows a relationship between reliability and security using the availability. [6] assumes that availability is only authorized personnel who can access data on the computer. On this basis, it says that hardware and software are reliable. Thus, [6] argues that the reliability and security relationship is as follows.

- No Reliability $\rightarrow$ No Availability $\rightarrow$ No Security
- Reliability + No Security + No Attack $\rightarrow$ Availability
- Reliability + No Security + Attack $\rightarrow$ No Availability

Without reliability, availability is not met. It leads to a lack of security. When reliability is met, and security is not met, availability is met if there are no malicious attacker attacks. And it argues that the reliability is satisfied, but the availability can be affected by malicious action.

B. Building Secure Software: How to Avoid Security Problems the Right Way [7]

- **Reliability:** A measurement of how robust your software.
- **Security:** Boils down to enforcing a policy that describes rules for accessing resources.

In [7], it says that reliability and security have many things in common, so many studies compare the two properties. Some researchers say that security is part of reliability, while others say the opposite. [7] argues that security is part of reliability because weaknesses can occur if the software is unreliable. If an attacker succeeds in attacking by using defects caused by lack of reliability, it becomes a vulnerability.

C. Software Engineering 10th Edition [8]

- **Reliability:** The ability of the system to deliver services as specified.
- **Security:** The ability of the system to protect itself against deliberate or accidental intrusion.

In [8], a computer system's dependability is a property of the system that reflects its trustworthiness. The dependability properties have five dimensions: availability, reliability, safety, security, and resilience. We only use reliability and security definitions in our study.

Increasing the system's dependability is expensive because it means additional costs in requirements, design, implementation, and testing. An example is the development of safety-critical systems. However, the software is more dependable, which means fewer weaknesses and the probability of the system's failure. So, it is important to develop a system that satisfies the five properties. In this paper, we use the definition of [8].

III. STATIC ANALYSIS

This chapter shows that all integer overflow generated by static analysis tools are weaknesses. We define a static analysis result's true and false positive in terms of software reliability and security. Then, we perform a static analysis of the binary search program code and part of Juliet CWE-190. Finally, we analyze the static analysis result and explain an example of a false positive.

A. True and False Positive Definition in Static Analysis

Using [8] definitions, we define a static analysis result's true and false positive as Table 1. If an alarm of a static analysis tool occurs as a specified input, it is a true positive in terms of reliability but a false positive in terms of security. Alarms caused by unspecified inputs are reversed. It has an exclusive relationship, so there is no overlap.

TABLE 1. TRUE AND FALSE POSITIVE IN TERMS OF RELIABILITY AND SECURITY

	Specified Input	Unspecified Input
Reliability	True Positive	False Positive
Security	False Positive	True Positive

- **Reliability**
 - **True Positive:** When an input defined in the specification is entered, the alarm generated after the static analysis is really a weakness.
 - **False Positive:** When an input defined in the specification is entered, the alarm generated after the static analysis is not a weakness.
- **Security**
 - **True Positive:** When an input that is not defined in the specification is entered, the alarm generated after the static analysis is really a weakness.
 - **False Positive:** When an input that is not defined in the specification is entered, the alarm generated after the static analysis is not a weakness.

B. Four Cases of Potential Integer Overflow

In this section, we discuss a potential integer overflow in terms of reliability and security. To discuss this, we use the well-known binary search as an example. A binary search is an algorithm that finds a specific value in a list sorted in ascending order [9]. We classify potential integer overflow into four cases in terms of reliability and security.

TABLE 2. FOUR CASES OF POTENTIAL INTEGER OVERFLOW IN TERMS OF RELIABILITY AND SECURITY

Case	Reliability	Security	Integer Overflow
1	Satisfying	Not Satisfying	Occurring
2	Not Satisfying	Not Satisfying	Occurring
3	Satisfying	Satisfying	Not Occurring
4	Not Satisfying	Satisfying	Occurring

We assume each case's requirements are as follows, and the developers write code according to each requirement without making mistakes.

- **Case 1:** Find N, a natural number in an array sorted from 1 to 100 by binary search.
- **Case 2:** Find N, a natural number in an array sorted from 1 to INT_MAX-1 by binary search.
- **Case 3:** Find N, a natural number in an array sorted from 1 to 100 by binary search. Also, the system must protect itself against deliberate or accidental intrusion.
- **Case 4:** Find N, a natural number in an array sorted from 1 to INT_MAX by binary search. Also, the system must protect itself against deliberate or accidental intrusion.

```

1 int binary_search(int array[], int search, int first, int last) {
2     int middle;
3
4     if (first > last)
5         return -1;
6
7     middle = (first + last) / 2; // Potential Integer Overflow!
8
9     if (array[middle] == search)
10        return middle;
11    else if (array[middle] < search)
12        return binary_search(array, search, middle + 1, last);
13    else
14        return binary_search(array, search, first, middle - 1);
15 }
```

Figure 1. Binary search code for Case 1, 2

Looking only at the requirements of Cases 1 and 2, there seems to be no problem. However, in Figure 1, the code implemented according to the requirements, an integer overflow will occur when adding first and last.

Case 1 satisfies reliability, but not security. Case 1 requirement is that finding N out of a number from 1 to 100. Then, first and last within the defined input range have a value from 1 to 100. However, first and last within the undefined input range can also be received from the binary_search function. In the code implemented according to Case 1, integer overflow will not occur due to first and last within the defined input range. Still, integer overflow will occur when first and last within the undefined input range are entered.

Case 2 does not satisfy reliability and security. According to the Case 2 requirement, first and last within the defined input range have numbers from 1 to INT_MAX-1. The first

and last within the undefined input range are INT_MAX. In this case, an integer overflow will occur regardless of whether it is defined as an input or not.

Since both cases 1 and 2 do not satisfy security, it allows that deliberate or accidental intrusion. The difference is whether or not the reliability is satisfied. Suppose a reliability-oriented group runs static analysis tools for a code implemented according to Case 1, and an integer overflow alarm occurs. In that case, they will judge that it is a false positive. However, suppose the same group runs a static analysis tool for a code implemented according to Case 2, and an integer overflow alarm occurs. In that case, they will determine that it is a true positive. And suppose a security-oriented group runs a static analysis tool for codes implemented according to Case 1 and Case 2, and an integer overflow alarm occurs. In that case, they will determine that it is a true positive.

```

1  int binary_search(int array[], int search, int first, int last) {
2      int middle;
3
4      if (0 > first || first > 99)
5          return -1;
6      if (0 > last || last > 99)
7          return -1;
8      if (first > last)
9          return -1;
10
11     middle = (first + last) / 2;
12
13     if (array[middle] == search)
14         return middle;
15     else if (array[middle] < search)
16         return binary_search(array, search, middle + 1, last);
17     else
18         return binary_search(array, search, first, middle - 1);
19 }

```

Figure 2. Binary search code for Case 3

Figure 2 is the code implemented according to the requirements of Case 3. Case 3 satisfies both reliability and security. According to Case 3 requirements, first and last within the defined input range have numbers from 1 to 100. And since deliberate or accidental intrusion is impossible, they cannot have a value other than 1 to 100. Therefore, integer overflow will not occur. If an alarm occurs by executing a static analysis tool for the code in Figure 2, this alarm is unconditionally false positive.

```

1  int binary_search(int array[], int search, int first, int last) {
2      int middle;
3
4      if (0 > first || first > INT_MAX)
5          return -1;
6      if (0 > last || last > INT_MAX)
7          return -1;
8      if (first > last)
9          return -1;
10
11     middle = (first + last) / 2; // Potential Integer Overflow!
12
13     if (array[middle] == search)
14         return middle;
15     else if (array[middle] < search)
16         return binary_search(array, search, middle + 1, last);
17     else
18         return binary_search(array, search, first, middle - 1);
19 }

```

Figure 3. Binary search code for Case 4

Figure 3 is the code implemented according to the requirements of Case 4. Case 4 does not satisfy reliability but security. According to Case 4 requirements, first and last within the defined input range have numbers between 1 and INT_MAX. Also, since deliberate or accidental intrusion is impossible, they cannot have values other than 1 to INT_MAX. When first and last have a value greater than 2/INT_MAX, an integer overflow will occur. Suppose an alarm occurs by executing a static analysis tool for the code in Figure 3. In that case, reliability-oriented groups will judge the alarm as true positive, and security-oriented groups will determine the alarm as false positive.

C. Static Analysis for the Four Cases

In this section, we analyze the results of the static analysis of the code for four cases.

TABLE 3. STATIC ANALYSIS RESULT FOR CODE IN FIGURE 1

Check	Function	Line	Col	Detail
Overflow	binary_search()	7	19	Warning: operation [+] on scalar may overflow (on MIN or MAX bounds of INT32)

Table 3 is the static analysis result for code in Figure 1. The Overflow checker reports that there is an overflow in line 7 of the binary_search function. Although the requirements of Case 1 and Case 2 are different, the code implemented according to each case is the same because a developer develops a program assuming that the program user enters an input defined in a specification.

The problem here is that it takes a lot of time and effort to determine whether this alarm is a true or false positive when looking only at Table 3. If Table 3 is the static analysis result of the code implemented according to Case 1, it is a false positive in terms of reliability. On the other hand, if the static analysis result of the code is implemented according to Case 2, it is a true positive in terms of reliability. Even if the defined first and last have a value of 100 based on Case 1, it is a false positive because overflow does not occur.

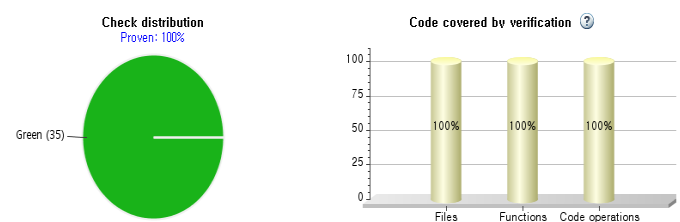


Figure 4. Static analysis result for code in Figure 2

Figure 4 is the static analysis result for code in Figure 2. In the circle graph, a Proven 100% means that there is no weakness. Since Figure 2 is the code that protects the system from deliberate or accidental intrusion, first and last can only be from 1 to 100. Therefore, there is no weakness in terms of reliability and security.

TABLE 4. STATIC ANALYSIS RESULT FOR CODE IN FIGURE 3

Check	Function	Line	Col	Detail
Overflow	binary_search()	13	19	Warning: operation [*] on scalar may overflow (result strictly greater than MAX INT32)

Table 4 is the result of static analysis for code in Figure 3. The overflow checker generates an alarm. The code in Figure 3 satisfies security but not reliability. When an alarm occurs, developers or test engineers analyze the code for that alarm and determine if it is the true positive or false positive. The developer or test engineer of the reliability-oriented group judges the result of Table 4 as true positive. The developer or test engineer of the security-oriented group determines the result of Table 4 as false positives.

TABLE 5. TRUE AND FALSE POSITIVE OF INTEGER OVERFLOW FROM A STATIC ANALYSIS PERSPECTIVE

View \ Range	Small	Large	Max
Reliability	Satisfying (False Positive)	Not Satisfying (True Positive)	Not Satisfying (True Positive)
Security	Not Satisfying (True Positive)	Not Satisfying (True Positive)	Satisfying (False Positive)
Total	True Positive	True Positive	True Positive

Table 5 shows whether the integer overflow caused after the static analysis is true or false positive. Integer overflows are divided into true or false positives depending on the input range defined in the specification and both properties. However, all integer overflow generated by static analysis tools is eventually a true positive weakness.

The small input range means no integer overflow due to the calculation of defined input values. An example of this is the requirement of Case 1. A large input range means that integer overflow can occur due to the calculation of defined input values. An example of this is the requirement of Case 2. An input range of MAX means that the calculation of defined input values will cause an integer overflow, but undefined input values will not cause it. An example of this is the requirement of Case 4.

The code corresponding to the first column satisfies reliability because it does not cause integer overflow. However, an integer overflow occurs in terms of security. We confirmed through Table 3 that an integer overflow occurs in the code of Figure 1 implemented according to Case 1. The code corresponding to the second column causes an integer overflow in terms of reliability and security. We also confirmed this through Table 3. The code corresponding to the third column has an integer overflow in terms of reliability. We confirmed through Table 4 that an integer overflow occurs in the code of Figure 3 implemented according to Case 4.

In Table 5, there are true positives and false positives in each column. Since there is at least one true positive when each column is combined, we conclude that all integer overflow generated by static analysis tools is a weakness. As a result, developers and test engineers do not need to

differentiate between true or false positives for integer overflow caused by static analysis tools.

D. False Positive Example Analysis of CWE-190

This section discusses an example in which a result of static analysis of the CWE-190: Integer Overflow test case among Juliet Test Suite for C/C++ is a false positive.

```

46 static void goodG2B()
47 {
48     char data;
49     void (*funcPtr) (char) = CWE190_Integer_Overflow__char_max_square_65b_goodG2BSink;
50     data = ' ';
51     data = 2;
52     funcPtr(data);
53 }

```

Figure 5. goodG2B Function

Figure 5 is the code of lines 46 to 53 of the CWE190_Integer_Overflow__char_max_square_65a.c file. This code calls the function of Figure 6 with parameter 2.

```

38 void CWE190_Integer_Overflow__char_max_square_65b_goodG2BSink(char data)
39 {
40     /* POTENTIAL FLAW: if (data*data) > CHAR_MAX, this will overflow */
41     char result = data * data;
42     printHexCharLine(result);
43 }
44
45 }

```

Figure 6. goodG2BSink Function

Figure 6 is the code of lines 38 to 45 of the CWE190_Integer_Overflow__char_max_square_65b.c file. The developer wrote the code of the two files, and the result to be 4.

TABLE 6. STATIC ANALYSIS RESULT FOR CODE IN FIGURE 6

Check	Function	Line	Detail
Overflow	CWE190_Integer_Overflow__char_max_square_65b_goodG2BSink(char)	42	Warning: operation [conversion from int32 to int8] on scalar may overflow (result strictly greater than MAX INT8)

Table 6 is the static analysis result for the code of Figure 6. As a result of analyzing the code flow of Figures 5 and 6, the result is 4. We can expect no integer overflow when running the static analysis tool. However, the overflow checker says that there is an integer overflow in line 42 in Table 6. It is a false positive in terms of reliability.

We did a detailed analysis to determine why the static analysis tool said an integer overflow exists in this line. The static analysis tool we used created a main function that directly calls Figure 6 and inserts random argument values. The range of values that the data variable can have is -128 to 127. So, the range of values that the result variable has is 0 to 16384. Therefore, Table 6 is an integer overflow caused by the lack of security.

```

38 void CWE190_Integer_Overflow__char_max_square_65b_goodG2BSink(char data)
39 {
40     /*
41      * POTENTIAL FLAW: if (data*data) > CHAR_MAX, this will overflow */
42     if(data > -12 && data < 12)
43     {
44         char result = data * data;
45         printHexCharLine(result);
46     }
47     else
48     {
49         printf("Integer Overflow");
50     }
51 }
52 }

```

Figure 7. Correct code for goodG2BSink function

Figure 7 is the correct code for removing the integer overflow from the code in Figure 6. Since the result of multiplying two data must be -128 or more and 127 or less, data must have a value greater than -12 and less than 12.

TABLE 7. STATIC ANALYSIS RESULT FOR CODE IN FIGURE 7

Check	Function	Line	Detail
Overflow	CWE190_Integer_Overflow_char_max_square_65b_goodG2BSink() 44		Operation [conversion from int32 to int8] on scalar does not overflow in INT8 range
Overflow	CWE190_Integer_Overflow_char_max_square_65b_goodG2BSink() 44		Operation [*] on scalar does not overflow in INT32 range

We performed a static analysis for the code in Figure 7 using a static analysis tool. As a result, we confirmed that Overflow in Table 6 changed from orange to green, which proved no weakness, as shown in Table 7.

IV. CONCLUSION

The importance of software is increasing in countries, societies, and lives. Several agencies have defined the concept of software assurance, which includes the definitions of reliability and security. Now, software engineers should not look at software quality and security assurance separately.

In this paper, we analyze the binary search program and CWE-190 test case example and performed the static analysis. As a result, we conclude that all integer overflow generated by static analysis tools are true positive. It solves the difficulties of distinguishing true positive and false positive, which are the limitations of static analysis. After performing static analysis, the developer should modify the code to satisfy reliability and security.

In future research, we will expand our study to analyze all Juliet Test Suite for C/C++. We will propose that the performance indicators for comparing static analysis tools should be changed by analyzing false positives in the static analysis results.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2018R1A2B6009122).

REFERENCES

- [1] S. Planning, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, 2002.
- [2] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bug," IEEE software, vol. 25, no. 5, pp. 22–29, 2008.
- [3] The MITRE Corporation, "Common Weakness Enumeration." <https://cwe.mitre.org/>. (accessed December 1, 2020).
- [4] The MITRE Corporation, "2020 CWE Top 25 Most Dangerous Software Weaknesses." https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html. (accessed December 1, 2020).
- [5] NIST, "Juliet." <https://samate.nist.gov/SRD/testsuite.php>. (accessed December 1, 2020).
- [6] E. Burtescu, "Reliability and Security - Convergence or Divergence," Informatica Economica, vol. 14, no. 4, pp. 68–77, 2010.
- [7] J. Viega, and G. McGraw, "Building Secure Software: How to Avoid Security Problems the Right Way," Addison-Wesley, pp. 15, 2011.

- [8] I. Sommerville, "Software Engineering. 10th," Book Software Engineering. 10th, Series Software Engineering, Addison-Wesley, 2015.
- [9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms," 3rd ed., MIT Press and McGraw-Hill, pp.39, 2009.



Hong Jun Choi received B.S. degree in the School of Computer Software, Daegu Catholic University, Korea in 2019. He is currently a M.S. candidate in School of Cybersecurity, Korea University. His current research interests are in static analysis, secure software engineering, formal methods.



Hyuk Lee received the B.S. degree from the University of Technology Sydney, Sydney, Australia, in 2006, and M.S. degree in 2009 and the Ph.D. degree in 2019 from the Korea University, Seoul, Korea. He is currently a research professor with the School of Cybersecurity, Korea University, Seoul, Korea. His current research interests are in formal methods, constraint problem solving, and secure software engineering.



Jin-Young Choi received the B.S. degree from Seoul National University, Seoul, Korea, in 1982, received the M.S. degree from Drexel University, Philadelphia, PA, in 1986, and the Ph.D. degree from the University of Pennsylvania, Philadelphia, in 1993. He is currently a Professor with the School of Cybersecurity, Korea University, Seoul, Korea. His current research interests are in real-time computing, formal methods, programming languages, process algebras, security, and secure software engineering.