

# Analysis of software weakness detection of CBMC based on CWE

Minjae Byun

Department of Information Security  
Korea university  
Seoul, Korea  
minjae\_byun@korea.ac.kr

Yongjun Lee

Department of Information Security  
Korea university  
Seoul, Korea  
yjlee@formal.korea.ac.kr

Jin-Young Choi

Department of Information Security  
Korea university  
Seoul, Korea  
choi@formal.korea.ac.kr

**Abstract**—Model checking is a method of verifying whether a target system satisfies a specific property using mathematical and logical proofs. Model checking tools to verify design (1) require a formal description of the design and (2) there can be discrepancies between the model and actual implementation. To solve these problems, various tools such as CBMC and BLAST that can directly input C codes have been proposed. However, in terms of security, it is difficult to figure out which software weaknesses these tools can verify. In this paper, we matched the properties that CBMC can verify with corresponding CWEs, considering interdependencies of CWEs. We also conducted an experiment using Juliet Test Suite to check CBMC can actually verify the codes including these CWEs.

**Index Terms**—CBMC, information security, model checking, software weakness

## I. INTRODUCTION

Model checking is a method of verifying whether a target system satisfies a specific property using mathematical and logical proofs. It can assure us that behaviors of the target system meet desirable properties as it looks at all possible behaviors of a system. Model checking also shows the great promise in security, especially detecting and eradicating security bugs or vulnerabilities [1]–[4].

Model checking tools to verify design [5]–[7] require a formal description of the design [8], so the effort and the time are required for developers to build the model. Also, there can be discrepancies between the model and actual implementation. To solve these problems, tools that can directly input C codes have been proposed [9]–[11]. They get a C program as an input and check whether it satisfies certain properties. CBMC, the tool used in this paper, is for the formal verification of ANSI-C programs using Bounded Model Checking. It has been used in various studies and some studies used it to find security bugs [12]–[14].

According to CBMC, it supports almost all ANSI-C language features and can verify buffer overflow, pointer safety, memory leaks, division by zero, not-a-number, and arithmetic overflow. This description is rough so the users can't know what kind of buffer overflow or pointer safety CBMC can verify. In this paper, we made the list of CWEs that can be included in those properties. It contains CWE-190, 416,

476 that are listed in the 2019 CWE Top 25 Most Dangerous Software Errors [15]. Then we conducted an experiment using Juliet Test Suite [16] to check CBMC can verify the codes including these CWEs.

The remainder of this paper is divided into five sections. Section II presents the background of this paper. Section III gives how we match the properties with CWEs and the full list of CWEs. Section IV shows the design of the experiment, execution, and analysis including the results. Finally, conclusions are provided in Section V.

## II. BACKGROUND

### A. CBMC

CBMC is a Bounded Model Checker for C and C++ programs developed by Carnegie Mellon university and has been developed and maintained for more than ten years [17]. It receives a C program and a property as an input and generates a corresponding CNF formula. The formula is passed to the SAT solver and if the formula is satisfied, CBMC generates a counterexample. Figure 1 shows the logic of CBMC [18].

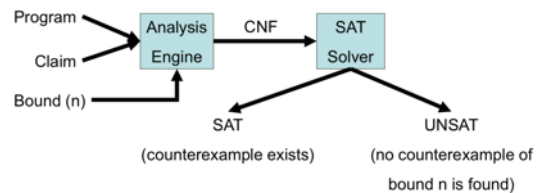


Fig. 1: The logic of CBMC

CBMC can verify the following properties [19]:

- Division by zero: Check whether there is a division by zero in the program.
- Not-a-Number: Check whether floating-point computation may result in NaNs.
- Pointer Safety: Search for NULL-pointer dereferences or dereferences of other invalid pointers.
- Arithmetic Overflow: Check whether a numerical overflow occurs during an arithmetic operation.
- Buffer Overflow: For each array access, check whether the upper and lower bounds are violated.

- Memory Leaks: Check whether the program constructs dynamically allocated data structures that are subsequently inaccessible.
- Undefined shifts: Check for shifts with excessive distance.

### B. Juliet Test Suite for C/C++

Juliet Test Suite was developed by the NSA's Center for Assured Software(CAS) specifically for use in testing static analysis tools [16]. Version 1.3 contains examples for 118 different Common Weakness Enumeration(CWE) with 64,099 test cases. Each CWE has several examples, some with weakness present and some with the weakness mitigated [20] and test cases are embedded in variations of different control flow and data flow patterns.

### III. THE LIST OF CWES

The properties that CBMC can verify are rough and they do not provide the list of CWES they can find. To get a better understanding of which software weaknesses they can verify, we matched the properties with CWES. Table 1 shows the list of CWES mapped with properties. CWE Version 3.4(released Sep 19, 2019) was used and our matching is done according to the following rules.

- 1) As CWE has the tree-like relationships between weaknesses, we referred to the CWE VIEW: Research Concepts [21]. When the properties were matched with a certain parent CWE, we also included possible child CWES.
- 2) Some properties like undefined shifts and float overflow were excepted as there is no corresponding CWE.
- 3) Some CWES like CWE-119, 120 and 825 were excepted after matching as there are no test cases in Juliet Test Suite.

### IV. EXPERIMENT

#### A. Dataset and Environment

**Dataset:** We used 14,668 test cases corresponding to 20 CWES above in Juliet Test Suite to test CBMC. We only use the test cases in the form of `*[0-9].c`, which means excluding Cpp and header files and also the test cases ending with an alphabet. We used `_bad` functions in each test case to test whether CBMC actually verifies codes with software weaknesses.

**Environment:** We used the latest version (5.11) of CBMC and the operating system is Ubuntu 18.04 LTS with 12 vCPUs and 32GB memory.

#### B. Execution of the experiment

CBMC provides various options to customize the behavior of CBMC [9]. We used the following parameters in this experiment depending on the properties to verify.

- Conversion Error: `--conversion-check`
- Not-a-Number: `--nan-check`
- Buffer Overflow: `--bounds-check --pointer-check --pointer-overflow-check`

- Pointer Safety (including Pointer Overflow): `--pointer-check --pointer-overflow-check`
- Memory Leaks: `--memory-leak-check --pointer-check --pointer-overflow-check`
- Division By Zero: `--div-by-zero-check`
- Arithmetic Overflow: `--signed-overflow-check --unsigned-overflow-check`

CBMC uses bounded model checking and the verification result may change depending on the number of unwinding. We started from 10 unwinds and proceeded with the experiment by increasing it gradually. Additional experiments were conducted on some CWES of buffer overflow after 100 unwinds.

#### C. Result

As we tested whether CBMC actually verifies codes with software weaknesses, we compare only recall per individual CWES in this paper. The recall is the ratio of the bad cases actually found in the entire bad case.

$$Recall = TP / (TP + FN)$$

TP means True Positive and FN means False Negative.

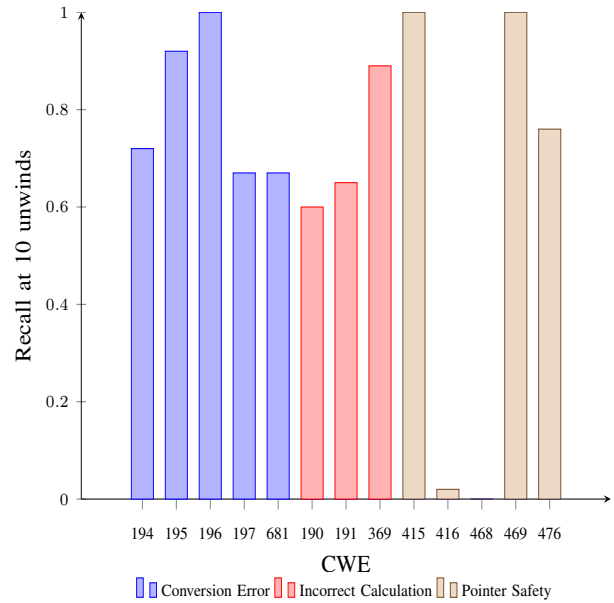


Fig. 2: Recall for CWES

Fig.2 illustrates the recall of CBMC per individual CWES verified within 10 unwinds. Incorrect Calculation includes Arithmetic Overflow and Division by Zero. The analysis of the false negative will be provided in section IV-D.

**Conversion Error and Not-a-Number:** “Unsigned to signed conversion errors”(CWE-196) were 100% verified. Among “incorrect conversion between numeric types”(CWE-681), Not-a-Number cases(CWE 681\_Incorrect\_Conversion\_Between\_Numeric\_Types\_doubleNaN2int\_\*) were all verifiable but double to float conversion cases(CWE

TABLE I: The list of CWEs

| Property                                    | CWE                                                                      | #  |
|---------------------------------------------|--------------------------------------------------------------------------|----|
| Conversion Error                            | CWE-194 Unexpected Sign Extension                                        | 1  |
|                                             | CWE-195 Signed to Unsigned Conversion Error                              | 2  |
|                                             | CWE-196 Unsigned to Signed Conversion Error                              | 3  |
|                                             | CWE-197 Numeric Truncation Error                                         | 4  |
| Conversion Error, Not-a-Number              | CWE-681 Incorrect Conversion between Numeric Types                       | 5  |
| Arithmetic Overflow                         | CWE-190 Integer Overflow or Wraparound*                                  | 6  |
|                                             | CWE-191 Integer Underflow (Wrap or Wraparound)                           | 7  |
| Division By Zero                            | CWE-369 Divide By Zero                                                   | 8  |
| Pointer Safety (including Pointer Overflow) | CWE-415 Double Free                                                      | 9  |
|                                             | CWE-416 Use After Free*                                                  | 10 |
|                                             | CWE-468 Incorrect Pointer Scaling                                        | 11 |
|                                             | CWE-469 Use of Pointer Subtraction to Determine Size                     | 12 |
|                                             | CWE-476 NULL Pointer Dereference*                                        | 13 |
| Buffer Overflow                             | CWE-121 Stack-based Buffer Overflow                                      | 14 |
|                                             | CWE-122 Heap-based Buffer Overflow                                       | 15 |
|                                             | CWE-124 Buffer Underwrite ('Buffer Underflow')                           | 16 |
|                                             | CWE-785 Path Manipulation Function Without Max Sized Buffer              | 17 |
| Memory Leaks                                | CWE-126 Buffer Over-read                                                 | 18 |
|                                             | CWE-127 Buffer Under-read                                                | 19 |
|                                             | CWE-401 Memory Leak (Missing Release of Memory after Effective Lifetime) | 20 |

\* marks those in the CWE/SANS top 25 list.

681\_Incorrect\_Conversion\_Between\_Numeric\_Types\_\_double2float\_\*) were not verified.

**Incorrect Calculation (Arithmetic Overflow and Division by Zero):** CBMC shows about 60% recall for both “integer overflow”(CWE-190) and “integer underflow”(CWE-191). It could verify “divide by zero”(CWE-369) except test cases using the `fscanf` library function.

**Pointer Safety:** CBMC verified all “double free”(CWE-415) and “use of pointer subtraction to determine size”(CWE-469) but showed near 0% recall for “use after free”(CWE-416). “Incorrect pointer scaling”(CWE-468) couldn’t be verified at all.

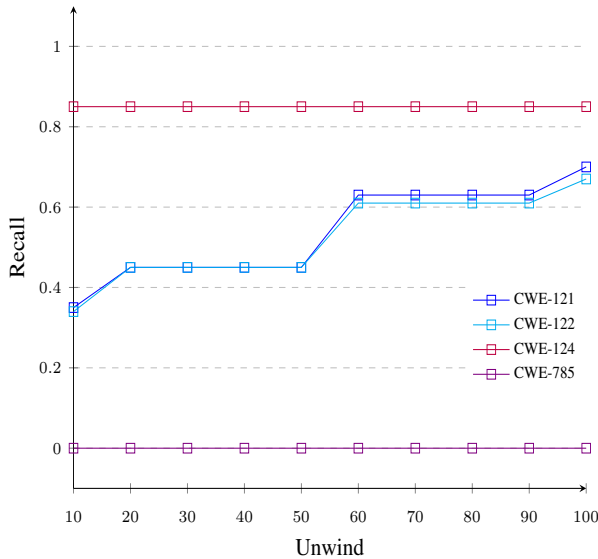


Fig. 3: Recall for Buffer Overflow Weaknesses

Fig.3 and Fig.4 show the recall of CBMC for Buffer Overflow and Memory Leaks from 10 unwinds to 100 unwinds.

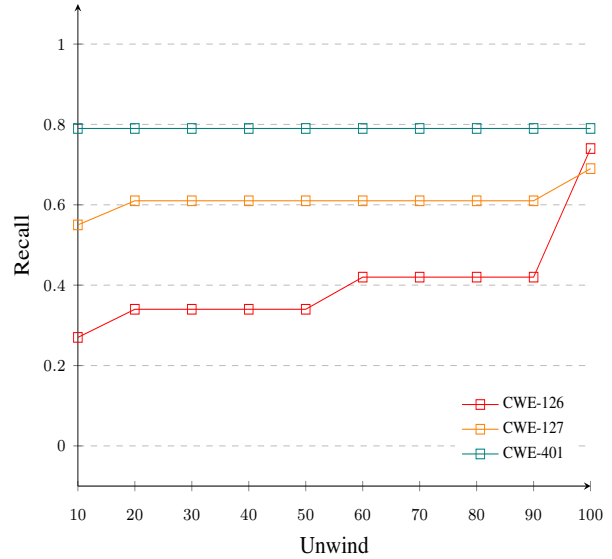


Fig. 4: Recall for Memory Leaks Weaknesses

**Buffer Overflow:** CBMC verified 85% of “Buffer Underwrite”(CWE-124) but couldn’t verify “Path manipulation function without max sized buffer”(CWE-785) at all. “Stack-based buffer overflow”(CWE-121) and “Heap-based buffer overflow”(CWE-122) were not sufficiently verified within 100 unwinds and additional tests were done with more unwinds. CBMC showed 75% recall for CWE-121 and 72% recall for CWE-122 from 200 to 500 unwinds.

**Memory Leaks:** 79% of “Memory Leaks”(CWE-401) could be verified. Not freeing memories when `realloc` fails(CWE401\_Memory\_Leak\_\_malloc\_realloc\_\*) were not verified. CBMC could verify “Buffer Over-read”(CWE-126) for 74% and “Buffer Under-read”(CWE-127) for 69% at 100 unwinds.

#### D. Analysis of the result

CBMC could successfully verify all test cases of CWE-196, 415, and 469. The average recall for each property is as follows. The result of CWE-785 was excepted from the average as it couldn't verify for environmental reasons. The average value was rounded to two decimal places.

- Conversion Error: 78%
- Arithmetic Overflow: 62%
- Division by Zero: 89%
- Pointer Safety: 63%
- Buffer Overflow: 76%
- Memory Leaks: 73%
- **Average: 72%**

Some test cases weren't verified for the following reasons.

**Using the Windows header file:** "Path manipulation function without max sized buffer"(CWE-785) test cases include `windows.h`. `Windows.h` is a Windows-specific header file which includes declarations for both Unicode and ANSI versions of the Windows API [22]. The experiment was done using the Linux operating system so it couldn't be verified.

**Array too large for flattening:** Some test cases exceeded the limit of the solver so they couldn't be verified with the message "array too large for flattening".

**Standard input library functions:** "Divide by Zero"(CWE-369) reported different results for `int` and `float` test cases using the `fscanf` function.

Listing 1: CWE369\_Divide\_by\_Zero\_float\_fscanf\_01

```
...
float data;
data = 0.0F;
fscanf (stdin, "%f", &data);
{
    int result = (int)(100.0 / data);
    printIntLine(result);
}
...
```

Listing 2: CWE369\_Divide\_by\_Zero\_int\_fscanf\_divide\_01

```
...
int data;
data = -1;
fscanf(stdin, "%d", &data);
printIntLine(100 / data);
...
```

Test cases for `float` (Listing 1) initiates `data` (the divisor) as 0 and CBMC could find the possibility of division by zero. Test cases for `int` (Listing 2) initiates `data` as -1 and CBMC couldn't detect division by zero.

And the test cases using conversion functions like `atof` (CWE681\_Incorrect\_Conversion\_Between\_Numeric\_Types\_double2float\_\*) couldn't be verified as CBMC couldn't interpret the meaning of the functions.

#### V. CONCLUSION

This paper is focused on the evaluation of the ability of CBMC to verify software weaknesses. For this purpose,

we chose 20 CWEs and conducted an experiment using 14,668 test cases of Juliet Test Suite. Our experimental results showed that CBMC verified "Unsigned to Signed Conversion Error"(CWE-196), "Double Free"(CWE-415), and "Use of Pointer Subtraction to Determine Size"(CWE-469) for 100% recall. The average recall for 19 CWEs except "Path manipulation function without max sized buffer"(CWE-785) which uses the Windows header file is 72%. In the future work, we will do additional preprocessing on test cases to handle the problems mentioned in IV-D to improve the recall of CBMC.

#### REFERENCES

- [1] J. Li and J. Li, "Model checking security vulnerabilities in software design," in *2010 6th International Conference on Wireless Communications Networking and Mobile Computing (WiCOM)*, pp. 1–4, IEEE, 2010.
- [2] B. Schwarz, H. Chen, D. Wagner, G. Morrison, J. West, J. Lin, and W. Tu, "Model checking an entire linux distribution for security violations," in *21st Annual Computer Security Applications Conference (ACSAC'05)*, pp. 10–pp, IEEE, 2005.
- [3] R. Hadjidi, X. Yang, S. Tlili, and M. Debbabi, "Model-checking for software vulnerabilities detection with multi-language support," in *2008 sixth annual conference on privacy, security and trust*, pp. 133–142, IEEE, 2008.
- [4] R. W. Ritchey and P. Ammann, "Using model checking to analyze network vulnerabilities," in *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pp. 156–165, IEEE, 2000.
- [5] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.
- [6] W. Yi, P. Pettersson, and M. Daniels, "Automatic verification of real-time communicating systems by constraint-solving," in *Formal Description Techniques VII*, pp. 243–258, Springer, 1995.
- [7] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.
- [8] R. C. Armstrong, R. J. Punnoose, M. H. Wong, and J. R. Mayo, "Survey of existing tools for formal verification," *SANDIA REPORT SAND2014-20533*, 2014.
- [9] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168–176, Springer, 2004.
- [10] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker b last," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.
- [11] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *International Conference on Computer Aided Verification*, pp. 184–190, Springer, 2011.
- [12] R. Wojtczuk, "Uqbtng: A tool capable of automatically finding integer overflows in win32 binaries," in *22nd Chaos Communication Congress*, pp. 168–176, 2005.
- [13] F. Dörre and V. Klebanov, "Practical detection of entropy loss in pseudo-random number generators," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 678–689, ACM, 2016.
- [14] S. Chattopadhyay and A. Roychoudhury, "Symbolic verification of cache side-channel freedom," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2812–2823, 2018.
- [15] The MITRE Corporation, "Cwe view: Weaknesses in the 2019 cwe top 25 most dangerous software errors." <https://cwe.mitre.org/data/definitions/1200.html>. (accessed October 9, 2019).
- [16] NIST, "Juliet." <https://samate.nist.gov/SRD/testsuite.php>. (accessed October 9, 2019).
- [17] D. Kroening and M. Tautschnig, "Cbmcc-c bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 389–391, Springer, 2014.
- [18] Software Engineering Institute Carnegie Mellon University, "Introduction to cbmc." [http://www.cs.cmu.edu/emc/15414-f11/lecture/lec26\\_CBMC.pdf](http://www.cs.cmu.edu/emc/15414-f11/lecture/lec26_CBMC.pdf). (accessed October 9, 2019).

- [19] D. Kroening, "Cprover manual." <http://www.cprover.org/cprover-manual/properties/>. (accessed October 9, 2019).
- [20] A. Wagner and J. Sametinger, "Using the juliet test suite to compare static security scanners," in *2014 11th International Conference on Security and Cryptography (SECRYPT)*, pp. 1–9, IEEE, 2014.
- [21] The MITRE Corporation, "Cwe view: Research concepts." <https://cwe.mitre.org/data/definitions/1000.html>. (accessed October 9, 2019).
- [22] Microsoft, "Using the windows headers." <https://docs.microsoft.com/en-us/windows/win32/winprog/using-the-windows-headers>. (accessed October 9, 2019).

**Minjae Byun** received the two B.E. degrees in computer science and engineering and information security from Korea University, Seoul, South Korea, in 2018. She is currently pursuing the M.E. degree in information security at Graduate School of Information Security, Korea University, Seoul, South Korea. Her current research interests are in secure software engineering, formal methods, and cryptography.

**Yongjun Lee** received the two B.S degrees in Department of computer and information security and optical engineering from Sejong University, Seoul, South Korea, in 2018. He is currently pursuing the M.E. degree in information security at Graduate School of Information Security, Korea University, Seoul, South Korea. His current research interests are in formal methods, security vulnerability assessment, deep learning, and secure software engineering.

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