International Journal of Security and Its Applications
Vol.8, No.1 (2014), pp.333-342
http://dx.doi.org/10.14257/ijsia.2014.8.1.31

A Study on the Optimization Method for the Rule Checker in the Secure Coding

JaeHyun Kim[1] and YangSun Lee[1]*

[1]*Dept. of Computer Engineering, Seokyeong University*
16-1 Jungneung-Dong, Sungbuk-Gu, Seoul 136-704, KOREA
statsr@skuniv.ac.kr,
**Corresponding Author: yslee@skuniv.ac.kr*

Abstract

Today's software allows data transfer with the use of internet. Therefore, there is always a threat of attack by hackers. These security weaknesses cause a critical economic loss which is a direct cause of software security invasion accidents. Recently in order to solve these security weaknesses, rather than strengthening the security system from the external environment, many have started to realize it is essential and most efficient for programmers to develop stronger software. Internationally, resolving software weakness from the coding stage to prevent security incidents by providing a coding guide is rising as a security issue. Especially, user demands of software are becoming enormous and complicated. In order to reduce weaknesses that could lie in the software have to be removed and the costs for these increases as the development process progresses. This leads to issues nowadays with removing the security weaknesses from the coding stage. This technique is called secure coding and not only is the academic and the industrial world showing interest in this technique, but also national agencies are showing great interest. Especially in Korea, the electronic government business has decided to introduce secure coding and all developed programs will apply the security coding methodology. Rule checker, the object of study of this research, is a core tool for secure coding which is used to analyze security weaknesses existing in programs using a rule base. Especially, it can be used in the developmental stage and examination stage which makes an efficient composition of rule checker very important. In this research, a maximized technique to compose a rule checker with most efficiency has been proposed.

Keywords: *Secure Coding, Rule Checker, DFA Optimization, Optimizer Generator*

1. Introduction

Security incidents which are becoming a hot issue socially not only cause enormous economic losses but also lead to harm by leaking one's private information. These security incidents are mostly caused by vulnerability in most software. Especially, today's programs transfer data through the internet which makes it difficult to secure trust of the data being input and output [1, 2].

Due to these reasons, it has become a trend to provide coding guide in order to reduce software weakness from the coding stage. Thus, if vulnerability of software weaknesses can be blocked from the development stage, the huge costs for efforts to modify the vulnerability recognized in the operation stage can be reduced and it can largely contribute to developing software safe from hackers.

ISSN: 1738-9976 IJSIA

CWE(Common Weakness Enumeration)[3], CERT(Computer Emergency Response Team) [4] and other institutes in foreign countries have drawn a list of vulnerabilities and have started research to provide a guide for secure coding. Also, domestic and foreign software development companies are putting in efforts to develop high quality software through the help of internal coding guides.

Rule checker, the object of study of this research, is a core tool for secure coding which is used to analyze security weaknesses existing in programs using a rule base. Especially, it can be used in the developmental stage and examination stage which makes an efficient composition of rule checker very important. In this research, a maximized technique to compose a rule checker with most efficiency has been proposed.

Rule checker is a tool which uses a rule base composed by the security program expert to inspect vulnerabilities existing in a given program. Therefore, the number of vulnerabilities inspected by the rule checker depends on the number of rules. The larger the number of rules, the number of vulnerabilities may be found. However, many rules decrease the performance of the execution speed of the rule checker. This may be a critical problem to the objective of the rule checker as fast feedback needs to be pro-vided the programmer in the developmental stage.

In this research, a technique to enhance the analysis performance when the rule checker uses multiple rules will be researched.

The contents of the thesis are as follows. First, in Chapter 2, secure coding and vulnerability analysis tool based on coding rule will be examined. Next, in Chapter 3, the technique proposed in this thesis will be introduced through rule optimization model and algorithms. In Chapter 4, the results after applying the proposed algorithm will be analyzed and evaluated. Lastly in Chapter 5, the conclusion and future direction of research will be discussed.

2. Related Studies

2.1. Secure Coding

The paradigm of programming started from sequential programming to structured programming developing to object orientated programming. With this, the philosophy of programming has developed from a precise result being drawn from a précised entered value by precise programming to a highly trusted programming strong from external factors [5]. Recently, most application programs work in a network connected environment. As a result, a lot of effort is being put in to prevent security incidents. Amongst these, the importance of secure coding which writes safe programs from hackers from the developmental stage has started to stand out.

Even until now, security systems to prevent software from security incidents mostly include network firewall, user authentication systems but according to Gartner's report, 75 percent of software security incidents are caused by application programs which contain vulnerability. Also, after development is completed, the safety of the program has to be considered from the developmental stage as the costs to make up the vulnerability are very high. Therefore, rather than strengthening the security system from external environment, programmers developing a strong software is the most essential and efficient method to higher the security level.

Recently, these problems have been recognized and researches on secure coding which allows secure code to be written to prevent hackers from the developmental stage are actively in progress. Especially, CWE which analyzes vulnerabilities which can occur in programming language have analyzed and specified various vulnerabilities that may occur in the stage of

writing source code per language. Also, CERT has defined secure coding to write safe source codes. Cigital has classified 61 regulations to remove vulnerabilities which have been classified by Seven Pernicious Kingdoms [6] proposed by Katrina Tsipenyuk, Brian Chess and Gary McGraw. The coding regulations proposed by Cigital have been defined in XML format and these can be entered and used by programs such as weakness analysis programs. In the car and aviation industry where a flaw in the software can cause a fatal problem, JSF(Joint Strike Fighter), MISRA Coding rule and other coding regulations have been introduced already and efforts are being put in continuously to develop high quality software.

2.2. Coding Rule Based Weakness Analysis Tool

Secure coding rule is also used in tools to analyze vulnerability and each tool uses a rule specialized for its own analysis method. In this research, in order to optimize weakness inspection rules used in rule checkers, five weakness analysis tools HP Fortify SCA, Coverity Prevent, Findbugs, PMD and Jlint were investigated.

HP Fortify SCA [7] is vulnerability detecting tools developed by Fortify. HP Fortify SCA supports 12 languages including C/C++, Java. It uses static analysis and dynamic analysis methods to detect vulnerability in source code. The vulnerability in-formation found in provided to the user along with statistics.

Coverity's Coverity Prevent [8] is a source code static analysis tool. Coverity Prevent lists up vulnerabilities found in the total code. Each list includes location within the source code where the vulnerability was found, cause of the vulnerability and etc.

Findbugs [9] was developed as a static analysis tool in Maryland University which works in JRE and can also be used in Eclipse plug-in development environments. It is a tool which can also analyze Java programs which is open source software which can be used freely in GNU LGPL. FindBugs analyzes Java programs at a byte code level using bug patterns created in advance which gives it a characteristic of not requiring the source. Bug patterns can be expanded which allows relevant fields to be easily inspected when weaknesses are added. Currently, there are 401 basic bug patterns in FindBugs.

PMD [10] is a weakness analysis tool progressed as an open project. It can be used as a plugin in various development tools such as Eclipse, JBuilder, Netbeans and others. Like Findbugs, it is a tool which can analyze Java programs. In the recently improved version, it can analyze jps, xsl, ecmascript and xml including Java and it is the open software with BSD license. PMD uses a predefined rule to analyze programs on a source code base inspecting possible bugs, dead code, suboptimal code, over-complicated expressions, and duplicate code. These rules can be expanded like Findbugs. Rules may be defined using Java class files of Xpath. Currently, PMD has 271 basic rules.

Jlint [11] was developed by Konstantin Knizhnik and is a weakness analysis tool currently worked as an open project. It is optimized to inspect C style programming language syntax and problems with synchronization of Java programs. It has 33 inspection fields.

3. Rule Optimization Method

3.1. Rule Optimizer Model

The rule optimizer is a tool which helps the rule checker for secure coding to inspect weaknesses more efficiently by optimizing the rules subject to inspection. Generally, the

weaknesses subject to inspection by the rule checker are made by fields. As the number of inspection field increases, the number of rules increases as well. When the number of rules increases, scanning and analysis of the target code for inspection increase as well. Even if the number of rules increase, a technique to reduce the time to analyze the source code is very important. Especially, as the size of source for inspection increases, the time required drastically increases as well.

In this research, these rules were analyzed in advance and optimization patterns were made. Using this, an optimized rule to analyze the total set of rules effectively is proposed. The rule optimizer receives an optimized rule pattern and uses this to create a DFA (Deterministic Finite Automata). Each created DFA is merged into one DFA which is normalized once more into a form suitable for optimization. The normalized DFA is converted into a pattern table with actions which will be used in the optimizer recorded by the table creator. Figure 1 shows a model of the rule optimizer creator.

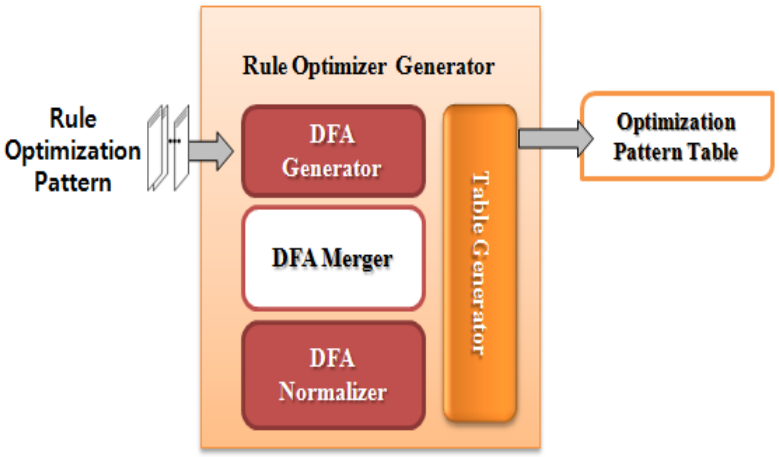


Figure 1. Rule Optimizer Generator Model

The rule optimizer uses the optimized table pattern to convert the received rule set entered into an optimized DFA form. The optimized DFA uses the rule checker and is able to analyze more effectively compared to using the rule set directly. Figure 2 is a rule optimized model using a created optimized pattern table.

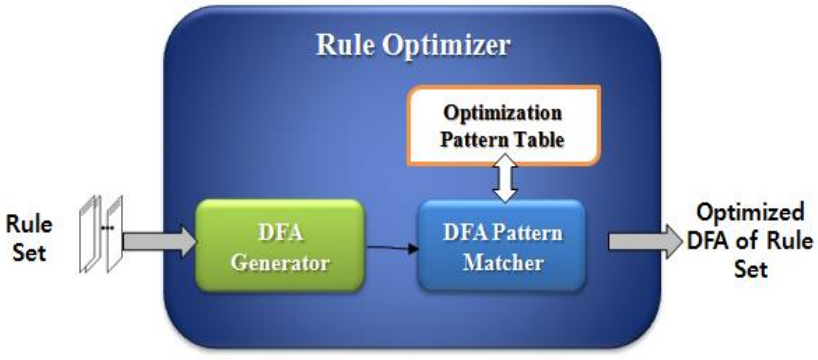


Figure 2. Rule Optimizer Model

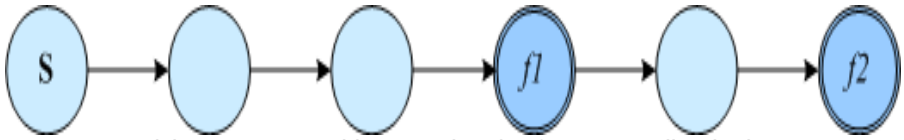
3.2. Rule Optimization Algorithms

Algorithms for rule optimization are divided mainly into algorithms which create optimized DFA information and algorithms to convert rule sets using created information into DFA. Table 1 introduces symbols used in algorithms.

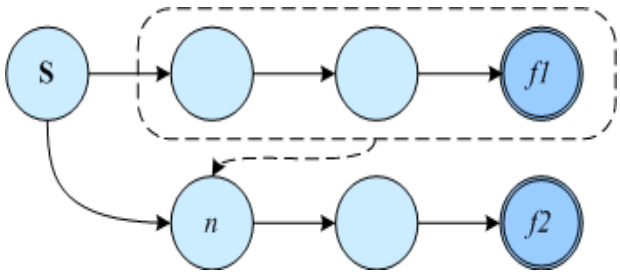
Table 1. Symbol Notations for Proposed Algorithm

C_0
$LR(0)_{set}$: *set of LR(0) items*
I : *set of* $LR(0)_{set}$*s*
I_i : i_{th} *element of* $LR(0)_{set}$
Lookahead
$LA_{set(i)}$: *lookahead set of* I_i
State (S)
S_{Set} : *set of states*
S_S : *start state*
S_C : *current state*
S_F : *final state*
Rule States
$Rule_{Set}$: *set of rule states*
Optimization Table
$OPT[S, R \in Rule_{Set}]$: *optimization table*

In order to create optimized DFA, DFA for rules recorded with optimized patterns must be created and these must be merged to create a combined DFA. In this process, as shown in Figure 3, even though it is not in a terminal state due to merging of DFA, there are cases where an end state is created. Therefore, these DFA always need to be normalized in order to change end states to terminal states.



(a) DFA created by merging (Before normalization)



(b) DFA has two final states independently after normalization

Figure 3. Example of DFA Normalization

Table 2 and Table 3 each are DFA create-merge algorithm and normalization algorithm for merged DFA.

Table 2. DFA Create-Merge Algorithm

```
Algorithm create_merge_dfa;
begin
    make S_S
    stateCounter ← 0
    S_C ← S_S
    for all I_i ∈ I
        S_Set ← Φ
        if I_i have reduce item and I_i have kernel item(s) then
            S_Set ∪ normalize_dfa(I_i)
            I ∪ calculation_C0 (I_i)
        fi
        S_Set ∪ S_C
        if LA_set(i) = Φ then
            set_final_state(S_C)
        else
            for all S_C ∈ S_Set
                for all LA ∈ LA_set(i)
                    stateCounter ← stateCounter + 1
                    make_state(S_C+ stateCounter)
                    set_child(S_C, S_C+ stateCounter, LA)
                end for
            end for
        fi
        S_C ← S_i + 1
    end for
end.
```

Table 3. DFA Normalization Algorithm

```
Algorithm normalize_dfa(LR(0)_set L);
  begin
    for all kernel item LA ∈ L
        stateCounter ← stateCounter + 1
        make_ state(S_C+ stateCounter)
        S_Set ∪ S_C+ stateCounter
        set_child(S_S, S_C+ stateCounter, replace item)
    end for
    return S_Set
  end.
```

When DFA is created for optimized rule pattern, this information is used to create the pattern table. In the pattern table, information of three actions exists. Shift for partial matching optimization, reduce for applying of optimization rule by total matching and advance for searching next patterns when there is no match found. Table 4 shows algorithms to record information of each action.

Table 4. Optimization Pattern Table Generation Algorithm

```
Algorithm generate_ table;
begin
    for all state Si ∈ S
        for all R ∈ Rule_Set
            if LA_set(i) ∩ R then
                if get_ child_state(Si, R) ∈ SF then
                    OPT[Si, R] ← get_no_final_state(Si, R)
                else
                    OPT[Si, R] ← get_no_ child_state(Si,R)
                fi
            else
                OPT[Si, R] ← 0
            fi
        end for
    end for
end.
```

The generated table information is used by the rule optimizer and Table 5 shows the optimized algorithm.

Table 5. Pattern Matching Algorithm for Optimization

```
Algorithm pattern_matching;
begin
    SC ← SS
    IP ← IC ← I0
    for IP ∈ I
        if OPT[SC, IC] ∈ SF then
            SC ← SS
            IP ← IC ← IRi
        else if OPT[SC, IC] ∈ S then
            SC ← DFAT[SC, IC]
            IC ← IC+1
        else
            SC ← SS
            IP ← IC ← IP+1
        fi
    end for
end.
```

4. Experimental Results

In this chapter, the experiment carried out using the proposed optimization technique and analysis results are introduced. First, the inspection rule grammar used in this thesis is as shown in Table 6. It is written in EBNF and the codes corresponding to the codes are judged to be codes with weaknesses.

Next, example is a weakness inspection rule composed using the rule grammar. The rule checker basically converts the inspection rule to DFA and then processes. If there are same inspection components while composing the rules, each rule is merged into a DFA to increase the efficiency. For example, in rules 1, 2 and 4 [DEF("fid")] and qualified-by SYNCHRONIZED in EXCLUDE can be merged.

Table 6. Rule Grammar Example

```
specification ::= { groupdef } { rule }
groupdef ::= GROUP #name = '{' { funcall } '}'
rule ::= header { constraint }
constraint ::= UNSAFE pattern { EXCLUDE pattern }
pattern ::= scopepat { arrow sort scopepat }
```

Next, example is a weakness inspection rule composed using the rule grammar. The rule checker basically converts the inspection rule to DFA and then processes. If there are same inspection components while composing the rules, each rule is merged into a DFA to increase the efficiency. For example, in rules 1, 2 and 4 [DEF("fid")] and qualified-by SYNCHRONIZED in EXCLUDE can be merged.

Table 7. Weakness Checking Rules

1	RULEID 367-java-001 RULENAME "Time-of-check Time-of-use (TOCTOU) Race Condition"
	UNSAFE [#Ext()] --> if [$<FIELD>] in-funtion "fid" in-subclass-of Thread EXCLUDE [$<FIELD>] in SYNCHRONIZED EXCLUDE [DEF("fid")] qualified-by SYNCHRONIZED
2	RULEID 367-java-002 RULENAME "Time-of-check Time-of-use (TOCTOU) Race Condition"
	UNSAFE [#Ext()] --> if [FIELD($$)] in-funtion "fid" in-subclass-of Runnable EXCLUDE [FIELD($$)] in SYNCHRONIZED EXCLUDE [DEF("fid")] qualified-by SYNCHRONIZED
3	RULEID 383-java-001 RULENAME "J2EE Bad Practices： Direct Use of Threads"
	UNSAFE [$<java.lang.Thread>]
4	RULEID 609-java-001 RULENAME "Double-Checked Locking"
	UNSAFE [$1 := new java.lang.Object.Object(...)] in CONDITION in SYNCHRONIZED in CONDITION in-funtion "fid" EXCLUDE [DEF("fid")] qualified-by SYNCHRONIZED
5	RULEID 9402-java-001 RULENAME "J2EE Bad Practices: Non-Serializable Object Stored in Session"
	UNSAFE [Session.addAttribute(...)] EXCLUDE [Session.addAttribute($$, $<java.io.Serializable>)]
6	RULEID 9404-java-001 RULENAME "Race Condition: Singleton Member Field"
	UNSAFE[$<FIELD>] <-- if [HttpServletRequest.getParameter(...)]
7	RULEID 9405-java-001 RULENAME "Race Condition: Static Database Connection(dbconn)"
	UNSAFE [FIELD$<java.sql.Connection>] qualified-by STATIC

The performance of Java based open source code programs (JFreeChart, Trove, Heritrix, JMoney) before and after applying the optimization were evaluated to check the performance of the optimized rule checker. Figure 4 shows a reduction of overall time taken for inspection compared to time taken before optimization was applied. An increase of 17~32% in

performance can be seen. Figure 4 shows reduction rate of inspection time before and after applying optimization.

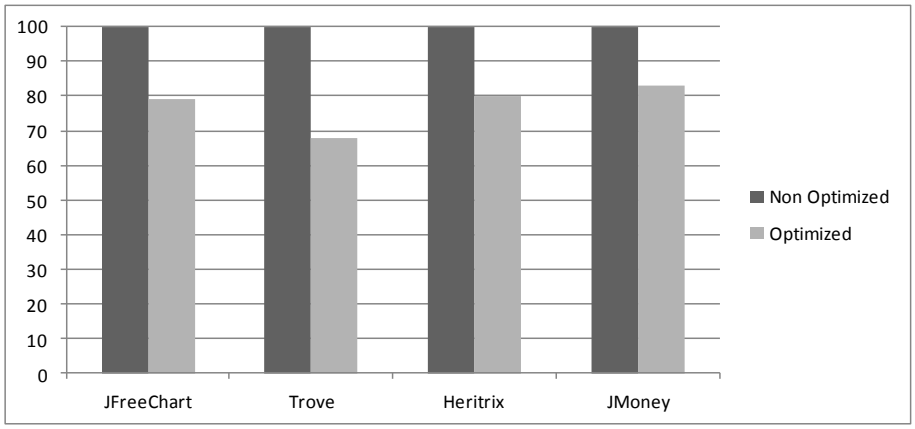


Figure 4. Weakness Checking Time Reduction Rates

5. Conclusions and Further Researches

The proposed research is a technique to enhance the performance of the rule checker which is a tool of secure coding used to develop safe programs. As the number of rules in increased in the rule checker, the fields inspected in the program will increase allowing safer development of the program. However, the numerous ruler checkers decrease the execution speed and if the program source becomes larger and larger, these problems become a cause to an increase in development expenses. Therefore, by using the optimization technique proposed in this research, a more efficient rule checker can be made resolving these problems.

Weakness analyzers generally require higher analysis costs as the number of analysis fields and degree of precision are increased. Therefore maintaining balance from a financial and engineering perspective is an important problem. Through this research, a method to resolve this problem can be proposed and it has a large academic meaning. Also industrially, it is possible to make an efficient analysis tool it can be used as various analysis tools for program security environments which are being enforced recently.

In the future, research on rule optimization patterns to enhance the performance and increase the precision of the rule checker will be carried out.

Acknowledgements

This Research was supported by Seokyeong University in 2012.

References

[1] G. McGraw, Software Security: Building Security In, Addison-Wesley, **(2006)**.
[2] J. Viega and G. MaGraw, “Software Security: How to Avoid Security Problems the Right Way”, Addison-Wesley, **(2006)**.
[3] Common Weakness Enumeration (CWE): A Community-Developed Dictionary of Software Weakness Types, http://cwe.mitre.org/.
[4] J. McManus and D. Mohindra, “The CERT Sun Microsystems Secure Coding Standard for Java”, CERT, **(2009)**.

[5] Y. Son, Y. Lee and S. Oh, "Design and Implementation of the Compiler with Secure Coding Rules for Developing Secure Mobile Applications in Memory Usages", International Journal of Smart Home, vol. 6, no. 153, **(2012)**.
[6] K. Tsipenyuk, B. Chess and G. McGraw, "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors", Security & Privacy, IEEE, vol. 81, **(2005)**.
[7] Fortify SCA, https://www.fortify.com/products/hpfssc/.
[8] Coverity, Inc., Coverity Static Analysis, http://www.coverity.com/products/ static-analysis.html.
[9] FindBugs, http://findbugs.sourceforge.net/.
[10] PMD, http://pmd.sourceforge.net/pmd-5.0.0/.
[11] Jlint, http://jlint.sourceforge.net/.

Authors

JaeHyun Kim, he received the B.S. degree from the Dept. of Mathematics, Hanyang University, Seoul, Korea, in 1986, and M.S. and Ph.D. degrees from Dept. of Statistics, Dongguk University, Seoul, Korea in 1989 and 1996, respectively. He was a chairman of Dept. of Internet Information 2002-2007. Currently, he is a member of the Korean Data & Information Science Society and a Professor of Dept. of Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include mobile programming, cloud system and data analysis.

YangSun Lee, he received the B.S. degree from the Dept. of Computer Science, Dongguk University, Seoul, Korea, in 1985, and M.S. and Ph.D. degrees from Dept. of Computer Engineering, Dongguk University, Seoul, Korea in 1987 and 2003, respectively. He was a Manager of the Computer Center, Seokyeong University from 1996-2000, a Director of Korea Multimedia Society from 2004-2005, a General Director of Korea Multimedia Society from 2005-2006 and a Vice President of Korea Multimedia Society in 2009. Also, he was a Director of Korea Information Processing Society from 2006-2010 and a President of a Society for the Study of Game at Korea Information Processing Society from 2006-2010. And, he was a Director of Smart Developer Association from 2011-2012. Currently, he is a Professor of Dept. of Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include smart system solutions, programming languages, and embedded systems.

