A Theoretical Framework for Software Vulnerability Detection based on Cascaded Refinement Network

Richard Amankwah
Presbyterian College of Education
P. O Box 27 Akropong-Akuapem
Ghana

Patrick Kwaku Kudjo
Sch. of Comp. Sci.
Datalink Institute
P.O. Box CO2481

Beatrice Korkor Agyemang
Presbyterian College of Education
P. O Box 27
Akropong-Akuapem
Ghana

Kofi Mensah
Presbyterian College of Education
P. O Box 27 Akropong-Akuapem
Ghana

Bright Brew
Presbyterian College of Education
P. O Box 27 Akropong-Akuapem
Ghana

Samuel Yeboah Antwi
Presbyterian College of Education
P. O Box 27 Akropong-Akuapem
Ghana

ABSTRACT
Software vulnerability detection is an active area of research in the software engineering domain. This is partly due to the continuous disclosure of security vulnerabilities. Although previous studies demonstrate the usefulness of employing several detection techniques, models, tools in detecting software vulnerabilities, the improvement of effectiveness of these detection models and tools is still a major challenge to researchers and practitioners. Cascaded Refinement Network (CRN) is novel model that has been successfully applied in several domains of studies such as image analysis, however its application to the field of vulnerability analysis has not been investigated. Motivated by the model effectiveness in these fields of studies, we investigate its feasibility within the domain of vulnerability detection using a theoretical framework. The analysis involves first presenting a general overview of the static analysis tools, and then an overview of the theoretical framework for vulnerability detection based on the CRN. The preliminary findings show that the concept is feasible within the domain of vulnerability detection.

General Terms
Software Engineering, Information Security

Keywords
Software Vulnerability; Static Analysis; Cascaded Refinement Network

1. INTRODUCTION
The existence of vulnerabilities in software products are catalyst for attack. Although there is no universal definition for software vulnerability, previous studies have given varied explanation of the concept. Kanga et al. [1] defined software vulnerability as the fault that can be viciously cause damage to software systems. In another study, Krsul [2] describe software vulnerability as defects in software systems that allows an attacker to violate an explicit or implicit security policy to achieve some impact. Jimenez et al. [3] as well defined software vulnerability as a flaw, weakness and errors in software systems that can be exploited by an attacker in order to alter the normal behavior of the system. The aforementioned definitions clearly show that software errors are the main causes of information security breaches. It is worth noting that if these vulnerabilities are not detected and corrected it creates an avenue for attackers to exploit that weakness and break into the software product, hence the need to investigate the various strategies and techniques that can be used in detecting and fixing these weaknesses. Recently, several models, techniques and tools have been proposed to find such weaknesses, the most widely applied tool are the static analysis tools. The static analysis involves analyzing the source code of a program without executing the actual programs, thus avoiding the risk associated with the execution of the malicious programs [4]. According to Black and Fong [5] static analysis techniques are software security assurance tools that detect flaws at various stage of the software development life cycle. Additionally, the static analysis techniques and tools are very effective in bug identification because of its rapidity, simplicity [6]. Generally, the static analysis tools detect security vulnerabilities by scanning the program source code. It is important to reiterate here that, researchers and practitioners often expend more efforts to detect and analysis static vulnerabilities in software application written in high-level language, such as C, C++, C#, Java, or PHP because it often involves the analysis of several hundreds of source codes. This makes the detection of vulnerability in source code a very difficult task.

Hence the need to investigate other alternative techniques and tools that can effectively be applied for improved vulnerability detection. Although researchers have used static analysis tools to detect a lot of loopholes in software in recent years and published them in major databases [7],[8],[9], challenges still exist in relation to its effectiveness and efficiency. In this study, we investigate the feasibility of apply the cascaded refinement network for improved vulnerability detection. Cascaded Refinement Network is a semantic label map that produces an image with photographic appearance that conforms to an input layout. We chose the Cascaded Refinement Network because a combination of these methods have achieved state-of-the-art performance in other areas [10],[11]. The proposed method would (1) enable developers, users and all stakeholders to pay attention to the severe weakness and deal with it (2) resolve the problem of false positive associated with static analysis tools (3) reduce cost associated with bug management.

The study makes the following contributions:

i. We present a general overview of the static analysis tools and methods

ii. We present a theoretical framework for software vulnerability detection method based on cascaded refinement network

The remaining sections of the paper are structured as follows. Section 2 presents a review of the static analysis. Section 3 presents a detailed overview of the static analysis tools and

methods. The theoretical framework based on the cascaded refinement network is presented in section 4. Section 5 summarizes the study and provides future research directions

2. STATIC ANALYSIS

The static analysis technology has grown from early lexical analysis to formal verification method and its detection capability has now improved a lot. But with the effort of researcher, static analysis tools have become more and more powerful. The most widely used static analysis techniques are lexical analysis, type inference, theorem proving, data flow analysis, model checking and symbolic execution. We briefly describe these techniques below:

2.1 Lexical analysis

Lexical analysis is a grammar structure analysis, similar to the C compiler. The analysis involves dividing the program into several fragments and analysis the lines of codes of such program to detect if there are flaws or loopholes in the syntax, semantics subroutines of the program. Failure to consider the aforementioned variables can result in high false positive.

2.2 Type Inference

Type inference is the process of inferring the type variables and functions of the compiler and judging whether its access of variables and functions are in accordance with the type rules. Programming language system includes a mechanism for defining the data types and rules

2.3 . Data Flow Analysis

Data flow analysis involves collecting semantic information from the program code, and with algebraic method to determine the definition and usage of the variables at compiling time. By using the control flow graph data flow analysis determines whether a value in the program is assigned to the possible vulnerability.

2.4 Rule Checking

Rule checking involves analyzing the security of the program by using pre-established safety rules. There are some safety rules in program designing: Non-adherence to these rules brings about security implications.

2.5 Constraint Analysis

Constraint analysis is divided into constraint generation and constraint solving program analysis process. Constraint generation is to establish variable types or analyze restraint system between different states using the rules of constraint generation; constraint solving is to solve the constraint system.

2.6 Patch Comparison

Patch comparison includes source code patch comparison and binary code patch comparison, and is mainly used to find "known" loopholes. After the software security vulnerabilities are found, the manufacturers usually release corresponding patches, so you can compare the code with patches to determine the location and causes of vulnerability.

2.7 Symbolic Execution

Symbolic execution is to represent the program's input by using symbol values rather than actual data, and produce algebraic expressions about the input symbols in the implementation process. By constraint solving method symbolic execution can detect possibility of errors.

2.8 Abstract Interpretation

Abstract interpretation is a formal description of program analysis. Generally, it involves analyzing and tracking program attributes and users concern.

2.9 Theorem Proving

Theorem proving is based on semantic analysis of the program, and can solve problems of infinite state systems. Theorem proving first converts the program into logic formulas, and then proves the program is a valid theorem by using axioms and rules.

2.10 Model Checking

Model checking process first constructs formal model for the program such as state machine or directed graph, then traverses and compares the model to verify whether the system meets pre-defined characteristics.

3. STATIC ANALYSIS TOOLS

In this section, we briefly discuss eight widely used static analysis tools.

3.1 ITS4

ITS4 [12] is a tool based on lexical analysis technique. It maintains a vulnerability database to read out the contents of the database at runtime and compare with the program codes. The database can be added, modified and deleted.

3.2 SPLINT

SPLINT (Secure Programming Lint) [13] is the expansion of LCLINT tool (for detecting buffer overflows and other security threats). It employs several lightweight static analyses. SPLINT need to use notes to perform cross-program analysis. SPLINT set up models for control flow and loop structure by using heuristic technology.

3.3 UNO

UNO [14] uses model checking to find loopholes in the code. UNO is named for the first character of three software defects: the use of uninitialized variables, dereferencing Nil-pointers, and Out-of-bound array indexing.

3.4 Check style

Check style is the most useful tool to help programmers write standard Java coding. Programmers can integrate Check style in development environment and use it to automatically check whether the Java codes are standard. Check style is configurable and can almost support all the coding standards.

3.5 ESC / Java

ESC/Java (Extended Static Checker for Java) [8] is a static detection tool based on theorem proving, and can find run-time error in Java code. Programmers can build ESC/ Java into the program verification environment, or install ESC / Java plug-in in the Eclipse.

3.6 Findbug

Findbug [15] is an open source static detection tools, which check the class or JAR files? By comparing binary codes with the defect model set, Findbug can detect latent problems. Findbug is not to find loopholes through analyzing the form and structure of class files, but by using the visitor

pattern. At present Findbug contains about 50 error pattern detectors. Find bug [16],[17] is one of the ASA tool for bug finding in Java; its detect error or violation in the programming practice, design pattern and automatically enumerate the violation into scariest, scary, troubling and concern as shown Fig 1. There are many other classification or prioritization of the bugs [18] such as according to category, pattern, class, or package alphabetically which is not the scope of this work. However, the gap is that errors ranking method may lead to that some true errors are often residing at the bottom of report list. As reported in previous work, these false errors popularly called false positives can cause static detection tools useless by hiding real errors in the code. This may also frustrate users to stop using this tool in detecting errors for large scale software product.

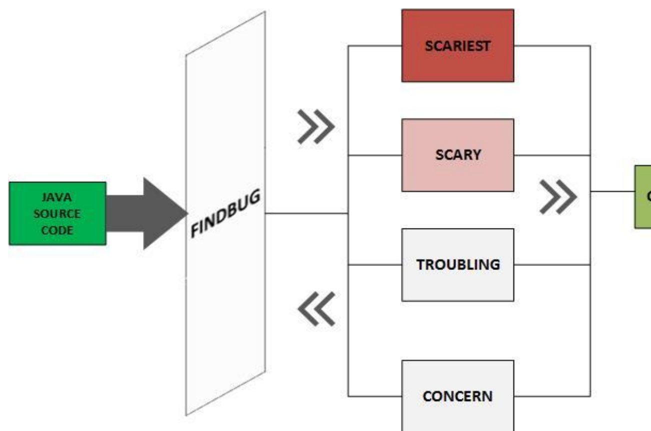

Fig 1: **Flow Diagram of Findbug**

3.7 PMD

PMD is an open source, rule-based static detection tool. PMD scans Java source codes and finds some potential problems, such as wrong code, duplicate code, fussy code or code to be further optimized. PMD includes a default rule set. In addition, it allows users to develop new rules and use it. Automatic Static analysis tools detect bug in source code without running the program. PMD works the same way very similar to conventional static analysis tools. This naturally means that the tool involves the generation and traversal of an abstract syntax tree. There are three ways in which PMD can be used: as a command line, an Eclipse plugin, or an Ant target element. As an Eclipse plugin, the plugin comes with a PMD perspective. In the Package Explorer, the files with violations are marked with error marks, but those error marks are a bit confusing because they are identical to compilation error marks. In the source editors, the violations are shown with markers. There is a Violation Overview window that is meant to provide a summary of violations, and it also provides the ability to toggle severity levels showed in the view; however, this functionality doesn't seem to be working yet. As an Ant target element, Ant automates the running of PMD, pretty similarly to what a batch file would do.

4. PROPOSED FRAMEWORK

This part of the paper presents the architecture and description of the proposed approach as shown in in Figure 2. The figure will be followed by a brief description of the main techniques used and the motivations that justify their use.

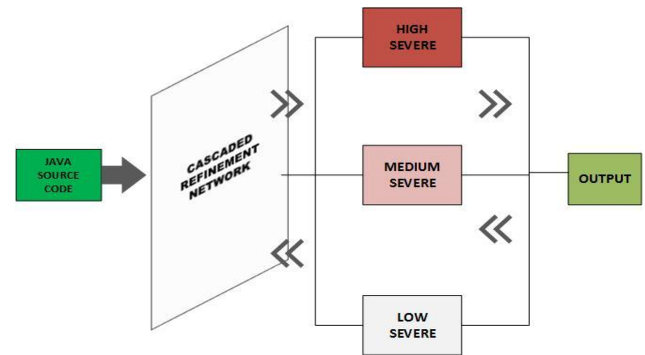

Fig: 2 Proposed Framework Based on the Cascaded Refinement Network

Cascaded Refinement Network [10] work by when presented a semantic label map, the network produces an image with photographic appearance that conforms to the input layout. The approach thus functions as a rendering engine that takes a two-dimensional semantic specification of the scene and produces a corresponding photographic image. The approach shows that photographic images can be synthesized from semantic layouts by a single feedforward network with appropriate structure, trained end-to-end with a direct regression objective. The Cascaded Refinement Network (CRN) is a cascade of refinement modules. Each module M I operate at a given resolution. The resolution of the first module (M0) is set to a default (i.e. 4x8). Resolution is doubled between consecutive modules (from M I −1 to M i). Let w i ×h i be the resolution of module i. The first module, M0, receives an input (downsampled to w 0 ×h 0) and produces a feature layer F0 at resolution w 0 × h 0 as output. All other modules M I (for i 6= 0) are structurally identical: M I receives a concatenation of the input (downsampled to w i ×h i) and the feature layer F i −1 (upsampled to w i ×h i) as input, and produces feature layer F I as output. The number of feature maps in F I is denoted by d i. Each module M I consists of three feature layers: the input layer, an intermediate layer, and the output layer. This is illustrated in Figure 3. The input layer has dimensionality w i ×h i × (d i −1 + c) and is a concatenation of the downsampled input (c channels) and a bilinearly upsampled feature layer F i −1 (d i −1 channels The intermediate layer and the output layer both have dimensionality w i ×h i ×d i .Each layer is followed by 3×3 convolutions, layer normalization, and LReLU nonlinearity [19].The output layer F¯I of the final module M¯I is not followed by normalization or nonlinearity. Instead, a linear projection (1×1 convolution) is applied to map F¯I (dimensionality w ¯ı ×h ¯ı ×d ¯ı) to the output (dimensionality w ¯ı ×h ¯ı ×3). The total number of refinement modules in a cascade depends on the output.

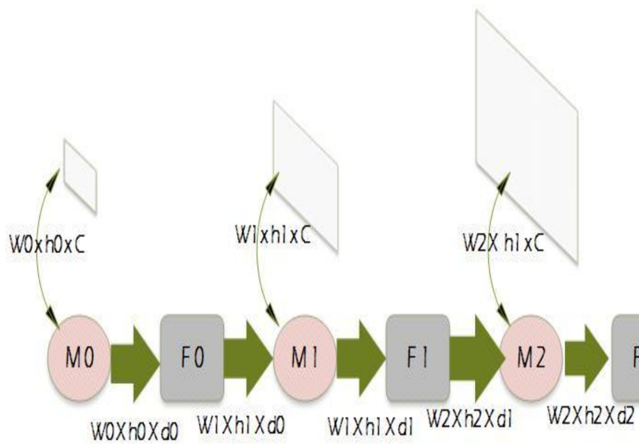

Fig: 3Architecture of CRN

The CRN consist of three layers of CNN: input layer, intermediate layer and output layer. Each module doubles the refinement process, Input layer upsampled feature maps of previous module plus downsampled input. The network reduces number of feature maps as cascade gets deeper until final module outputs. A vulnerability report will be given as output for further action and decisions. The potential exploitability of each detected vulnerability will be evaluated.

5. SUMMARY AND CONCLUSION

This paper, presented a theoretical framework for software vulnerability detection based on Cascaded Refinement network vulnerability to improve the detection of bugs in source code. The paper first provided an overview of the static analysis tools and techniques and subsequently detailed the proposed vulnerability detection based on the Cascaded Refinement network. Initial findings suggest that CRN is an effective tool for bug detection in large Java applications.

6. REFERENCES

[1] C. Kuang, Q. Miao, and H. Chen, "Analysis of software vulnerability," in Proceedings of the 5th WSEAS International Conference on Information Security and Privacy, 2006, pp. 218-223.

[2] I. V. Krsul, "Software vulnerability analysis," Purdue University, 1998.

[3] W. Jimenez, A. Mammar, and A. Cavalli, "Software Vulnerabilities, Prevention and Detection Methods: A Review1," Security in Model-Driven Architecture, p. 6, 2009.

[4] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in Security and privacy (SP), 2010 IEEE symposium on, 2010, pp. 497-512.

[5] P. E. Black and E. Fong, "Proceedings of Defining the State of the Art in Software Security Tools Workshop," NIST Special Publication, vol. 500, p. 264, 2005.

[6] A. Vetro, M. Morisio, and M. Torchiano, "An empirical validation of FindBugs issues related to defects," in Evaluation & Assessment in Software Engineering (EASE 2011), 15th Annual Conference on, 2011, pp. 144-153.

[7] P. Mell, K. Scarfone, and S. Romanosky, "A complete guide to the common vulnerability scoring system version 2.0," in Published by FIRST-Forum of Incident Response and Security Teams, 2007, p. 23.

[8] S. Hansman and R. Hunt, "A taxonomy of network and computer attacks," Computers & Security, vol. 24, pp. 31-43, 2005.

[9] M. Roesch, "Snort: Lightweight intrusion detection for networks," in Lisa, 1999, pp. 229-238.

[10] Q. Chen and V. Koltun, "Photographic image synthesis with cascaded refinement networks," in IEEE International Conference on Computer Vision (ICCV), 2017, p. 3.

[11] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao, "Joint face detection and alignment using multitask cascaded convolutional networks," IEEE Signal Processing Letters, vol. 23, pp. 1499-1503, 2016.

[12] I. Y.-L. Hsiao and C.-W. Jen, "A new hardware design and FPGA implementation for Internet routing towards IP over WDM and terabit routers," in Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on, 2000, pp. 387-390.

[13] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," IEEE software, vol. 19, pp. 42-51, 2002.

[14] G. J. Holzmann, "Static source code checking for user-defined properties," in Proc. IDPT, 2002.

[15] D. Hovemeyer and W. Pugh, "Finding bugs is easy," ACM Sigplan Notices, vol. 39, pp. 92-106, 2004.

[16] M. N. Al-Ameen, M. M. Hasan, and A. Hamid, "Making findbugs more powerful," in Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on, 2011, pp. 705-708.

[17] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," IEEE software, vol. 25, 2008.

[18] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation exploitation in error ranking," in ACM SIGSOFT Software Engineering Notes, 2004, pp. 83-93.

[19] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in Proc. icml, 2013, p. 3.