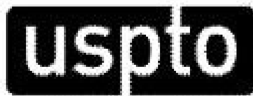


UNITED STATES
PATENT AND TRADEMARK OFFICE

P.O. Box 1450
Alexandria, VA 22313 - 1450
www.uspto.gov

ELECTRONIC ACKNOWLEDGEMENT RECEIPT

APPLICATION #	RECEIPT DATE / TIME	ATTORNEY DOCKET #
19/438,467	**12/31/2025 08:14:28 PM Z ET**	**BP25-382US1**

Title of Invention

METHOD AND APPARATUS FOR IDENTIFYING VULNERABLE SOFTWARE VERSIONS THROUGH SEMANTIC PAIR MAPPING BASED ON CODE LINE DEPENDENCY USING SECURITY PATCHES

Application Information

APPLICATION TYPE	Utility - Nonprovisional Application under 35 USC 111(a)	PATENT #	-
CONFIRMATION #	4017	FILED BY	SUYEON JIN
PATENT CENTER #	73814179	FILING DATE	-
CUSTOMER #	178942	FIRST NAMED INVENTOR	Heejo LEE
CORRESPONDENCE ADDRESS	-	AUTHORIZED BY	CHOONG WON CHO

DECLARATION AND ASSIGNMENT

As the below named inventor, I hereby declare that:

This declaration is directed to the attached application, or if not attached hereto, the below-identified application:

Title of Invention: **METHOD AND APPARATUS FOR IDENTIFYING VULNERABLE SOFTWARE VERSIONS THROUGH SEMANTIC PAIR MAPPING BASED ON CODE LINE DEPENDENCY USING SECURITY PATCHES**

Filing date:

Application number:

The above-identified application was made or authorized to be made by me.

I believe that I am the original inventor or an original joint inventor of a claimed invention in the application.

I hereby acknowledge that any willful false statement made in this declaration is punishable under 18 U.S.C. 1001 by fine or imprisonment of not more than five (5) years, or both.

WHEREAS, **KOREA UNIVERSITY RESEARCH AND BUSINESS FOUNDATION**, located at **145, Anam-ro, Seongbuk-gu, Seoul 02841 Republic of Korea**, hereinafter referred to as "Assignee," is desirous of acquiring the entire right, title and interest in and to said invention and application and in and to any Letters Patent of the United States and foreign countries which may be granted on the same;

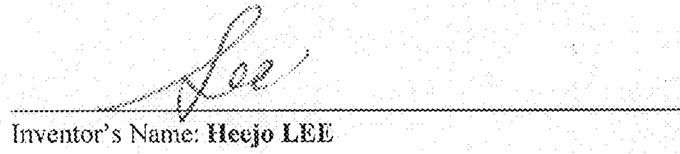
NOW, THEREFORE, in consideration of the promises, mutual covenants, and other good and valuable consideration, the receipt and sufficiency of which is hereby acknowledged, I do sell, assign and transfer unto the said Assignee, and Assignee's successors and assigns, the entire right, title and interest in and to said invention and said application, including rights to claim priority and any foreign applications for said invention, and including any non-provisionals, continuations, continuations-in-part, divisions, renewals, substitutions, conversions, reissues, prolongations or extensions thereof, and in and to any and all Letters Patent which may hereafter be granted on the same in the United States and any and all foreign countries,

AND, I hereby agree to, cooperate with Assignee and perform any and all necessary and reasonable acts that Assignee lawfully may request, in the prosecution of said application and any applications for said invention, to obtain or maintain Letters Patent for said invention, and to vest title thereto in Assignee, or Assignee's successors and assigns.

I hereby authorize and request Broadview IP LAW, PC, 3200 El Camino Real, Suite 260, Irvine, CA 92602, to insert herein above the title, application number, and filing date of said application when known.

Dec. 30, 2025

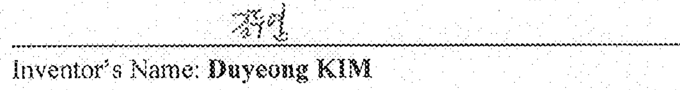
Date



Inventor's Name: **Heejo LEE**

2025. 12. 30

Date



Inventor's Name: **Duyeong KIM**

METHOD AND APPARATUS FOR IDENTIFYING VULNERABLE SOFTWARE VERSIONS THROUGH SEMANTIC PAIR MAPPING BASED ON CODE LINE DEPENDENCY USING SECURITY PATCHES

CROSS REFERENCE TO RELATED APPLICATION

[01] The present application claims the benefit of Korean Patent No. 10-2025-0137821 filed in the Korean Intellectual Property Office on September 24, 2025, the entire contents of which are incorporated herein by reference.

BACKGROUND OF THE INVENTION

Field of the Invention

[02] The present invention relates to the field of software security technology, and more particularly, to a technique of effectively identifying vulnerable versions of software by mapping code lines directly related to occurrence of vulnerability and code lines semantically connected thereto by means of data or control dependency as a semantic pair using security patch information.

[03] Meanwhile, this application has been supported by the national research and development projects as described below.

Project Unique Number: 2710068997

Project Number: II220277

Ministry Name: Ministry of Science and ICT

Title of Project Management (Specialized) Organization: Institute of Information & Communication Technology Planning & Evaluation

Title of Research Business: Development of basic technique for information security (R&D, Informatization)

Title of Research Project: Development of SBOM automatic generation and integrity verification technique for security of software supply network

Title of Project Implementing Organization: Korea University Research and Business Foundation

Research Period: January 1, 2025 ~ December 31, 2025

National Research and Development Projects that have supported this invention.

Project Unique Number: 2710069045

Project Number: 00440780

Ministry Name: Ministry of Science and ICT

Title of Project Management (Specialized) Organization: Institute of Information & Communication Technology Planning & Evaluation

Title of Research Business: Development of basic technique for information security (R&D, Informatization)

Title of Research Project: Integrated vulnerability management platform technique based on linkage of SBOM and VEX for internalization of security throughout lifecycle of software supply network

Title of Project Implementing Organization: Korea University Research and Business Foundation

Research Period: January 1, 2025 ~ December 31, 2025

National Research and Development Projects that have supported this invention.

Project Unique Number: 2710074877

Project Number: II201819

Ministry Name: Ministry of Science and ICT

Title of Project Management (Specialized) Organization: Institute of Information & Communication Technology Planning & Evaluation

Title of Research Business: Development of core human resources for leading digital technologies (R&D)

Title of Research Project: Development of premium ICT human resources (Korea University)

Title of Project Implementing Organization: Korea University Research and Business Foundation

Research Period: January 1, 2025 ~ December 31, 2025

Background of the Related Art

[04] Software security vulnerability acts as a major entry point of cyberattack. Particularly, in software based on C/C++, low-level security flaws, such as memory management

errors and vulnerabilities in using pointers, frequently occur and generate security incidents. The key subject of security response is to identify security vulnerabilities in an early stage and clearly specify affected software versions.

[05] Approaches widely used currently to detect security vulnerabilities rely on public databases such as the National Vulnerability Database (NVD) or algorithm-based tools that track code lines deleted from patches. These techniques identify existence of vulnerabilities on the basis of relatively simple code difference, and report whether a specific software version is vulnerable. Practically, these methods are utilized by security experts as important reference data in the process of managing vulnerabilities and setting the scope of response.

[06] However, existing techniques have some limitations. When patched code and deleted code are compared simply based on the difference in the statements, existence of vulnerabilities may not be correctly identified when grammatical expressions of the codes change. In addition, since the existing techniques may not track the evolution process of functions or consider the semantic dependencies in the codes, it is difficult to correctly determine whether vulnerabilities still exist in a specific version. A problem of overreporting the start point of vulnerable versions or mistaking an insecure version as secure on the contrary by misidentifying an end point may occur according thereto. Furthermore, there is a limitation in that a discrepancy is generated between an actual security state and the database as special cases where patches are applied only to intermediate versions are not reflected.

[07] The problems like this may invite fatal consequences in the process of security response. Actually, incorrect version information may make managers to believe by mistake a vulnerable system as secure and leave attack surfaces unattended, or on the contrary, it may invite inefficiency of repeatedly performing unnecessary security measures on a secure system.

[08] Accordingly, a new technical approach is needed to more precisely specify vulnerable versions by analyzing core lines that cause the vulnerabilities and code lines semantically linked thereto, as well as syntactic differences.

SUMMARY OF THE INVENTION

[09] Therefore, the present invention has been made in view of the above problems, and it is an object of the present invention to provide a technique of identifying a scope of versions affected by software security vulnerabilities more accurately and reliably. To this end, the present invention proposes a technique of determining whether there is a vulnerability by analyzing core code lines that may cause the vulnerability, together with code lines semantically linked thereto, rather than simply relying on the difference in the statements, so that the vulnerabilities can be identified consistently even when some of the expressions of the code are changed.

[10] In addition, another object of the present invention is to precisely identify when a specific function is vulnerable and when the function becomes secure as patches are applied by tracing the lineage of the function and analyzing evolution of the code in each version. This may improve the efficiency and reliability of security responses by systematically deriving start and end versions corresponding to each vulnerability, and reducing the discrepancy between information stored in a security database and an actual state of software.

[11] Meanwhile, the technical problems of the present invention are not limited to the technical problems mentioned above, and unmentioned other technical problems can be clearly understood by those skilled in the art from the following description.

[12] To accomplish the above objects, according to one aspect of the present invention, there is provided a method performed by a software vulnerable version identification apparatus

operated by a processor, the method comprising: an operation of receiving vulnerability information to be analyzed and acquiring security patch information and a source code corresponding to the vulnerability information; an operation of classifying a function including code lines deleted from the source code due to a security patch as a vulnerable function and a function including code lines added in the source code due to the security patch as a patch function, on the basis of the security patch information; an operation of specifying each of the vulnerable function and the patch function as essential lines related to security, and specifying a line declaring variables used in any one first essential line, a line assigning a value to a variable, or a line determining whether or not to execute the first essential line as a first dependent line; an operation of generating a semantic pair in which vulnerability is semantically connected by mapping the first essential line and the first dependent line; an operation of collecting functions of all versions for at least one or more first functions selected to be analyzed among the vulnerable function and the patch function as an analysis target; and an operation of determining whether each version is vulnerable on the basis of a ratio of the semantic pairs included in a function of each version.

[13] In addition, the acquiring operation may include: an operation of collecting a commit URL from a CVE detailed page of an NVD database including the vulnerability information and acquiring security patch information corresponding to the commit URL; and an operation of acquiring information on the source code by copying an entire software repository from a software repository address included in the commit URL into a local environment.

[14] In addition, the classifying operation may include: an operation of extracting a source code file of a version before applying a patch and a source code file of a version after applying the patch from the software repository; an operation of extracting a function including a code line where a + or − mark is assigned due to the security patch from each source code file

using Universal Ctags; and an operation of classifying a function including a code line where a − mark is assigned as a vulnerable function and a function including a code line where a + mark is assigned as a patch function among the extracted functions.

[15] In addition, the operation of generating a semantic pair may include: an operation of generating an essential line corresponding to a vulnerable function and a dependent line mapped thereto as a vulnerable semantic pair; and an operation of generating an essential line corresponding to a patch function and a dependent line mapped thereto as a patch semantic pair.

[16] In addition, the operation of generating a semantic pair may include: an operation of removing a line in which the code added by the patch and the code deleted by the patch are syntactically identical, from the semantic pair; and an operation of removing, when there is a dependent line included in both the vulnerable semantic pair and the patch semantic pair, the dependent line from the patch semantic pair to prioritize an evidence of vulnerability.

[17] In addition, the determining operation may include: an operation of calculating a vulnerability similarity as a proportion of all vulnerable semantic pairs included in a first version; an operation of calculating a patch similarity as a proportion of all patch semantic pairs included in the first version; and an operation of determining the first version as a vulnerable version when the vulnerability similarity is greater than a preset vulnerability threshold and the patch similarity is smaller than a preset patch threshold, for the first version of the first function.

[18] In addition, the operation of collecting functions of all versions may include an operation of collecting functions of all versions for the first function by executing a command of “git log --follow -L:<function>:<file>” on the software repository and tracking history of change in the first function.

[19] In addition, the determining operation may include: an operation of removing annotations and spaces included in the code lines; an operation of converting all characters in the code lines into lowercase; and an operation of excluding a preset statement included in the line codes from the analysis target, in order to normalize the code lines of a function corresponding to each version of the first function before comparing on the basis of the semantic pairs for the functions of each version.

[20] In addition, the first function includes a plurality of functions, and the determining operation may include: an operation of deriving a vulnerability determination result for each function targeting the plurality of functions included in a specific software version; an operation of classifying, when at least one among the plurality of functions is determined as vulnerable, the software version as a vulnerable version; and an operation of generating final vulnerable version information in a form of a range of a start version and an end version of software by performing an operation of determining vulnerability of each function and classifying the vulnerable version on all versions of the software.

[21] According to another embodiment, there is provided a software vulnerable version identification apparatus comprising: a memory including instructions; and a processor that performs a predetermined operation based on the instructions, wherein the operation of the processor includes: an operation of receiving vulnerability information to be analyzed and acquiring security patch information and a source code corresponding to the vulnerability information; an operation of classifying a function including code lines deleted from the source code due to a security patch as a vulnerable function and a function including code lines added in the source code due to the security patch as a patch function, on the basis of the security patch information; an operation of specifying each of the vulnerable function and the patch function as

essential lines related to security, and specifying a line declaring variables used in any one first essential line, a line assigning a value to a variable, or a line determining whether or not to execute the first essential line as a first dependent line; an operation of generating a semantic pair in which vulnerability is semantically connected by mapping the first essential line and the first dependent line; an operation of collecting functions of all versions for at least one or more first functions selected to be analyzed among the vulnerable function and the patch function as an analysis target; and an operation of determining whether each version is vulnerable on the basis of a ratio of the semantic pairs included in a function of each version.

BRIEF DESCRIPTION OF THE DRAWINGS

[22] FIG. 1 is a view showing the configuration of a software vulnerable version identification apparatus according to an embodiment.

[23] FIG. 2 is a flowchart illustrating the steps of the operation performed by a software vulnerable version identification apparatus according to an embodiment.

[24] FIG. 3 is a conceptual view showing the flow of data processed by a software vulnerable version identification apparatus according to an embodiment on the basis of the operation of FIG. 2.

[25] FIG. 4 is an exemplary view showing an operation of classifying vulnerable functions or patch functions in a source code according to an embodiment.

[26] FIG. 5 is an exemplary view showing an operation of searching for essential lines and dependent lines corresponding thereto according to an embodiment.

[27] FIG. 6 is a view showing the result of an experiment evaluating performance of various indicators in a vulnerable version determination process using existing techniques and the technique of the present invention.

[28] FIG. 7 is a view showing the difference in calculating a range of vulnerable versions by the type using the NVD and the technique of the present invention.

[29] FIG. 8 is a table showing the result of quantitatively analyzing the difference between various types of vulnerable version information using the NVD and the technique of the present invention.

[30] FIG. 9 is a graph showing the time required to analyze the range of vulnerable versions for each CVE using the technique of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

[31] Details of the objects and technical configurations of the present invention and operational effects according thereto will be more clearly understood by the following detailed description based on the drawings attached in the specification of the present invention. An embodiment according to the present invention will be described in detail with reference to the accompanying drawings.

[32] The embodiments disclosed in this specification should not be construed or used as limiting the scope of the present invention. For those skilled in the art, it is natural that the description including the embodiments of the present specification have various applications. Accordingly, any embodiments described in the detailed description of the present invention are

illustrative for better describing of the present invention, and are not intended to limit the scope of the present invention to the embodiments.

[33] The functional blocks shown in the drawings and described below are merely examples of possible implementations. Other functional blocks may be used in other implementations without departing from the spirit and scope of the detailed description. In addition, although one or more functional blocks of the present invention are expressed as separate blocks, one or more of the functional blocks of the present invention may be combinations of various hardware and software configurations that perform the same function.

[34] In addition, the expressions including certain components are expressions of "open type" and only refer to existence of corresponding components, and should not be construed as excluding additional components.

[35] Furthermore, when a certain component is referred to as being “connected” or “coupled” to another component, it may be directly connected or coupled to another component, but it should be understood that other components may exist in between.

[36] Hereinafter, various embodiments of the present invention will be described with reference to the accompanying drawings. However, it should be understood that this is not intended to limit the present invention to specific embodiments, but to include various modifications, equivalents, and/or alternatives of the embodiments of the present invention.

[37] The present invention proposes a software vulnerable version identification apparatus 100 that implements a technique of effectively identifying vulnerable versions of software by mapping code lines directly related to occurrence of vulnerability and code lines semantically connected thereto by means of data or control dependency as a semantic pair using security patch information.

[38] Hereinafter, the configuration of the software vulnerable version identification apparatus 100 of the present invention and the operation of each component will be examined.

[39] FIG. 1 is a view showing the configuration of the software vulnerable version identification apparatus 100 according to an embodiment (hereinafter, referred to as an 'apparatus 100').

[40] Referring to FIG. 1, the apparatus 100 according to an embodiment may include a memory 110, a processor 120, an input/output interface 130, and a communication interface 140.

[41] The memory 110 may store data acquired from an external device or data generated by itself. The memory 110 may store instructions that may perform the operation of the processor 120. For example, the memory 110 may store vulnerability information, security patch information, source code, semantic pairs, information on each version of a function, and the like.

[42] The processor 120 is a computing device that controls the overall operation. The processor 120 may execute instructions stored in the memory 110. The operation of the apparatus 100 according to the embodiment of this document may be understood as an operation performed by the processor 120.

[43] The input/output interface 130 may include a hardware interface or a software interface for inputting or outputting information.

[44] The communication interface 140 allows to transmit and receive information through a communication network. To this end, the communication interface 140 may include a wireless communication module or a wired communication module.

[45] The apparatus 100 may be implemented in various types of apparatuses that can perform an operation through the processor 120 and transmit and receive information through a

network. For example, although the apparatus may be implemented as a server, a computer apparatus, a portable communication apparatus, a smart phone, a portable multimedia apparatus, a laptop computer, a tablet PC, or the like, it is not limited to these examples.

[46] FIG. 2 is a flowchart illustrating the steps of the operation performed by the apparatus 100 according to an embodiment. FIG. 3 is a conceptual view showing the flow of data processed by the apparatus 100 according to an embodiment on the basis of the operation of FIG. 2. The operation of the apparatus 100 according to the embodiment of FIGS. 2 and 3 may be understood as an operation performed by the processor 120.

[47] Meanwhile, each step disclosed in FIGS. 2 and 3 is merely a preferred embodiment in achieving the objects of the present invention, and some steps may be added or deleted as needed, and any one step may be included and performed in another step. The order of each operation disclosed in FIGS. 2 and 3 is arranged only for convenience of understanding, and this order is not limited to a time-series order, and the order may be changed to operate in a different way according to the choice of the designer.

[48] Referring to FIGS. 2 and 3 together, at step S1010, the apparatus 100 may receive vulnerability information to be analyzed and acquire security patch information and a source code corresponding to the vulnerability information.

[49] Here, the vulnerability information is a unique identifier assigned to specify a security flaw of software, and may be in a form such as Common Vulnerabilities and Exposures (CVE) ID.

[50] The security patch information refers to information on the change of code opened by the developers or suppliers to solve the vulnerability information, and may include, for example, a commit identifier (e.g., commit hash) corresponding to a patch, the URL where the commit is

posted (e.g., GitHub commit URL), a list of changed files included in the patch, diff information indicating code lines added (+) or deleted (−) in each file, and the like.

[51] For example, the apparatus 100 may search for the detail page of the Common Vulnerabilities and Exposures (CVE) of a vulnerability database such as the National Vulnerability Database (NVD) using the vulnerability information, and secure security patch information by collecting the commit URL written in the detail page of the CVE. In addition, the apparatus 100 may acquire the source code by copying a corresponding repository into the local environment (e.g., git clone) using the address of the software repository written in the commit.

[52] The source code means all the code stored in a software repository corresponding to the security patch information, and include, for example, a full copy of the repository (e.g., a working copy acquired using the git clone), source files (e.g., .c, .cpp, .java, .py, etc.), a version of each source file at a specific time point (e.g., a snapshot identified by a commit hash), and version management history (e.g., commit log, branch information).

[53] For example, when a specific CVE ID (such as CVE-2025-XXXX) is input, the apparatus 100 may search the NVD database using the CVE ID as a keyword, extract the GitHub commit URL included in the search result, and secure the source code to be analyzed by copying the entire software repository into the local environment using the URL. In the present invention, the function and changed code lines before and after applying a patch are extracted using the source code, and all versions of the function are collected according to the history of change in the function and used thereafter for generation of semantic pairs and determination of vulnerability.

[54] At step S1020, on the basis of the security patch information, the apparatus 100 may classify a function including code lines deleted from the source code due to the security patch

as a vulnerable function and a function including code lines added in the source code due to the security patch as a patch function.

[55] FIG. 4 is an exemplary view showing an operation of classifying vulnerable functions or patch functions in a source code according to an embodiment.

[56] Referring to FIG. 4, the apparatus 100 may extract functions including code lines having a deletion (-) mark and functions including code lines having an addition (+) mark, respectively, by comparing source code files before and after a security patch is applied. For example, in the example shown in FIG. 4, a function including a deleted code line having a − mark, such as nstrips64 = TIFFhowmany_64(...), is classified as a vulnerable function, and a function including an added code line having a + mark, such as nstrips = TIFFhowmany_32(...) and if (nstrips == 0), is classified as a patch function.

[57] As the apparatus 100 extracts, to this end, a source code file of a version before applying the patch and a source code file of a version after applying the patch from the software repository, and extracts a function including a code line where a + or − mark is assigned from each file using a parsing tool such as Universal Ctags, it may classify a function where a − mark is assigned as a vulnerable function and a function having a + mark as a patch function among the extracted functions.

[58] At step S1030, the apparatus 100 may specify each of the vulnerable function and the patch function as essential lines related to security, and specify a line belonging to any one among a line declaring variables used in any one first essential line, a line assigning a specific value to a variable used in any one of the first essential lines, and a line determining whether or not to execute the first essential line as a first dependent line.

[59] That is, when there is a statement in which a variable referenced in the first essential line is defined first within a function, the variable declaration line may be specified as the first dependent line as it is absolutely necessary for normal execution of the first essential line.

[60] In addition, when there is a statement in which a specific value is assigned to a variable used in the first essential line, the assignment statement may be specified as the first dependent line as it directly affects the operation result of the first essential line.

[61] In addition, when execution of the first essential line is controlled by a conditional statement (if, while, etc.) or a branch statement, the line of the conditional statement or branch statement may be specified as the first dependent line as it provides important contextual information that determines whether or not to execute the first essential line.

[62] Here, the term "first" is merely an identifier used to specify any one among a plurality of essential lines and does not limit the meaning of the sequence. Therefore, the terms "first essential line" and "first dependent line" are only convenient notations used to exemplarily refer to any one of all essential lines.

[63] To this end, the apparatus 100 may analyze C/C++ source code using a code analysis tool such as Joern, and generate a code property graph (CPG). The code property graph is a graph structure that comprehensively expresses data flow, control flow, and abstract syntax tree information, and the apparatus 100 may identify data dependency and control dependency between code lines through the code property graph. The results of the dependency analysis are utilized to systematically grasp the connection relationships between the essential lines and the dependent lines, and support to configure semantic pairs more accurately as a result.

[64] FIG. 5 is an exemplary view showing an operation of searching for essential lines and dependent lines corresponding thereto according to an embodiment.

[65] Referring to FIG. 5, the red highlighted portion in the code indicates an essential line directly related to security (e.g., vulnerable code line in a vulnerable function), and the blue highlighted portions indicate dependent lines having a data or control dependency relationship with the essential line. For example, as the line declaring variable stripbytes and the line assigning value rowblockbytes define or update the variables used in the essential line, they are detected as data-dependent lines. In addition, the lines comparing rowsperstrip in the conditional statements are detected as control-dependent lines as they are statements that control whether or not to execute the essential line.

[66] Accordingly, the apparatus 100 may analyze data flow and control flow on the basis of a specific essential line and identify all dependent lines semantically connected to the essential line, and may configure semantic pairs directly or indirectly related to vulnerability through the operations described below.

[67] At step S1040, the apparatus 100 may generate a semantic pair in which vulnerability is semantically connected by mapping the first essential line and the first dependent line corresponding to the first essential line. That is, the apparatus 100 does not generate a semantic pair only for one specific essential line, but searches for each essential line and a dependent line corresponding thereto for the essential lines of all vulnerable functions and patch function, and generate a plurality of semantic pairs by mapping them.

[68] Here, the semantic pair is not simply a relationship based on the physical proximity of code lines, but refers to a pair reflecting a relationship influencing each other in terms of program execution logic, such as variable references, value assignments, and control flow branches. For example, when a variable used in an essential line is declared or initialized in a specific dependent line, or when whether or not to execute an essential line is controlled by a conditional

dependent line, the two lines are considered as being semantically connected and form a single semantic pair. Therefore, the semantic pair becomes a unit that can explain how security vulnerabilities propagate through a code execution path and data flow.

[69] At this point, the apparatus 100 may classify the semantic pairs into two categories by generating an essential line corresponding to a vulnerable function and a dependent line mapped thereto as a vulnerable semantic pair, and generating an essential line corresponding to a patch function and a dependent line mapped thereto as a patch semantic pair. Through this, the apparatus 100 may clearly distinguish a cause of generating the security vulnerability and a modification for supplementing it even within the same code structure, and more precisely determine whether a corresponding version is vulnerable in the stages thereafter by comparing how many vulnerable semantic pairs or a patch semantic pairs does each version of a function include.

[70] Additionally, the apparatus 100 may refine unnecessary or duplicate relationships in the process of generating semantic pairs to increase the reliability and accuracy of the semantic pairs.

[71] For example, when there is a line in which a code added by a patch and a code deleted by a patch are syntactically identical, the apparatus 100 may remove the line from the semantic pair. Accordingly, unnecessary duplicate pairs that may occur due to a simple change in the location or difference in the expression of a code can be removed.

[72] In addition, when the same dependent line is included in both the vulnerable semantic pair and the patch semantic pair, the apparatus 100 may priority the evidence of vulnerability by leaving the dependent line only in the vulnerable semantic pair and removing it from the patch semantic pair. This may reduce ambiguity in the process of determining vulnerability and strengthen validity of a vulnerable code.

[73] At step S1050, the apparatus 100 may collect functions of all versions for at least one or more first functions selected to be analyzed among the vulnerable function and the patch function as an analysis target.

[74] Here, the first function is a function that the user is interested in and has specified as a target for analysis among the functions that have been changed due to the security patch, and may be any one among the vulnerable function and the patch function. That is, the first function is a core function selected according to the analysis object of the user, and may track the process of generating and fixing vulnerabilities by collecting all versions of the function, and may be used thereafter for generation of semantic pairs and determination of vulnerability.

[75] For example, the apparatus 100 may collect functions of all versions for the first function by executing the command of "git log --follow -L:<function>:<file>" on the software repository and tracking the history of change in the first function. At this point, the history of change in the first function may include details of addition, deletion, and modification of code of function units, and the apparatus 100 may acquire a snapshot of the first function existing at each time point of commit of the source code repository on the basis of the history. Accordingly, the apparatus 100 may track code lines included in the initial version of the first function and how the code lines are changed in the subsequence versions, and as a series of these version-specific functions are collected, the time point of occurring a specific vulnerability and the time point of applying a patch can be analyzed continuously.

[76] At step S1060, the apparatus 100 may determine whether each version is vulnerable on the basis of the ratio of the semantic pairs included in the function of each version for the first function.

[77] At this point, the apparatus 100 may normalize the code lines of a function corresponding to each version before comparing on the basis of the semantic pairs so that the comparison based on the semantic pairs is not distorted by the code style or unnecessary difference in the symbols. Specifically, the apparatus 100 may exclude unnecessary non-semantic elements by removing the annotations and spaces included in the code lines of a function corresponding to each version of the first function, resolve the inconsistency due to the difference between lowercase and uppercase characters by converting all characters in the code lines into lowercase, and set only the portions related to essential code operation as a comparison target by excluding specific statements set in advance (e.g., import statements, simple log output statements, etc.) from the analysis target.

[78] Accordingly, in order to quantitatively determine whether the function of each version for the first function is a vulnerable version, the apparatus 100 may determine whether the first version is vulnerable on the basis of the proportion of the semantic pairs included in the function of the first version, which is a specific version. Here, the term "first" is merely an identifier used to specify any one among a plurality of versions and does not limit the meaning of the sequence. Therefore, the term "first version" is only a convenient notation used to exemplarily refer to any one of all versions.

[79] Specifically, the apparatus 100 may calculate a vulnerability similarity as the proportion of all vulnerable semantic pairs included in the first version. That is, the apparatus 100 divides the number of vulnerable semantic pairs existing in the first version by the total number of vulnerable semantic pairs, and quantitatively calculates how many vulnerable behaviors before the security patch does the corresponding version preserve. Through this, the closer the vulnerability similarity is to 1, it means that it is similar to a state in which the patches are not applied, and the

closer the value is to 0, it indicates that the effect of removing the vulnerability due to the patches is reflected.

[80] In addition, the apparatus 100 may calculate a patch similarity as the proportion of all patch semantic pairs included in the first version. That is, the apparatus 100 divides the number of patch semantic pairs existing in the first version by the total number of patch semantic pairs, and quantitatively calculates how much the version reflects the modification operation after the security patch. Accordingly, the closer the patch similarity value is to 1, it means that it is a state in which the patches are more faithfully applied, and the closer the value is to 0, it indicates that there is almost no trace of applying the patches.

[81] Accordingly, when the vulnerability similarity is greater than a preset vulnerability threshold (e.g., 0.5) and the patch similarity is smaller than a preset patch threshold (e.g., 0.5), the apparatus 100 may determine the first version as a vulnerable version. For example, when the vulnerability similarity is calculated as 0.7 and the patch similarity is calculated as 0.3 for a specific first version in the case where the vulnerability threshold and the patch threshold are set to 0.5, respectively, the apparatus 100 may classify the first version as a vulnerable version as it is determined that the first version is in a state that does not sufficiently reflect the effect of the security patch although it includes a large number of code patterns related to vulnerability.

[82] This determination process is repeatedly applied to all versions of the first function to systematically derive whether each version is vulnerable or not, and through this, a range of vulnerable version of the software can be determined finally.

[83] Meanwhile, rather than only one first function is specified to be analyzed, a plurality of first functions may be specified simultaneously according to the interest of a user or purpose of analysis.

[84] Accordingly, the apparatus 100 may derive a vulnerability determination result for each function targeting a plurality of first functions included in a specific software version. At this point, when at least one among the plurality of functions is determined as vulnerable, the apparatus 100 may classify the software version as a version vulnerable overall. In addition, this determination procedure may be repeatedly applied to all versions of the software, and whether or not vulnerable may be systematically derived for each version. The apparatus 100 may synthesize these results to generate final vulnerable version information in the form of a range including the start version and the end version of the software.

[85] FIG. 6 is a view showing the result of an experiment evaluating performance of various indicators in a vulnerable version determination process using existing techniques and the technique of the present invention.

[86] Referring to FIG. 6, the technique of the present invention (CLOVERY) shows accuracy remarkably higher than those of the existing techniques (NVD, V0Finder, V-SZZ), and achieves 97.43% on the basis of the Macro-F1 score. At this point, the Precision is measured as 96.27%, and the Recall is measured as 98.62%, and it can be confirmed that high precision and recall are simultaneously secured in identifying versions that reflect security vulnerability. Accordingly, FIG. 6 experimentally shows that the technique of the present invention remarkably improves the accuracy and reliability of vulnerable version determination compared to existing techniques.

[87] FIG. 7 is a view showing the difference in calculating a range of vulnerable versions by the type using the NVD and the technique of the present invention.

[88] Referring to FIG. 7, it is shown that the technique of the present invention (CLOVERY) is different from the NVD as it precisely determines the start point and the end point

of a vulnerable version on the basis of a semantic pair. T1 and T2 show the difference in the start point of the vulnerable version, and in the case of T1, the NVD specifies a version earlier than that of the CLOVERY as the start point, and in the case of T2, the CLOVERY determines a start version earlier than that of the NVD. T3 and T4 show the difference in the end point of the vulnerable version, and in the case of T3, the CLOVERY suggests an earlier end point, and in the case of T4, the NVD suggests an earlier end point. Finally, T5 means a case where the difference occurs in the middle section of the vulnerable version, and whereas the NVD produces a continuous range of vulnerable version, the present invention shows, through an analysis of semantic pair units, that the unnecessarily expanded range can be reduced by identifying only a specific version that actually includes a vulnerable code. Therefore, FIG. 7 shows that the method of the present invention may more precisely specify the start point, the end point, and the middle section of a vulnerable version through a semantic analysis, rather than a simple syntactic comparison.

[89] FIG. 8 is a table showing the result of quantitatively analyzing the difference between various types of vulnerable version information using the NVD and the technique of the present invention.

[90] Referring to FIG. 8, the difference occurring in the process of calculating the range of vulnerable version of the two methods may be categorized into five types of T1 to T5. The result shown in FIG. 8 indicates that the technique of the present invention (CLOVERY) actually calculates a more precise and valid range of vulnerable version through a code-based semantic analysis, while showing various differences compared to the NVD even in the middle section, as well as in the start point and the end point of the vulnerable version.

[91] FIG. 9 is a graph showing the time required to analyze the range of vulnerable versions for each CVE using the technique of the present invention.

[92] Referring to FIG. 9, in the case of some early CVEs, as the details of the security patch are complex or the history of change in the code is extensive, there are also cases of requiring several thousand seconds (e.g., about 6,000 seconds or more). However, in most of the CVEs, it can be confirmed that the elapsed time abruptly decreases, and the analysis is completed within the range of a few hundred seconds, and thereafter, it can be processed stably within a short period of time for CVEs of approximately 2,000 or more. Therefore, FIG. 9 shows that the present invention may efficiently analyze a large-scale CVE set although the history of change in the code is complex.

[93] According to the embodiment described above, as the present invention analyzes whether or not a security patch is applied on the basis of semantic correlations rather than simple difference in the code, whether software is vulnerable can be determined more precisely. Specifically, as essential lines directly linked to vulnerability and dependent lines configuring the execution conditions or data flow thereof are considered together, consistent detection of vulnerability is possible although code expressions are partially modified or function names are changed. Through this, the start point and the end point of a vulnerable version can be clearly identified, and the problem of overreporting and omission generated in the existing databases can be reduced effectively.

[94] In addition, the present invention may accurately recall an actual security state by tracing the lineage of the function and reflecting evolution of the code for each software version, and based on this, the scope of vulnerable versions can be quickly derived in an automated method. This effect may be utilized by security personnel to instantaneously set a response scope and

determine the priority of patches, and provides a result of improving both the efficiency and reliability of security operation.

[95] It should be understood that various embodiments of this document and the terms used herein are not intended to limit the technical features described in this document to specific embodiments, but include various modifications, equivalents, or substitutes of the embodiments. In connection with the description of drawings, similar reference numerals may be used for similar or related components. The singular form of a noun corresponding to an item may include one or more items, unless the related context clearly indicates otherwise.

[96] In this document, each of phrases such as “A or B”, “at least one among A and B”, “at least either A or B”, “A, B, or C”, “at least one among A, B, and C”, and “at least either A, B, or C” may include all possible combinations of the items listed together in a corresponding phrase among the phrases. Terms such as “1st”, “2nd”, “first”, or “second” may be used only to distinguish a corresponding component from another corresponding component, and do not limit the components in any other aspect (e.g., importance or order). When a certain (e.g., a first) component is referred to as being “coupled” or “connected” to another (e.g., a second) component with or without a term such as “functionally” or “communicatively”, it means that the component may be connected to another component directly (e.g., wired), wirelessly, or through a third component.

[97] The term "module" used in this document may include a unit implemented in hardware, software, or firmware, and may be used interchangeably with terms such as logic, logic block, part, or circuit. A module may be an integrally configured component, or a minimum unit of a component or a portion thereof that performs one or more functions. For example, according

to an embodiment, a module may be implemented in the form of an application-specific integrated circuit (ASIC).

[98] Various embodiments of this document may be implemented as software (e.g., a program) including one or more commands stored in a storage medium (e.g., a memory) that can be read by a device (e.g., an electronic device). The storage medium may include a random-access memory (RAM), a memory buffer, a hard drive, a database, an erasable programmable read-only memory (EPROM), an electrically erasable read-only memory (EEPROM), a read-only memory (ROM), and/or the like.

[99] In addition, the processor in the embodiments of this document may call at least one command among one or more stored commands from the storage medium and execute the command. This allows the device to operate to perform at least one function according to the called at least one command. The one or more commands may include a code generated by a compiler or a code that can be executed by an interpreter. The processor may be a general-purpose processor, a Field Programmable Gate Array (FPGA), an Application Specific Integrated Circuit (ASIC), a Digital Signal Processor (DSP), and/or the like.

[100] The storage medium that can be read by a device may be provided in the form of a non-transitory storage medium. Here, 'non-transitory' only means that the storage medium is a tangible device and does not include signals (e.g., electromagnetic waves), and this term does not distinguish the cases where data is stored semi-permanently on the storage medium from the cases where data is stored temporarily.

[101] The method according to various embodiments disclosed in this document may be provided to be included in a computer program product. The computer program product may be traded between a seller and a buyer as goods. The computer program product may be distributed

in the form of a machine-readable storage medium (e.g., a compact disc read only memory (CD-ROM)), or may be distributed online (e.g., downloaded or uploaded) through an application store (e.g., Play Store) or directly distributed between two user devices (e.g., smartphones). In the case of online distribution, at least a part of the computer program product may be at least temporarily stored in a machine-readable storage medium, such as a memory of a manufacturer's server, an application store's server, or a server, or may be temporarily generated.

[102] According to various embodiments, each component (e.g., a module or a program) of the components described above may include a single or a plurality of entities. According to various embodiments, one or more of the components or operations of the components described above may be omitted, or one or more other components or operations may be added. Alternatively or additionally, a plurality of components (e.g., modules or a programs) may be integrated into a single component. In this case, the integrated component may perform one or more functions of each of the plurality of components in a way identical or similar to those performed by the corresponding component among the plurality of components before the integration. According to various embodiments, the operations performed by the modules, programs, or other components may be executed sequentially, in parallel, repeatedly, or heuristically, or one or more of the operations may be executed in a different order or omitted, or one or more other operations may be added.

[103] According to the present invention, as the present invention analyzes whether or not a security patch is applied on the basis of semantic correlations rather than simple difference in the code, whether software is vulnerable can be determined more precisely. Specifically, as essential lines directly linked to vulnerability and dependent lines configuring the execution conditions or data flow thereof are considered together, consistent detection of vulnerability is

possible although code expressions are partially modified or function names are changed. Through this, the start point and the end point of a vulnerable version can be clearly identified, and the problem of overreporting and omission generated in the existing databases can be reduced effectively.

[104] In addition, the present invention may accurately recall an actual security state by tracing the lineage of the function and reflecting evolution of the code for each software version, and based on this, the scope of vulnerable versions can be quickly derived in an automated method. This effect may be utilized by security personnel to instantaneously set a response scope and determine the priority of patches, and provides a result of improving both the efficiency and reliability of security operation.

[105] Meanwhile, the effects of the present invention are not limited to those mentioned above, and unmentioned other technical effects will be clearly understood by those skilled in the art from the following descriptions.

DESCRIPTION OF SYMBOLS

[106] 100: Apparatus

[107] 110: Memory

[108] 120: Processor

[109] 130: Input/Output Interface

[110] 140: Communication Interface

What is claimed is:

1. A method performed by a software vulnerable version identification apparatus operated by a processor, the method comprising:

an operation of receiving vulnerability information to be analyzed and acquiring security patch information and a source code corresponding to the vulnerability information;

an operation of classifying a function including code lines deleted from the source code due to a security patch as a vulnerable function and a function including code lines added in the source code due to the security patch as a patch function, on the basis of the security patch information;

an operation of specifying each of the vulnerable function and the patch function as essential lines related to security, and specifying a line declaring variables used in any one first essential line, a line assigning a value to a variable, or a line determining whether or not to execute the first essential line as a first dependent line;

an operation of generating a semantic pair in which vulnerability is semantically connected by mapping the first essential line and the first dependent line;

an operation of collecting functions of all versions for at least one or more first functions selected to be analyzed among the vulnerable function and the patch function as an analysis target; and

an operation of determining whether each version is vulnerable on the basis of a ratio of the semantic pairs included in a function of each version.

2. The method according to claim 1, wherein the acquiring operation includes:

an operation of collecting a commit URL from a CVE detailed page of an NVD database including the vulnerability information and acquiring security patch information corresponding to the commit URL; and

an operation of acquiring information on the source code by copying an entire software repository from a software repository address included in the commit URL into a local environment.

3. The method according to claim 2, wherein the classifying operation includes:

an operation of extracting a source code file of a version before applying a patch and a source code file of a version after applying the patch from the software repository;

an operation of extracting a function including a code line where a + or − mark is assigned due to the security patch from each source code file using Universal Ctags; and

an operation of classifying a function including a code line where a − mark is assigned as a vulnerable function and a function including a code line where a + mark is assigned as a patch function among the extracted functions.

4. The method according to claim 1, wherein the operation of generating a semantic pair includes:

an operation of generating an essential line corresponding to a vulnerable function and a dependent line mapped thereto as a vulnerable semantic pair; and

an operation of generating an essential line corresponding to a patch function and a dependent line mapped thereto as a patch semantic pair.

5. The method according to claim 4, wherein the operation of generating a semantic pair includes:

an operation of removing a line in which the code added by the patch and the code deleted by the patch are syntactically identical, from the semantic pair; and

an operation of removing, when there is a dependent line included in both the vulnerable semantic pair and the patch semantic pair, the dependent line from the patch semantic pair to prioritize an evidence of vulnerability.

6. The method according to claim 4, wherein the determining operation includes:

an operation of calculating a vulnerability similarity as a proportion of all vulnerable semantic pairs included in a first version;

an operation of calculating a patch similarity as a proportion of all patch semantic pairs included in the first version; and

an operation of determining the first version as a vulnerable version when the vulnerability similarity is greater than a preset vulnerability threshold and the patch similarity is smaller than a preset patch threshold, for the first version of the first function.

7. The method according to claim 1, wherein the operation of collecting functions of all versions includes an operation of collecting functions of all versions for the first function by executing a command of “git log --follow -L:<function>:<file>” on the software repository and tracking history of change in the first function.

8. The method according to claim 1, wherein the determining operation includes:

an operation of removing annotations and spaces included in the code lines;

an operation of converting all characters in the code lines into lowercase; and

an operation of excluding a preset statement included in the line codes from the analysis target, in order to normalize the code lines of a function corresponding to each version of the first function before comparing on the basis of the semantic pairs for the functions of each version.

9. The method according to claim 1, wherein the first function includes a plurality of functions, and the determining operation includes:

an operation of deriving a vulnerability determination result for each function targeting the plurality of functions included in a specific software version;

an operation of classifying, when at least one among the plurality of functions is determined as vulnerable, the software version as a vulnerable version; and

an operation of generating final vulnerable version information in a form of a range of a start version and an end version of software by performing an operation of determining vulnerability of each function and classifying the vulnerable version on all versions of the software.

10. A software vulnerable version identification apparatus comprising:

a memory including instructions; and

a processor that performs a predetermined operation based on the instructions, wherein

the operation of the processor includes:

an operation of receiving vulnerability information to be analyzed and acquiring security patch information and a source code corresponding to the vulnerability information;

an operation of classifying a function including code lines deleted from the source code due to a security patch as a vulnerable function and a function including code lines added in the

source code due to the security patch as a patch function, on the basis of the security patch information;

an operation of specifying each of the vulnerable function and the patch function as essential lines related to security, and specifying a line declaring variables used in any one first essential line, a line assigning a value to a variable, or a line determining whether or not to execute the first essential line as a first dependent line;

an operation of generating a semantic pair in which vulnerability is semantically connected by mapping the first essential line and the first dependent line;

an operation of collecting functions of all versions for at least one or more first functions selected to be analyzed among the vulnerable function and the patch function as an analysis target; and

an operation of determining whether each version is vulnerable on the basis of a ratio of the semantic pairs included in a function of each version.

ABSTRACT

The present invention relates to the field of software security technology, and more particularly, to a technique of effectively identifying vulnerable versions of software by mapping code lines directly related to occurrence of vulnerability and code lines semantically connected thereto by means of data or control dependency as a semantic pair using security patch information.

FIG.1

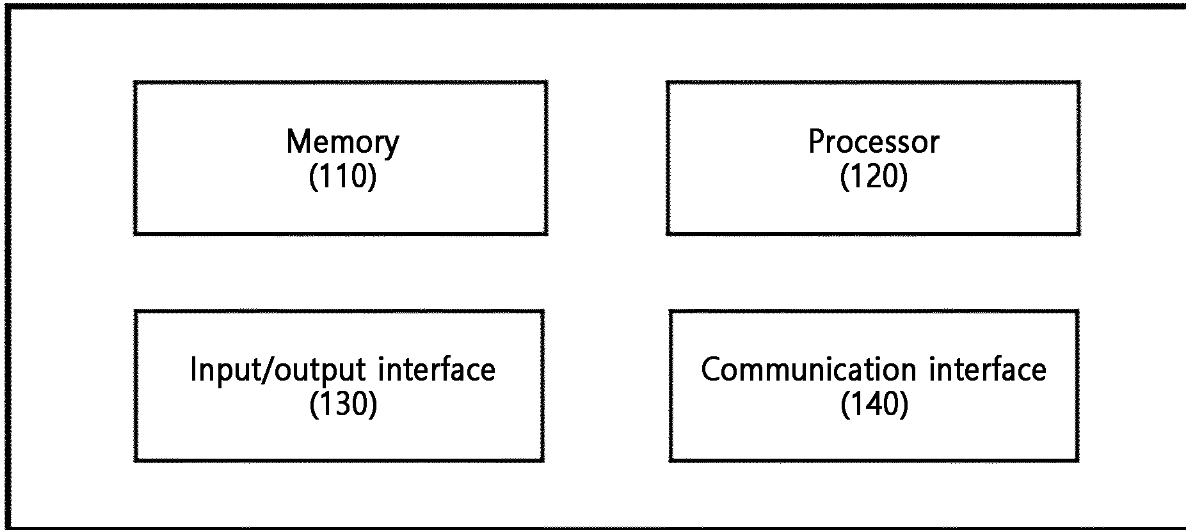


FIG.2

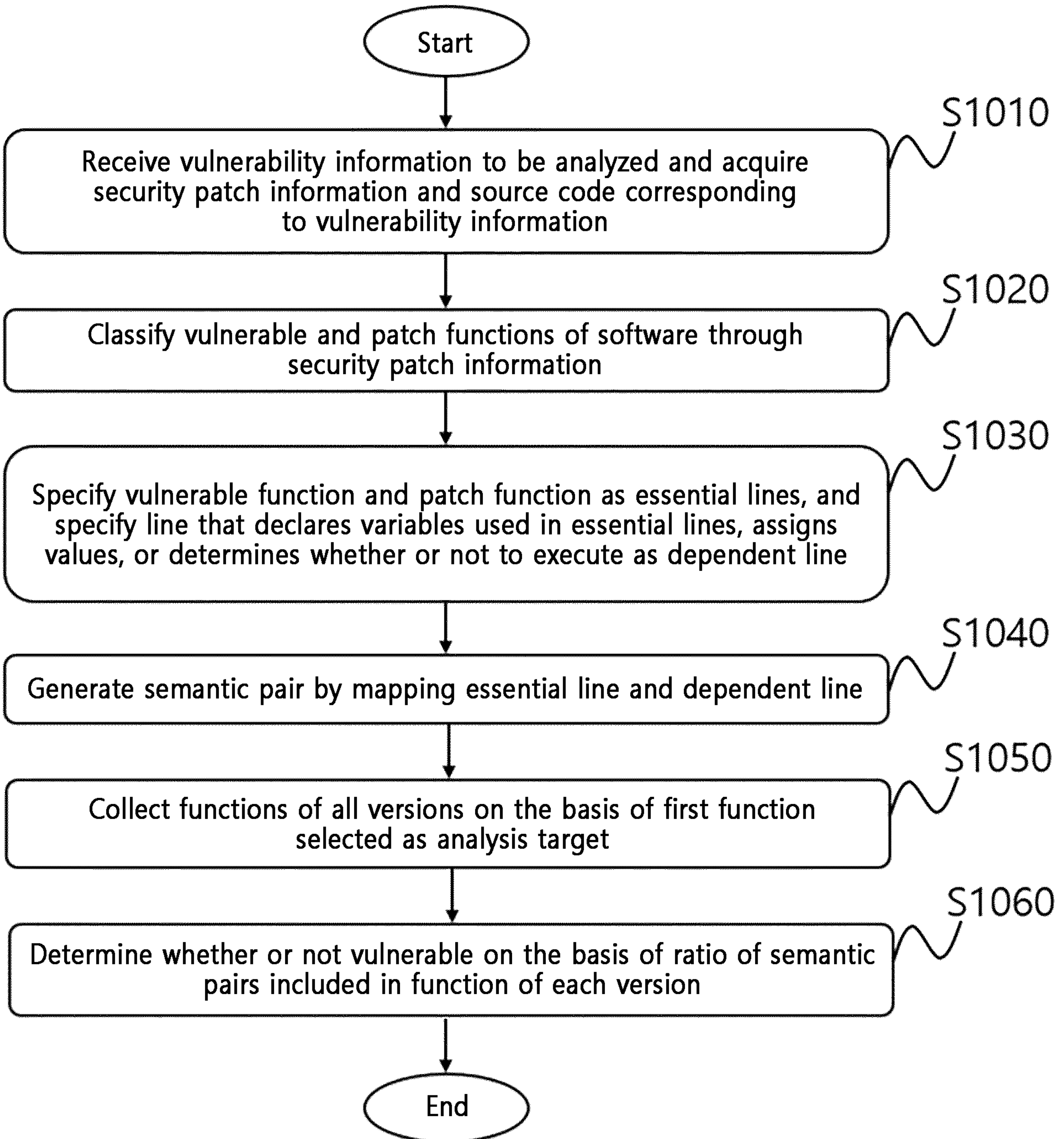


FIG.3

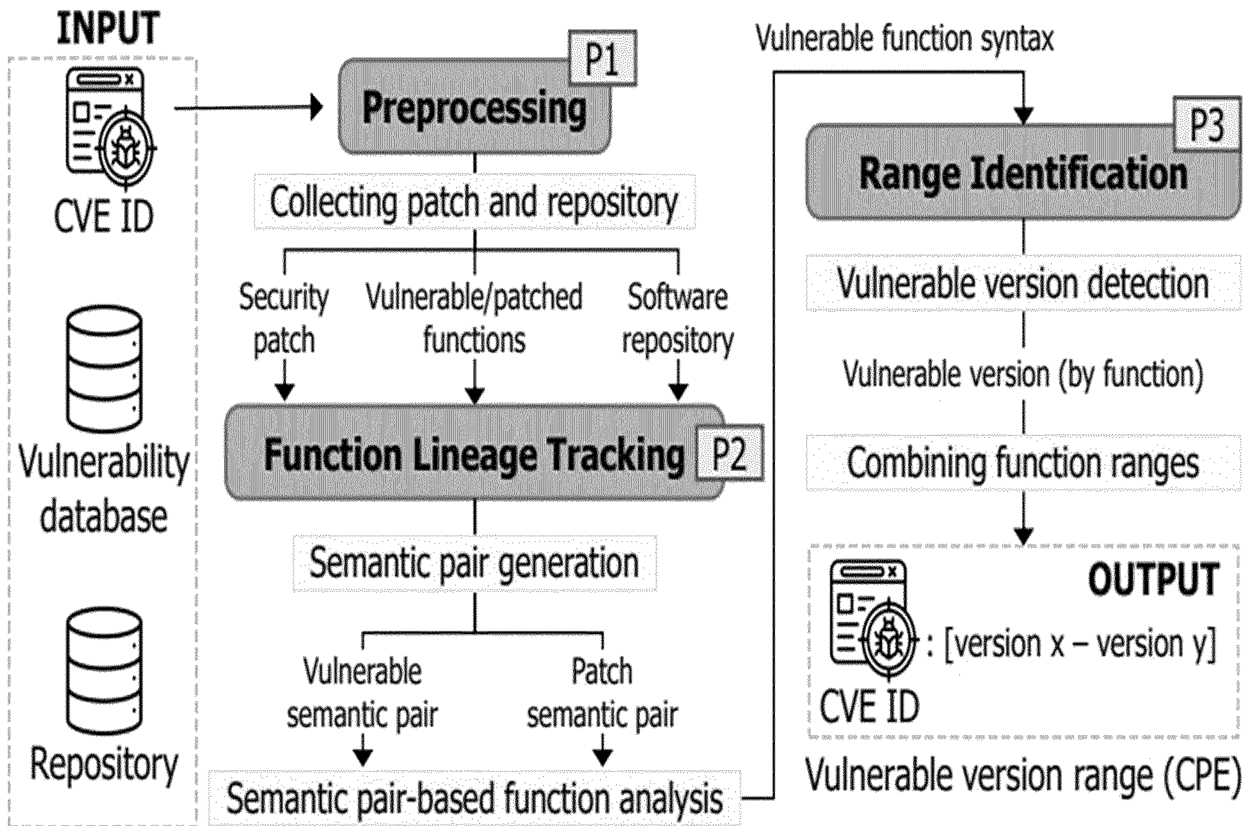


FIG.4

```
1 // CPE    : cpe:2.3:a:libtiff:libtiff:4.0.7:*:*:*:*:*:*:*
2 // Commit: 9a72a69e035ee70ff5c41541c8c61cd97990d018
3 // Path   : libtiff/tif_dirread.c
4 // Index : 3eec79c9d..570d0c327 100644
5 static void ChopUpSingleUncompressedStrip(TIFF* tif) {
6 ...
7   uint64 stripbytes;
8   uint64 rowblockbytes;
9 ...
10  if (rowblockbytes > STRIP_SIZE_DEFAULT) {
11    stripbytes = rowblockbytes;
12 ...
13  if (rowsperstrip >= td->td_rowsperstrip) {
14    return;
15 - nstrips64 = TIFFhowmany_64(bytecount, stripbytes);
16 - if ((nstrips64==0)||(nstrips64>0xFFFFFFFF))
17 -   return;
18 - nstrips32 = (uint32)nstrips64;
19 + nstrips = TIFFhowmany_32(td->td_imagelength, rowsperstrip);
20 + if( nstrips == 0 )
21 +   return;
```

FIG.5

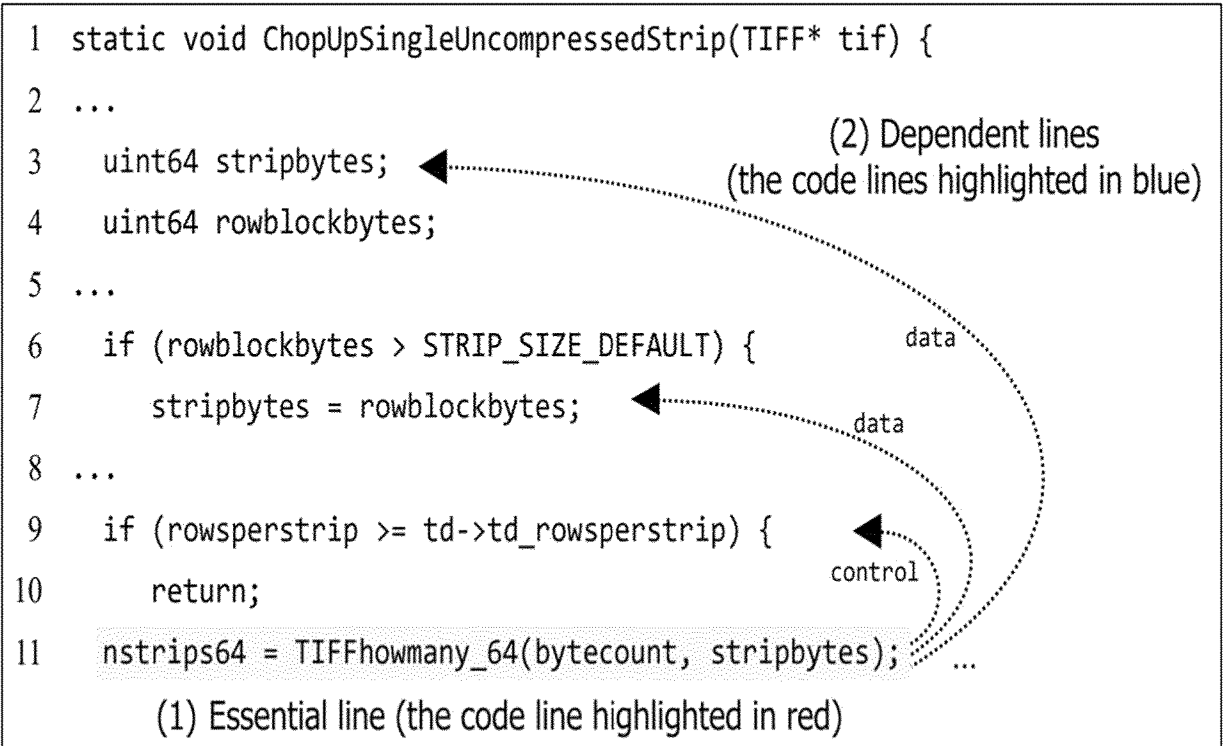


FIG.6

Validation method	#CVEs	NVD			VOFinder [11]		
		Precision	Recall	F1 score	Precision	Recall	F1 score
Reproduction	103	0.8627	0.8712	0.8669	**0.9973**	0.6645	0.7976
Author sites	54	0.8913	0.8612	**0.8759**	0.9129	0.5158	0.6592
Code review	1,345	0.8126	0.8566	0.8340	0.9450	0.5647	0.7070
Total	1,502	0.8189	0.8577	0.8379	0.9474	0.5698	0.7116

Validation method	#CVEs	V-SZZ [10]			CLOVERY		
		Precision	Recall	F1 score	Precision	Recall	F1 score
Reproduction	103	0.9781	0.6753	0.7990	0.8837	**0.9759**	**0.9275**
Author sites	54	**0.9212**	0.7292	0.8140	0.7302	**0.9245**	0.8160
Code review	1,345	0.9182	0.6258	0.7443	**0.9781**	**0.9894**	**0.9837**
Total	1,502	0.9224	0.6329	0.7507	**0.9627**	**0.9862**	**0.9743**

FIG.7

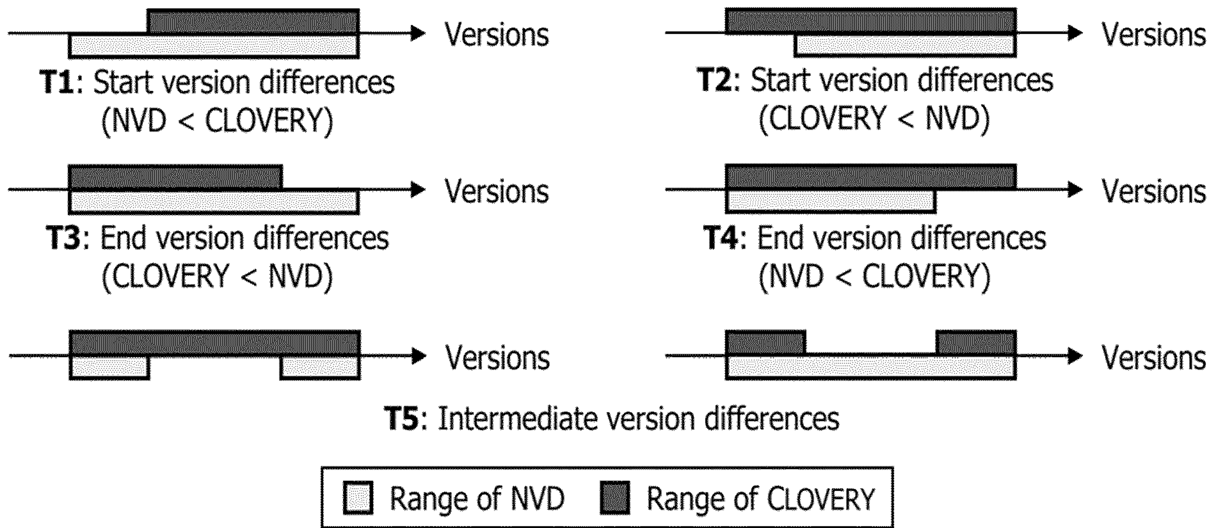


FIG.8

Type	Description	#CVEs	Ratio
T1	Start `diff`. (NVD < CLOVERY)	578	58.62% (578/986)
T2	Start `diff`. (CLOVERY < NVD)	325	32.96% (325/986)
T3	End `diff`. (CLOVERY < NVD)	194	19.68% (194/986)
T4	End `diff`. (NVD < CLOVERY)	357	36.21% (357/986)
T5	Intermediate `diff`.	184	18.66% (184/986)

FIG.9