(12) STANDARD PATENT (11) Application No. **AU 2004232058 B2**
(19) AUSTRALIAN PATENT OFFICE

(54) Title
Method and system for detecting vulnerabilities in source code

(51) International Patent Classification(s)
G06F 19/00 (2006.01) ***G06F 11/36*** (2006.01)
G06F 9/44 (2006.01) ***G06F 12/14*** (2006.01)
G06F 11/30 (2006.01)

(21) Application No: **2004232058** (22) Date of Filing: **2004.04.15**

(87) WIPO No: **WO04/095176**

(30) Priority Data

(31) Number	(32) Date	(33) Country
10/825,007	**2004.04.15**	**US**
60/464,019	**2003.04.18**	**US**

(43) Publication Date: **2004.11.04**
(44) Accepted Journal Date: **2010.05.27**

(71) Applicant(s)
International Business Machines Corporation

(72) Inventor(s)
Rose, Larry;Dahany, John J.;Rehbein, Chris;Berg, Ryan James;Peyton, John;Gottlieb, Robert

(74) Agent / Attorney
Davies Collison Cave, 1 Nicholson Street, Melbourne, VIC, 3000

(56) Related Art
"A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities", WAGNER et al., Proc. of the Network and Distributed System Security Symposium, February 2000
EP 1079303 A (HEWLETT PACKARD CO., US.) 29 February 2001

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
4 November 2004 (04.11.2004)

PCT

(10) International Publication Number
WO 2004/095176 A2

(51) International Patent Classification[7]: G06F

(21) International Application Number:
PCT/US2004/011625

(22) International Filing Date: 15 April 2004 (15.04.2004)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/464,019 18 April 2003 (18.04.2003) US
10/748,831 30 December 2003 (30.12.2003) US

(71) Applicant *(for all designated States except US)*: **OUNCE LABS, INC.** [US/US]; 230 Third Avenue, Prospect Place, Waltham, MA 02451 (US).

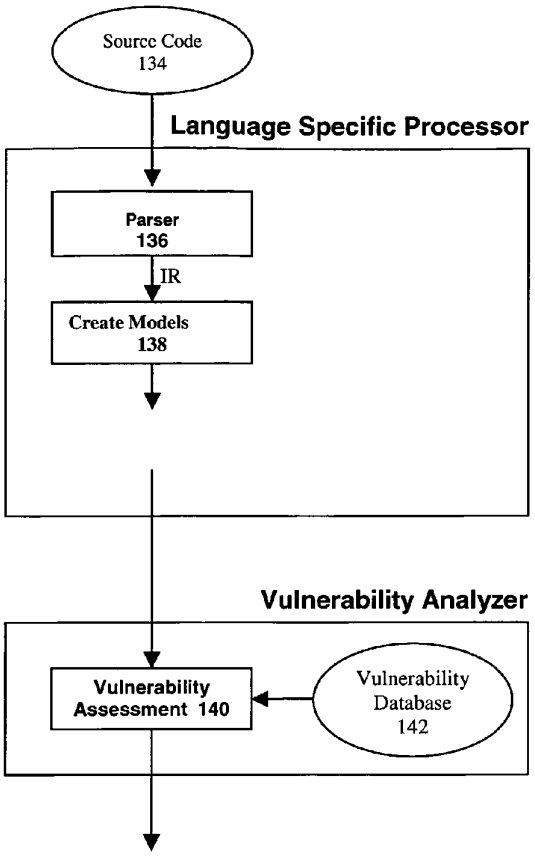
(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: **BERG, Ryan, James** [US/US]; 608 Dutton Road, Sudbury, MA 01776-1945 (US). **ROSE, Larry** [US/US]; 257 Boston Road, Chelmsford, MA 01824-4803 (US). **PEYTON, John** [US/US]; 156 Medford Street, Arlington, MA 02474-3112 (US). **DAHANY, John, J.** [US/US]; 60 Indian Lane, Canton, MA 02021-3516 (US). **GOTTLIEB, Robert** [US/US]; 52 Vine Brook Road, Westford, MA 01886-4218 (US). **REHBEIN, Chris** [US/US]; 3 Fifield Street, Watertown, MA 02472-2703 (US).

(74) Agents: **DICHIARA, Peter, M.** et al.; Hale and Dorr LLP, 60 State Street, Boston, MA 02109 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG,

[Continued on next page]

(54) Title: METHOD ANS SYSTEM FOR DETECTING VULNERABILITIES IN SOURCE CODE



(57) Abstract: A method and system of detecting vulnerabilities in source code. Source code is parsed into an intermediate representation. Models (e.g., in the form of lattices) are derived for the variables in the code and for the variables and/or expressions used in conjunction with routine calls. The models are then analyzed in conjunction with pre-specified rules about the routines to determine if the routine call posses one or more of pre-selected vulnerabilities.

WO 2004/095176 A2

PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

METHOD AND SYSTEM FOR DETECTING VULNERABILITIES IN SOURCE CODE

Background of the Invention

Field of the Invention

[0001] The invention relates to computer system security and more particularly to a method and system that detects computer source code vulnerabilities, which may pose security risks.

Discussion of Related Art

[0002] One of the problems associated with developing computer programs is the difficulty in detecting *"vulnerabilities"* in the programs. As used herein, the term *"vulnerability"* refers to a section of user source code which, when executed, has the potential to allow external inputs to cause improper or undesired execution. Typical vulnerabilities include buffer overflow; race conditions; and privilege escalation, each of which poses a vulnerability to the desired, controlled execution of the program. Reviewing source code for vulnerabilities is a difficult, time-consuming process. It requires a full understanding of all potential vulnerabilities, how to spot them, and how to fix them.

[0003] Prior methods of detecting vulnerabilities in source code include conducting a lexical analysis of the source code. This involves conducting a search of well-known vulnerabilities and pointing them out as potential vulnerabilities. A problem with this method is that it generates too many false positives. Another method involves conducting a manual, line-by-line analysis of the code. However, this method is very labor intensive.

Summary of the Invention

[0004] The invention provides a method and system for detecting vulnerabilities in source code.

[0005] According to the present invention there is provided a computer implemented method of detecting vulnerabilities in a pre-existing source code listing, said source code listing having a listed sequence of expressions, each expression including a set of operands and operators to transform values of the operands, said listed sequence of

expressions having an inherent control flow indicative of the run-time execution of the expressions and an inherent data flow indicative of the run-time transformations of operand values, said source code listing further having routine calls, said routine calls including arguments with which to invoke a routine, said source code listing being stored in a computer-readable medium, said computer implemented method comprising the acts of:

executing computer instructions to analyse the source code listing to create computer models of the operands, said models each including a corresponding initial set of information to represent the range of values for said operand, and said models being transformed, in response to analysis of the source code, to have a transformed range of values to correspond to the operand transformations expressed in the source code listing, said models being stored in computer memory, and wherein each model specifies pre-determined characteristics about and possible values for each operand as a result of said source code expressions;

executing computer instructions to use said operand models to create models of said arguments to routine calls, said argument models being stored in computer memory;

executing computer instructions to use said argument models in conjunction with pre-specified criteria for the corresponding routine calls to determine whether the routine calls possess vulnerabilities as a consequence of the arguments and known routine behaviour; and

generating a report that identifies the vulnerabilities said report being viewable by a developer-user, so the developer-user may address the vulnerabilities identified in the report by modifying the source code listing if necessary.

[0006] The invention also provides a computer implemented method of detecting vulnerabilities in a pre-existing source code listing, said source code listing having a listed sequence of expressions, each expression including a set of operands and operators to transform values of the operands, said listed sequence of expressions having an inherent control flow indicative of the run-time execution of the expressions and an inherent data flow indicative of the run-time transformations of operand values, said source code listing further having routine calls, said routine calls including arguments with which to invoke a routine said source code listing being stored in a computer-readable medium, said

computer implemented method comprising the acts of:

executing computer instructions to analyse the source code listing to create computer models of arguments to routine calls in the source code listing, said argument models being stored in computer memory, said argument models each including a corresponding initial set of information to represent the range of values for said argument, and said argument models being transformed, in response to analysis of the source code, to have a transformed range of values to correspond to the transformations expressed in the source code listing, said models being stored in computer memory, and wherein each argument model specifies pre-determined characteristics about and possible values for each argument as a result of said source code expressions;

executing computer instructions to use said argument models in conjunction with pre-specified criteria for the corresponding routine calls to determine whether the routine calls possess vulnerabilities as a consequence of the arguments and the routine behaviour; and

generating a report that identifies the vulnerabilities said report being viewable by a developer-user, so the developer-user may address the vulnerabilities identified in the report by modifying the source code listing if necessary.

[0007] The invention also provides a computer implemented utility for detecting vulnerabilities in a pre-existing source code listing, said source code listing having a listed sequence of expressions, each expression including a set of operands and operators to transform values of the operands, said listed sequence of expressions having an inherent control flow indicative of the run-time execution of the expressions and an inherent data flow indicative of the run-time transformations of operand values, said source code listing further having routine calls, said routine calls including arguments with which to invoke a routine, said source code listing being stored in a computer-readable medium, said utility comprising a computer-readable medium encoded with:

executable instructions for analysing the source code listing to create computer models of the operands, said models each including a corresponding initial set of information to represent the range of values for said operand, and said models being transformable, in response to analysis of the source code, to have a transformed range of values to correspond to the operand transformations expressed in the source code listing,

said models, storable in a computer memory, and wherein each model specifies pre-determined characteristics about and possible values for each operand as a result of said source code expressions;

executable instructions for using the operand models to create models of arguments to routine calls in the source code listing, said argument models being stored in computer memory; and

executable instructions for using the argument models in conjunction with pre-specified criteria for the corresponding routine calls to determine whether the routine calls possess vulnerabilities as a consequence of the arguments and known routine behaviour; and

executable instructions for generating a report that identifies the vulnerabilities said report being viewable by a developer-user, so the developer-user may address the vulnerabilities identified in the report by modifying the source code listing if necessary.

[0007A] The invention also provides a computer implemented method of detecting vulnerabilities in a pre-existing source code listing, said source code listing having a listed sequence of expressions, each expression including a set of operands and operators to transform values of the operands, said listed sequence of expressions having an inherent control flow indicative of the run-time execution of the expressions and an inherent data flow indicative of the run-time transformations of operand values, said source code listing further having routine calls, said routine calls including arguments with which to invoke a routine, said arguments including expression-references and operand-references to computer files, said source code listing being stored in a computer-readable medium, said computer implemented method comprising the acts of:

executing computer instructions to analyse the source code listing to create computer models of said control flow to indicate the run-time sequence in which routine calls will be invoked and to create computer models of said arguments for the routine calls using a flow insensitive analysis, wherein said control flow models include a control flow graph, and wherein each of said models of arguments is stored in computer memory and specifies pre-determined characteristics about and a range of possible values for the corresponding argument as a result of said source code expressions;

executing computer instructions to use said computer models of said control flow

in order to determine a run-time sequence of execution of a pair of routine calls by traversing the control flow graph backwards, said pair of routine calls having a first routine call and second routine call in which execution of the first routine call precedes execution of said second routine call;

executing computer instructions to determine whether a second routine to be executed has a second argument with a corresponding modelled range of possible of values that includes a reference to a file that is also within a corresponding modelled range of possible values for a first argument of the first routine to be executed, so that a possibility of the first and second arguments referring to the same file is determined even when said expression-references and operand-references to computer files for said first and said second arguments are lexically dissimilar;

executing computer instructions to identify said sequence as a race condition vulnerability; and

generating a report that is viewable by a user and that identifies the race condition vulnerabilities, so the user may modify the source code listing to address the vulnerability if desired.

[0007B] The invention also provides a system for detecting vulnerabilities in a pre-existing source code listing, said source code listing having a listed sequence of expressions, each expression including a set of operands and operators to transform values of the operands, said listed sequence of expressions having an inherent control flow indicative of the run-time execution of the expressions and an inherent data flow indicative of the run-time transformations of operand values, said source code listing further having routine calls, said routine calls including arguments with which to invoke a routine, said arguments including expression-references and operand-references to computer files, said source code listing being stored in a computer-readable medium, said system comprising:

computer-executable instructions on a computer-readable medium to analyse the source code listing to create computer models of said control flow to indicate the run-time sequence in which routine calls will be invoked and to create computer models of said arguments for the routine calls using a flow insensitive analysis, wherein said control flow models include a control flow graph, and wherein each of said models of arguments is stored in computer memory and specifies pre-determined characteristics about and a range

of possible values for the corresponding argument as a result of said source code expressions;

computer-executable instructions on a computer-readable medium to use said computer models of said control flow to determine a run-time sequence of execution of a pair of routine calls by traversing the control flow graph backwards, said pair of routine calls having a first routine call and second routine call in which execution of the first routine call precedes execution of said second routine call;

computer-executable instructions on a computer-readable medium to determine whether a second routine to be executed has a second argument with a corresponding modelled range of possible of values that includes a reference to a file that is also within a corresponding modelled range of possible values for a first argument of the first routine to be executed, so that a possibility of the first and second arguments referring to the same file is determined even when said expression-references and operand-references to computer files for said first and said second arguments are lexically dissimilar;

computer-executable instructions on a computer-readable medium to identify said sequence as a race condition vulnerability; and

computer-executable instructions on a computer-readable medium to generate a report that is viewable by a user and that identifies the race condition vulnerabilities, so the user may modify the source code listing to address the vulnerability if desired.

[0007C] The invention also provides a computer implemented method of detecting vulnerabilities in a pre-existing source code listing, said source code listing having a listed sequence of expressions, each expression including a set of operands and operators to transform values of the operands, said listed sequence of expressions having an inherent control flow indicative of the run-time execution of the expressions and an inherent data flow indicative of the run-time transformations of operand values, said source code listing being expressed in multiple programming languages, said source code listing further having routine calls including arguments with which to invoke a routine, said source code listing being stored in a computer-readable medium, said computer implemented method comprising the acts of:

executing computer instructions to create a single intermediate representation of said source code listing regardless of programming language by parsing said source code

listing;

executing computer instructions to provide a database having computer-readable records associated with pre-identified routines, each record specifying an argument vulnerability condition for arguments of said corresponding pre-identified routine, that, if satisfied, presents a vulnerability during execution of said routine;

executing computer instructions to statically analyse said intermediate representation of said source code listing to create computer models of the operands, said models representing expected transformation of the operands from run time execution of a computer program created by compilation of said source code listing, said models being stored in computer memory;

executing computer instructions said operand models to create models of said arguments to routine calls, said argument models being stored in computer memory;

executing computer instructions to retrieve, from said database, a record corresponding to routine calls represented in said intermediate representation;

executing computer instructions to compare said argument models with said condition specified in the retrieved record to determine whether the routine call possesses vulnerabilities as a consequence of the arguments; and

generating a report that identifies detected vulnerabilities, said report being viewable by a developer-user, so the developer-user may address the vulnerabilities identified in the report by modifying the source code listing if necessary.

[0007D] The invention also provides a method of detecting privilege escalation vulnerabilities in a pre-existing source code listing, said source code listing having a listed sequence of expressions, each expression including a set of operands and operators to transform values of the operands, said source code listing further having routine calls, said routine calls including arguments with which to invoke a routine, said source code listing being stored in computer readable medium having computer executable instructions, wherein a privilege escalation vulnerability is an uncontrolled escalation of system privileges that allows unauthorized access to system resources, the method comprising:

providing a list specifying routines that potentially cause privilege escalation vulnerabilities;

providing pre-specified ranges of values for arguments of routines in the list that

cause privilege escalation vulnerabilities;

analysing the source code listing to identify calls to routines specified in the list;

analysing the source code listing to semantically analyse arguments of the identified routine calls to determine routine calls that possess privilege escalation vulnerabilities using the pre-specified ranges of values;

wherein semantically analysing the arguments of the identified routine calls comprises analysing the source code listing to create computer models of the arguments, each model specifying a range of values that each corresponding argument can take when the source code listing is executed; and

generating a report that identifies the vulnerabilities.

Brief Description of The Drawings

[0008] In the Drawings,

Fig. 1 shows a flow chart of the steps of the system and method of one embodiment of the present invention;

Fig. 2 shows an example of an integral lattice;

Fig. 3 shows an example of a memory size lattice;

Fig. 4 shows an example of a data size lattice;

Fig. 5 shows an example of a null terminated lattice;

Fig. 6 shows an example of a memory location lattice;

Fig. 7 shows an example of a string value lattice;

Fig. 8 shows an example of a data origin lattice;

Fig. 9 shows a flow chart of the steps performed by the Flow-Insensitive Analysis according to one embodiment of the present invention;

Figs. 10A-B show a flow chart of the steps performed in processing expressions according to the Flow-Insensitive Analysis of one embodiment of the invention;

Figs. 11A-B shows a flow chart of the steps performed in processing expressions according to the Call Site Analysis according to one embodiment of the present invention;

Fig. 12 shows a control flow graph according to one embodiment of the present invention; and.

Fig. 13 shows a flow chart of the steps of the system and method of one embodiment of the present invention.

Detailed Description

[0009] Preferred embodiments of the present invention provide a method and system for detecting vulnerabilities in source code. The term "vulnerability," as used herein, refers to a section of user source code which, when executed, has the potential to allow external inputs to cause improper or undesired execution.

[0010] Preferred embodiments of the present inventions provide a method and system for detecting vulnerabilities such as buffer overflow, race condition and privilege escalation.

[0011] Figure 13 is a flow chart depicting exemplary logic for analyzing computer programs to detect vulnerabilities such as buffer overflow, race conditions and privilege escalation. The processing has two basic blocks: language specific processing and vulnerability analysis. The language specific processing analyzes the source code and creates models. The language specific processing begins with a language parser 136 receiving the source code 134 to be analyzed and creating an intermediate representation (IR) therefrom. IRs are known in the art and thus the parsing logic is not described here.

Models 138 are created to describe certain characteristics of the source code, and the models are used in conjunction with a vulnerability database 142 in a vulnerability assessment 140 to determine whether a vulnerability exists.

[0012] Figure 1 is a flow chart depicting exemplary logic for analyzing computer programs for buffer overflow vulnerabilities according to certain embodiments of the invention. The processing has two basic blocks: language specific processing and vulnerability analysis.

[0013] The language specific processing analyzes the source code and models the arguments used to call select procedures, functions or routines. The models use a unique structure called a "vulnerability lattice." The vulnerability lattice is used to

specify, certain relevant information about the argument (whether a variable or expression) such as its memory size, its memory type, etc. This lattice specification is language independent.

[0014] The vulnerability analysis uses the vulnerability lattices and other information to analyze the affects of such routine calls with such arguments. This analysis is language independent. The analysis applies rules to determine whether a given routine call in the source code, including the arguments used in such call, pose an inherent vulnerability or risk for certain types of errors. For example, the analysis may determine that a certain routine call with certain arguments at a given location in the source code creates a potential for a buffer overflow error.

[0015] Both the language specific processing and the vulnerability assessment utilize lattice structures to model and analyze the variables and expressions that may be used as arguments to routines. By way of background, a lattice represents a refinement of knowledge about the value of an entity. Figure 2 shows an example of an integral lattice 22 for an integer value. The top value ($\top$) at the top of the lattice represents no knowledge of the value. The bottom value ($\bot$) at the bottom of the lattice represents an unknown value (i.e., no resolution about which of the possible values should be applied). The value(s) between the top value and the bottom value represent the possible value(s) of the entity. In the integral lattice 22 shown in Figure 2, the integers 0, 1, 2, 3, 4 and 5 are the possible values for the entity.

Language Specific Processing to Create Vulnerability Lattices for Arguments to Select Routines

[0016] The language specific processing begins with a language parser 12 receiving the source code 10 to be analyzed and creating an intermediate representation (IR) therefrom.

[0017] A flow-insensitive analysis 14 analyzes the IR and derives models about each variable in the code. These models are specified in lattice form and called vulnerability lattices. (Lattices in general are known.) Under preferred embodiments a vulnerability lattice (sometimes referred to as an "expression lattice" as well in the paragraphs below) includes a number of other lattices to describe important characteristics of a variable or expression (depending on whether the vulnerability lattice is associated with a variable or expression). More specifically, the vulnerability lattices provide information about the following:

- memory size;
- data size;
- whether data is null terminated;
- the kind of memory contained in a block of memory;
- the constant string value or values for a block of memory; and
- the origin of data.

[0018] When determining how a lattice should be set or modified the flow-insensitive analysis logic applies pre-determined merger rules for the various lattice types. This is used, for example, when analyzing expressions.

[0019] The flow-insensitive analysis logic also utilizes integral lattices to describe (again in lattice form) integral type variables.

[0020] Figure 3 depicts an example of a memory size lattice 24. Memory size lattice 24 is a lattice consisting of the values high, low, and a pair of non-negative integral values, indicating the possible range of sizes of a block of memory, either directly or referenced via a pointer. This lattice may be used to determine if certain memory operations will overflow the available memory. The merge rules for the memory size lattice 24 are as follows:

- a merge of a high value ($\top$) and any other value will result in the other value;
- a merge of a low value ($\bot$) and any other value will result in a low value; and
- a merge of two memory range lattice values will result in the following:
  - range maximum $\leftarrow$ $range_1$ maximum $\lceil$ $range_2$ maximum ($\lceil$ is the "maximum of" operator)
  - range minimum $\leftarrow$ $range_1$ minimum $\lfloor$ $range_2$ minimum ($\lfloor$ is the "minimum of" operator)

[0021] For example, an array declared in c or c++ as

```
char a[100];
```

[0022] would have a size of 100 bytes, that being the size of 1 entry (1 byte) multiplied by the number of elements in the array (100).

[0023] As another example, a memory size lattice representing a range of size values could be useful:

```
char a[100];

char b[200];

char *c = (i == 0) ? a : b;
```

[0024] The size of the block of memory pointed to by the variable `c` in this case could be either 100 bytes or 200 bytes, depending on whether the array `a` or the array `b` is selected, which in turn depends on whether another variable `i` is 0. The memory size lattice result for this variable would specify a maximum size of 200 and a minimum of 100 bytes.

[0025] Figure 4 depicts an example of a data size lattice 26. A data size lattice indicates the possible range of sizes of the known data within a block of memory, either directly or referenced via a pointer. This lattice may be used to determine if certain memory operations will overflow the available memory. In particular, it is generally used to indicate the size of a null terminated string, which may be shorter than the block of memory in which it is contained. The merge rules for the data size lattice 26 are as follows:

- a merge of a high value ($\top$) and any other value will result in the other value;
- a merge of a low value ($\bot$) and any other value will result in a low value; and
- a merge of two memory range lattice values will result in the following:
  - range maximum $\leftarrow$ $range_1$ maximum $\lceil$ $range_2$ maximum
  - range minimum $\leftarrow$ $range_1$ minimum $\lfloor$ $range_2$ minimum

[0026] Figure 5 depicts an example of a null terminated lattice 28. A null terminated lattice indicates whether or not the data is known to be null terminated, e.g., has a 0 value as the last entry to indicate the end of the data. It is typically used in connection with string structures. The range of data includes specifying that it is null terminated or is not null terminated. The merge rules for the null terminated lattice are as follows:

- a merge of a high value ($\top$) and any other value will result in the other value;
- a merge of a low value ($\bot$) and any other value will result in a low value;
- a merge of two identical non-high, non-low lattice values will result in the same lattice value; and
- a merge of two different non-high, non-low lattice values will result in the low ($\bot$) lattice value.

[0027] Figure 6 depicts an example of a memory location lattice 30. A memory location lattice indicates the kind of memory that the block of memory is contained within, e.g., stack memory, heap memory, static memory, and constant memory. Other kinds of memory may also be specified. The merge rules for the memory location lattice 30 are as follows:

- a merge of a high value ($\top$) and any other value will result in the other value;
- a merge of a low value ($\bot$) and any other value will result in a low value;
- a merge of two identical non-high, non-low lattice values will result in the same lattice value; and
- a merge of two different non-high, non-low lattice values will result in the low ($\bot$) lattice value.

[0028] Figure 7 depicts an example of a string value lattice 32. A string value lattice indicates the constant string value or values for the block of memory. The merge rules for a string value lattice are as follows:

- a merge of a high value ($\top$) and any other value will result in the other value;
- a merge of a low value ($\bot$) and any other value will result in a low value;
- a merge of two identical constant strings will result in that constant string as the lattice value; and
- a merge of two different constant strings will result in the low ($\bot$) lattice value.

[0029] Figure 8 depicts an example of a data origin lattice 34. A data origin lattice indicates the origin of the data, e.g., specifying that the data is internally generated (relative to the analyzed routine) or whether it is externally generated. Data of an unknown origin will have the low value. The merge rules for a data origin lattice are as follows:

- a merge of a high value ($\top$) and any other value will result in the other value;
- a merge of a low value ($\bot$) and any other value will result in a low value;
- a merge of two identical non-high, non-low lattice values will result in the same lattice value; and
- a merge of two different non-high, non-low lattice values will result in the low ($\bot$) lattice value.

[0030] A "vulnerability lattice" represents the attributes of a non-integral type variable (or expression). Under preferred embodiments, it incorporates the memory size lattice 24, data size lattice 26, null terminated lattice 28, memory location lattice 30, string value lattice 32, and data origin lattice 34.

[0031] Figure 9 shows a flow chart of the steps performed in the flow-insensitive analysis 14 of preferred embodiments of the invention. The flow-insensitive analysis 14 derives a vulnerability lattice for each non-integral type variable or expression and an integral lattice for each integral type variable or expression. The term expression lattice, as used herein, means either a vulnerability lattice, in the case of a non-integral

variable or expression or an integral lattice, in the case of an integral type variable or expression.

[0032] The flow begins with an initial test 36 to determine if the variable being analyzed is an array or structure. If so, the variable is associated with a vulnerability lattice. A test is then made in step 38 to determine if the variable is visible to other routines or passed into other routines as an argument.

[0033] If the variable is visible to other routines or passed into other routines as an argument, the vulnerability lattice for the variable is set, in step 40, to specify a memory size lattice having a value set to the size of the variable. All other values of the vulnerability lattice are set to low in step 40. Though not shown in the flow chart, if the variable is a constant initialized variable, the data size lattice, null terminated lattice, and string value lattice are set to indicate the initialized value of the variable.

[0034] If the variable is not visible to other routines or not passed into other routines as an argument, the memory size lattice is set to a value the size of the variable. All other values in the vulnerability lattice are set, in step 42, to high.

[0035] If the results of step 36 are "false" (meaning that the variable is not an array or structure), the flow proceeds to step 44. In step 44, a test is performed to determine whether the variable being analyzed is a pointer. If so, the logic proceeds to step 46 to determine if the pointer variable is visible to other routines, or if it is passed in to other routines as an argument.

[0036] If the variable is visible to other routines or passed into other routines as an argument, the pointer variable is associated with a vulnerability lattice and all values of the vulnerability lattice are set to low in step 49.

[0037] If the variable is not visible to other routines or not passed into other routines as an argument, the pointer variable is associated with a vulnerability lattice and all values of the vulnerability lattice are set to high in step 48.

[0038] If the results of step 44 are "false" (meaning that the variable is not an array or structure or pointer), the flow proceeds to step 50. In step 50 a test is performed to determine whether the variable being analyzed is an integral type variable. Integral type variables are associated with an integral lattice. If so, the logic proceeds to step 52 to determine if the integral variable is visible to other routines, or if it is passed in to other routines as an argument.

[0039] If the variable is visible to other routines or passed into other routines as an argument, it is associated with an integral lattice with all values set to low in step 56.

[0040] If the variable is not visible to other routines or not passed into other routines as an argument, the value in the integral lattice is set to high in step 54.

[0041] After the flow-insensitive analysis 14 derives a vulnerability lattice or integral lattice for each variable in the routine, the flow-insensitive analysis 14 visits each statement in the routine. The visits may be made in any order. Each expression within a statement is visited in such an order that before the expression is processed, all the expressions given as input (i.e., dependencies) to that expression are processed. For example, in the expression

```
a = (b + c) + d;
```

the partial, or sub-expressions `b` and `c` must be processed before the expression `(b + c)` is processed. Similarly, the sub-expressions `(b + c)` and `d` must be processed before the expression `(b + c) + d` is processed.

[0042] Figures 10A-B show a flow chart of the flow-insensitive analysis logic of preferred embodiments for processing each expression in a routine. The flow begins with an initial test 58 to determine if the expression being analyzed is for an address of a variable. If so, in step 60, a test is made to determine if that variable is to an array or structure or to determine if the variable is a constant string. If so, in step 64, a vulnerability lattice is associated with that expression and its memory size lattice is set to the size of the variable, and its memory location lattice is set to the kind of memory of the variable referenced. If the variable has a constant (const) attribute and it is a string, the data size lattice is set to the size of the string and the null terminated lattice is set to null terminated. The string value lattice is set to the value of the string. The data origin lattice is set to specify that the data origin is internal. If the expression is referring to the address of a variable but the variable is not a constant string, then in step 62 a vulnerability lattice is associated with that expression and its memory size lattice set to the size of the variable, and its memory location lattice is set to the kind of memory of the variable referenced. The other lattice entries are set to the low value. In addition, since the variable is address exposed (i.e., a pointer to it exists and it can potentially be modified by any pointer write to the pointer), in step 62 the vulnerability lattice whose address was taken has its data size lattice, null terminated lattice, string value lattice, and data origin lattice set to low (with the memory size lattice and memory location lattice remaining unchanged).

[0043] If the results of step 58 are "false" (meaning that the expression is not referring to the address of a variable), the flow proceeds to step 66. In step 66, a test is made to determine if the expression is for a value of a variable. If so, in step 68, a vulnerability lattice is associated with the expression and all lattice entries are set to low.

[0044] If the results of step 66 are "false" (meaning that the expression is not referring to the address or value of a variable), the flow proceeds to step 70. In step 70, a test is made to determine if the expression is for a constant string. If so, in step 72 a vulnerability lattice is associated with the expression and its memory size lattice is set to the size of the constant string, including null termination byte; its data size lattice is set to the size of the constant string, including the null termination byte; its null termination lattice is set to indicate that it is null terminated; its memory location lattice is set to indicate constant memory; its string value lattice is set to the contents of the string; and its data origin lattice is set to internal.

[0045] If the results of step 70 are "false" (meaning that the expression is not referring to the address or value of a variable and does not refer to a constant string), the flow proceeds to step 74. In step 74, a test is made to determine if the expression is for an integral constant (i.e., an integer). If so, in step 76 an integral lattice is associated with the expression, and its value is set to the integer value.

[0046] If the results of step 74 are "false" (meaning that the expression is not referring to the address or value of a variable and does not refer to a constant string or an integral constant), the flow proceeds to step 78. In step 78, a test is made to determine if the expression is a "question mark/colon operation." A question mark/colon operation is of the form $\langle expression_1 \rangle$? $\langle expression_2 \rangle$: $\langle expression_3 \rangle$. If so, in step 80 a vulnerability lattice is associated with the expression and its lattice entries are set to the results from merging the vulnerability lattices of $\langle expression_2 \rangle$ and $\langle expression_3 \rangle$ (which have been set previously).

[0047] If the results of step 78 are "false", the flow proceeds to step 82. In step 82, a test is made to determine if the expression is an assignment operation, i.e., assigning the expression to a variable. If so, in step 84 the expression lattice for the target variable (i.e., the one being assigned) is updated. Specifically, the prior values of the expression lattice are merged with the expression lattice for the expression being assigned to the target variable.

[0048] If the results of step 82 are "false", the flow proceeds to step 86. In step 86, a test is made to determine if the expression is for an integral operation. If so, in step 88 the integral value lattices for each input of the operation are used to compute a resulting integral lattice and value for the expression.

[0049] If the results of step 86 are "false", the flow proceeds to step 90. In step 86, a test is made to determine if the expression is for a "size of" operation, i.e., of the form `size of(<variable or type>)`. If so, in step 92 an integral lattice is associated with the expression and its value will be the size of the variable (or type).

[0050] If the tests for steps 58, 66, 70, 74, 78, 82, 86, and 90 are false, then a default assignment is made in step 94 in which all values of the expression lattice are set to low.

[0051] The following examples are exemplary code segments to be analyzed by flow-insensitive analysis logic to determine whether a buffer flow vulnerability exists. Each is followed by a description of how the flow-insensitive analysis logic models the variables and expressions with the various lattices mentioned above.

[0052] Example 1:

```
void test1(int i) {
  char buf[100];
  char *p;
  switch (i) {
    case 1:
      p = "1";
      break;
    case 2:
      p = "12";
      break;
    default:
      p = "123";
      break;
  }
  strcpy(buf, p);
}
void test1(int i) {
```

[0053] An integral lattice for the variable `i` is created because its declared of "int" type and its integral lattice values are set to low: `i` $\leftarrow \perp$

```
char buf[100];
```

[0054] A vulnerability lattice is associated with the variable "`buf`" and because it's an array its memory size lattice is set to the size of the structure: `buf` $\leftarrow$ 100.

Since this variable is local and not visible to other routines or passed as an argument, all other lattices are set high: $\leftarrow \top$, see step 42 of figure 9.

```
char *p;
```

[0055] A vulnerability lattice is associated with the variable **p**. Because it is a pointer and it is not visible to other routines or passed as an argument all lattices are set high: $\leftarrow \top$, see step 48 of figure 9.

```
switch (i) {
```

[0056] The integral lattice for "**i**" has the value $\perp$, see above.

```
case 1:
  p = "1";
```

[0057] This is an assignment operation and thus will trigger the logic of steps 82 and 84 of figures 10A-B. Consequently, the expression lattice for the variable being assigned will be the merge results of the prior value of the lattice for the variable (in this case high $\top$) and the expression lattice for the expression being assigned to the variable, in this case the expression "1". The expression `"1"` has the lattice:

memory size lattice ← 2
data size lattice ← 2
null terminated lattice ← null terminated
memory location lattice ← constant memory
data origin lattice ← internal
string value lattice ← `"1"`

[0058] The results of the merger rules are used for the vulnerability lattice for **p** and are as follows:

memory size lattice ← 2
data size lattice ← 2
null terminated lattice ← null terminated
memory location lattice ← constant memory
data origin lattice ← internal
string value lattice ← `"1"`

```
  break;
case 2:
  p = "12";
```

[0059] This too is an assignment operation and thus will trigger the logic of steps 82 and 84 of figures 10A-B. Consequently, the expression lattice for the variable being assigned will be the merge results of the prior value of the lattice for the variable (see above) and the expression lattice for the expression being assigned to the variable, in this case the expression "12". The expression `"12"` has the lattice

memory size lattice ← 3
data size lattice ← 3
null terminated lattice ← null terminated
memory location lattice ← constant memory
data origin lattice ← internal
string value lattice ← "12"

[0060] The results of the merger rules are used for the vulnerability lattice for `p` and are as follows:

memory size lattice ← range of 2 to 3
data size lattice ← range of 2 to 3
null terminated lattice ← null terminated
memory location lattice ← constant memory
data origin lattice ← internal
string value lattice ← ⊥

```
  break;
default:
  p = "123";
```

[0061] This too is an assignment operation and thus will trigger the logic of steps 82 and 84 of figures 10A-B. Consequently, the expression lattice for the variable being assigned will be the merge results of the prior value of the lattice for the variable (see above) and the expression lattice for the expression being assigned to the variable, in this case the expression "123". The expression `"123"` has the lattice

memory size lattice ← 4
data size lattice ← 4
null terminated lattice ← null terminated
memory location lattice ← constant memory
data origin lattice ← internal
string value lattice ← "123"

[0062] The results of the merger rules are used for the vulnerability lattice for `p` and are as follows:

memory size lattice ← range of 2 to 4
data size lattice ← range of 2 to 4
null terminated lattice ← null terminated
memory location lattice ← constant memory
data origin lattice ← internal
string value lattice ← ⊥

```
  break;
}
strcpy(buf, p);
```

[0063] Since the address of buf is implicitly taken for the argument, the logic of step 62 is triggered and the vulnerability lattice for buf is modified to set the data size lattice, memory size lattice, string value lattice and data origin lattice to unknown.

[0064] Since the expression p refers to the value of a variable, the logic of step 68 is triggered and all values in the vulnerability lattice of the expression p are set to unknown.

[0065] Example 2:

```
static char y[100];
void test2(char *z) {
  strcpy(y, z);
}


static char y[100];
```

[0066] A vulnerability lattice is associated with array y. Its memory size is set to 100, its memory kind lattice is set to static, and all other lattices are set low: $\leftarrow \perp$. This is done because the variable y is visible to other routines, see step 40 of figure 9.

```
void test2(char *z) {
```

[0067] A vulnerability lattice is associated with pointer variable z. All lattices are set low: $\leftarrow \perp$. This is done because the variable z is passed to other routines as an argument, see step 49 of figure 9.

```
strcpy(y, z);
```

[0068] Since the address of y is implicitly taken for the argument, the logic of step 62 is triggered and the vulnerability lattice for y is modified to set the data size lattice, memory size lattice, string value lattice and data origin lattice to unknown.

[0069] Since the expression z refers to the value of a variable, the logic of step 68 is triggered and all values in the vulnerability lattice of the expression z are set to unknown.

[0070] After the flow insensitive analysis is performed, the call site analysis logic 16 is invoked. The call site analysis 16 derives vulnerability lattices for each variable or expression argument passed at a call site within the routine being analyzed. (A "call site" is the location within the code where a call is made to a routine.) The arguments may be variables or expressions. Under preferred embodiments, the call

site analysis is limited to calls to only select routines, procedures, or functions, as not all routines pose a vulnerability risk.

[0071] Under preferred embodiments, the call site analysis 16 requires that each call site be visited; however, this need not be in any specific order. Each argument of the call is analyzed such that any subexpression dependencies are processed first; that is, for example, before an expression making up an argument is processed, all the subexpressions given as input to that expression are processed.

[0072] The call site analysis logic is similar to the flow-insensitive analysis logic. However, unlike the flow-insensitive analysis logic, in the call site analysis logic any expression referring to the value of a variable associates the vulnerability lattice for that variable with the expression making such reference. In addition, any assignment operation to a variable does not change the vulnerability lattice for that variable.

[0073] Figures 11A-B show a flow chart of the steps performed in analyzing expressions in the call site analysis 16. The flow begins with an initial test 96 to determine if the expression being analyzed is for an address of a variable. If so, in step 98, a test is made to determine if that variable is to an array or structure or to determine if the variable is a constant string. If so, in step 102, a vulnerability lattice is associated with that expression and its memory size lattice is set to the size of the variable, and its memory location lattice is set to the kind of memory of the variable referenced. If the variable has a constant (const) attribute and it is a string, the data size lattice is set to the size of the string and the null terminated lattice is set to null terminated. The string value lattice is set to the value of the string. The data origin lattice is set to specify that the data origin is internal. If the expression is referring to the address of a variable but the variable is not a constant string, then in step 100 a vulnerability lattice is associated with that expression and its memory size lattice set to the size of the variable, and its memory location lattice is set to the kind of memory of the variable referenced. The other lattice entries are set to the low value.

[0074] If the results of step 96 are "false", the flow proceeds to step 104. In step 104, a test is made to determine if the expression is for a value of a variable. If so, in step 106, a vulnerability lattice is associated with the expression and all lattice entries are set to lattice values associated with the variable.

[0075] If the results of step 104 are "false", the flow proceeds to step 108. In step 108, a test is made to determine if the expression is for a constant string. If so, in step 110 a vulnerability lattice is associated with the expression and its memory size lattice

is set to the size of the constant string, including null termination byte; its data size lattice is set to the size of the constant string, including the null termination byte; its null termination lattice is set to indicate that it is null terminated; its memory location lattice is set to indicate constant memory; its string value lattice is set to the contents of the string; and its data origin lattice is set to internal.

[0076] If the results of step 108 are "false", the flow proceeds to step 112. In step 112, a test is made to determine if the expression is for an integral constant (i.e., an integer). If so, in step 114 an integral lattice is associated with the expression, and its value is set to the integer value.

[0077] If the results of step 112 are "false", the flow proceeds to step 116. In step 116, a test is made to determine if the expression is a "question mark/colon operation." If so, in step 118 a vulnerability lattice is associated with the expression and its lattice entries are set to the results from merging the vulnerability lattices of <expression$_2$> and <expression$_3$> (which have been set previously).

[0078] If the results of step 116 are "false", the flow proceeds to step 120. In step 120, a test is made to determine if the expression is an assignment operation, i.e., assigning the expression to a variable. If so, in step 122 the expression lattice for the target variable (i.e., the one being assigned) remains the same as the prior expression lattice for the variable.

[0079] If the results of step 120 are "false", the flow proceeds to step 124. In step 124, a test is made to determine if the expression is for an integral operation. If so, in step 126 the integral value lattices for each input of the operation are used to compute a resulting integral lattice and value for the expression.

[0080] If the results of step 124 are "false", the flow proceeds to step 128. In step 128, a test is made to determine if the expression is for a "size of" operation. If so, in step 130 an integral lattice is associated with the expression and its value will be the size of the variable (or type).

[0081] If the tests for steps 96, 104, 108, 112, 116, 120, 124, and 128 are false, then a default assignment is made in step 94 in which all values of the expression lattice are set to low.

[0082] Referring back to the exemplary code segments analyzed in connection with the flow-insensitive analysis logic, the following processing takes place.

[0083] Example 1:

```
void test1(int i) {
  char buf[100];
  char *p;
  switch (i) {
    case 1:
      p = "1";
      break;
    case 2:
      p = "12";
      break;
    default:
      p = "123";
      break;
  }
  strcpy(buf, p);
}
```

[0084] The call to **strcpy** has its arguments analyzed for lattice values.

Argument 1 has the value **buf**, which has the vulnerability lattice values as follows:

Memory Size Lattice ← 100,
Data Size Lattice ← ⊥
Null Terminated Lattice ← ⊥
String Value Lattice← ⊥
Memory Location Lattice ← Stack Memory
Data Origin Lattice ← ⊥

[0085] Argument 2 has the value **p**, which has the vulnerability lattice values as follows:

Memory Size Lattice ← range of 2 to 4
Data Size Lattice ← range of 2 to 4
Null Terminated Lattice ← Null Terminated
String Value Lattice ← ⊥
Memory Location Lattice ← Constant Memory
Data Origin Lattice ← Internal
}

[0086] Example 2:

```
static char y[100];
void test2(char *z) {
  strcpy(y, z);
}
```

[0087] This call to **strcpy** has its arguments analyzed for lattice values.

Argument 1 has the value **y**, which has vulnerability lattice values as follows:

Memory Size Lattice ← 100,
Data Size Lattice ← ⊥
Null Terminated Lattice ← ⊥
String Value Lattice← ⊥
Memory Location Lattice ← Static Memory
Data Origin Lattice ← ⊥

[0088] Argument 2 has the value **z**, which has vulnerability lattice values as follows:

Memory Size Lattice ← ⊥
Data Size Lattice ← ⊥
Null Terminated Lattice ← ⊥
String Value Lattice ← ⊥
Memory Location Lattice ← ⊥
Data Origin Lattice ← ⊥
}

[0089] According to one embodiment of the invention, the vulnerability lattices are created for those arguments to library call sites that are known to have potential vulnerabilities. The library call sites may be identified in a database 20.

Language Independent Analysis of Vulnerability Lattices at Select Call Sites

[0090] Once the vulnerability lattices are created for the arguments to select routine calls, the source code is further analyzed in a language independent manner to determine if the source code has vulnerabilities that should be reported. Preferred embodiments of the invention perform such analysis with vulnerability assessment logic 18 operating in conjunction with a vulnerability database 20.

[0091] The vulnerability database 20 is a database containing information about a number of pre-identified routines. Among other things, it specifies the conditions that can cause a vulnerability. The conditions are specified as constraints to a vulnerability lattice for arguments passed to the routine.

[0092] The vulnerability assessment logic 18 operates as follows. Each *call site* in the source code, as analyzed by call site analysis 16, is examined, though this need not be in any specific order. The name of the called routine, and possibly information about its argument types, is used to create a routine lookup name.. This routine lookup name is used as a key in the vulnerability database 20 to discover if this call site is potentially vulnerable.

[0093] If the lookup fails to discover a corresponding entry, then the call site is determined to be not vulnerable, because the routine name has no known vulnerabilities specified in the database 20.

[0094] If the lookup discovers a corresponding entry, then the entry is examined for a list of matching actions, which are rules used to assess a specific call. Those matching actions are supplied in a specific order. Each matching action is compared to the vulnerability lattice for each argument to determine if the vulnerability lattice from the argument matches the requirement of the matching action. As illustrated in the example described below, if a match occurs, then the action reports a vulnerability for the examined call site. The report may then be used by a developer to address the potential vulnerability. Multiple vulnerabilities may be detected for a specific call site.

[0095] Referring back to the exemplary code segments analyzed in connection with the language specific processing logic, the following processing takes place in vulnerability assessment.

[0096] Example 1:

This is the example which had the following language-specific code:

```
strcpy(buf, p);
```

[0097] The call site analysis for this call yielded the following vulnerability lattice for the first argument `buf`:

Memory Size Lattice ← 100,
Data Size Lattice ← $\perp$
Null Terminated Lattice ← $\perp$
String Value Lattice← $\perp$
Memory Location Lattice ← Stack Memory
Data Origin Lattice ← $\perp$

[0098] The call site analysis also yielded the following vulnerability analysis for the second argument `p`:

Memory Size Lattice ← range of 2 to 4
Data Size Lattice ← range of 2 to 4
Null Terminated Lattice ← Null Terminated
String Value Lattice ← $\perp$
Memory Location Lattice ← Constant Memory
Data Origin Lattice ← Internal

[0099] The matching actions returned from the database 20 specify certain rules to be applied in assessing the vulnerability lattices for the call to routine strcpy(). In the particular case of the call to **strcpy** the rules check that the first argument has a minimum memory size that is larger than or the same size as the maximum data size for the second argument. In this way, the rules (matching actions) determine whether this specific call creates a risk of buffer overflow. In this case, no overflow is possible given the effectively semantic analysis of the source code involved.

[0100] The minimum memory size for argument 1 (100) is greater than or equal to the maximum data size for argument 2 (4), so the buffer cannot overflow. The data origin for argument 2 is internal, so it cannot be a vulnerability. The call is not marked as a vulnerability.

[0101] Example 2:

This is the example which had the following language-specific code:

```
strcpy(y, z);
```

[0102] The call site analysis for this call yielded the following vulnerability lattice for the first argument y:

Memory Size Lattice ← 100,
Data Size Lattice ← ⊥
Null Terminated Lattice ← ⊥
String Value Lattice← ⊥
Memory Location Lattice ← Static Memory
Data Origin Lattice ← ⊥

[0103] The call site analysis also yielded the following vulnerability analysis for the second argument z:

Memory Size Lattice ← ⊥
Data Size Lattice ← ⊥
Null Terminated Lattice ← ⊥
String Value Lattice ← ⊥
Memory Location Lattice ← ⊥
Data Origin Lattice ← ⊥

[0104] The matching actions returned from the database 20 specify certain rules to be applied in assessing the vulnerability lattices for the call to routine strcpy(). In the particular case of the call to **strcpy** the rules check that the maximum data size for

the second argument is $\perp$, and thus unknown. Thus, there is a possibility that the buffer can overflow. Likewise, the data origin of the second argument is $\perp$, and thus unknown. Thus, there is a second possibility of a vulnerability. (If the input is unknown or external, there is the possibility of the size being too small or the input being not of internal origin which would produce a vulnerability.) In the particular case of the call to strcpy: the maximum data size for argument 2 is $\perp$, so the buffer can overflow. The data origin for argument 2 is $\perp$, so it can be a vulnerability. The call is marked as a vulnerability.

[0105] The embodiments described above are directed to a method of detecting buffer overflow vulnerabilities. As noted above, the method may be used to detect other

[0106] vulnerabilities, such as race condition and privilege escalation.

Race Condition

[0107] As used herein, the term "race condition" means a pair of routine calls that happen sequentially in a program and which, if not performed atomically (i.e. without interruption by another thread or process on the machine), could become a vulnerability. A typical example is a call to determine the access rights of a file, and a subsequent call to write or read of that file based on the access. If the process is interrupted between the two calls and the file attributes are modified during the interruption, the second call may be reading the wrong information or writing to an inappropriate file.

[0108] The following is an exemplary code segment to be analyzed to determine whether a race condition exists. It uses access() and fopen() to illustrate a related pair of calls that could be vulnerable.

[0109] Example 3:

```
... some code A ...
1) r = access( filename, ... )
... some code B ...
2) if( r ) then
   ... some code C ...
3) fopen( filename, ... )
   ... some code D ...
```

[0110] In this example, there is a call to access() for a particular filename, followed by a test of the return value from access(). If the test succeeds, fopen() is called for the same filename. Placeholders are listed for arbitrary code that could

happen around the numbered statements. The fopen() call is reachable from the access() call; which means that there are no other operations on the file between the two calls, and the fopen() call will follow the access() call if the test succeeds.

[0111] While this example shows the argument to access() and fopen() as a single variable name, it is possible that the argument could be any arbitrary expression such as filename_list[i] (an entry in an array of names), or fullpath + baselen (a pointer to a character string baselen characters into fullpath). The important point is that the runtime value of that argument is the same for both calls.

[0112] As in the embodiments described above for buffer overflow conditions, a lexical analyzer is used to generate an IR for the code to be analyzed for a race condition. In this embodiment, the IR includes information about declarations in the program, and records information about identifiers in the program such as their type. It can distinguish function declarations from function calls.

[0113] A control flow graph is provided to show the basic block structure and the branches between the blocks which determine program control flow. An example of a control flow graph is shown in Fig. 12. The rectangular entities 140 are basic blocks (contiguous, straight line statements with no branching, representing "if", "while", etc.); ovals 142 are regions of code with arbitrary control flow inside; and arrows 144 represent control flow between basic blocks or code regions.

[0114] Using the control flow graph, the system traverses backward from the block containing the open() call through the blocks preceding it. In the example shown, it goes to the block containing the call to access () and notes that the access() call precedes the open () call. Knowing that the calls are related, it examines the argument list of each call, focusing on the arguments corresponding to the filename. As a heuristic, it compares the structure of the corresponding expressions. In this example, it would find that both expressions are references to the same variable, and it would conclude that the two calls are referencing the same file and as a result, a race condition vulnerability would be flagged.

[0115] In another embodiment of a system for detecting race condition vulnerability, data flow analysis could be used with the system described above in order to provide information about the value of variables at different points in the program. For example, it could determine whether the variable filename had the same value in both the access() call and the fopen() call). Data flow analysis could also be

used to determine whether an argument to access() described as fullpath + baselen, had the same value as an argument to fopen() described as filename.

Privilege Escalation

[0116] Privilege escalation vulnerabilities can arise when an application with a high level of system privileges can be made to perform actions outside of the intended design, allowing an outside party to gain privileged access to the system that they would not otherwise posses.

[0117] The following is an exemplary code segment for detection of privilege escalation.

[0118] Example 4:

```
void somefunc(){
... SetSecurityDescriptorDacl( &descriptor, TRUE,
        NULL /* ACL */, FALSE );
}
```

[0119] In this example, a Windows API call sets security attributes for a resource. The vulnerability is that a resource's ACL (access control list) should never be set to null because the resource would then be accessible or modifiable by an unauthorized user.

[0120] As in the embodiments described above, a language parser is used to create an IR from the source code. The IR provides a symbol table which includes information for all types, constants, variables and functions declared in the file. The information for function 'somefunc' includes a reference to the statements of 'somefunc.' Statements of the IR include the control flow statements of the applicable language ("if," "while," "for," etc. in C or C++) and expressions (including assignment, function calls, arithmetic operations, etc.). Function call information includes a symbol table entry for the routine being called, and a list of expressions corresponding to the arguments. A database of possible vulnerable calls is provided.

[0121] The IR is traversed, with each function definition and statement within that definition being visited. The routine being called at function call node is matched against the database information. When there is a match, the function call is looked at in greater detail.

[0122] The particular conditions which make a particular call vulnerable are previously determined. In the example shown above, it is known that that the potential issue is that the third argument to SetSecurityDescriptonDacl() should not be

NULL. The IR for this call would point to SetSecurityDescriptorDacl as the routine being called, and there would be four entries in the list of expressions for arguments. The first would be the address of the variable 'descriptor', and the last three would be the IR constants for TRUE, NULL, and FALSE.

[0123] Finding a match with SetSecurityDescriptorDacl would trigger a deeper examination of the arguments to the call. In this case, knowledge about SetSecurityDescriptorDacl's potential vulnerability would cause an examination of the third argument. The IR directly describes this as NULL, and this call site would be flagged as vulnerable.

[0124] As described above, preferred embodiments of the invention analyze certain semantic characteristics of the source code itself to determine whether a vulnerability potentially exists. For example, arguments to a routine may be algorithmically analyzed in view of some known behavior about the routine (e.g., that a routine copies one argument to a buffer pointed to by another argument) to detect problematic calls. This approach avoids the many false positives found in known prior art approaches and proposals.

[0125] To date, security experts analyzed code using known rules to look for vulnerabilities but this was labor intensive and error prone. The invention automates the semantic analysis for vulnerabilities such as buffer overflow, race condition and privilege escalation. It also provides a framework so that as other vulnerabilities get discovered the matching actions for the detection of such may be specified and incorporated into the preferred system.

[0126] In the embodiment described above, the source code is (a) all or part of the text for an executable program written in the ANSI C language as described in the ANSI Standard X3J11, and with commonly used extensions, such as those provided by the Microsoft and GNU compiler; or (b) all or part of the text for an executable program written in the ANSI C++ language as described in the ANSI Standard X3J16, and with commonly used extensions, such as those provided by the Microsoft and GNU compilers. It is understood, however, that the invention may be used to analyze source code written in other languages as well.

[0127] While the invention has been described in connection with certain preferred embodiments, it will be understood that it is not intended to limit the invention to those particular embodiments. On the contrary, it is intended to cover all alternatives, modifications and equivalents as may be included in the appended claims.

Some specific figures and source code languages are mentioned, but it is to be understood that such figures and languages are, however, given as examples only and are not intended to limit the scope of this invention in any manner.

[0128] The reference in this specification to any prior publication (or information derived from it), or to any matter which is known, is not, and should not be taken as an acknowledgment or admission or any form of suggestion that that prior publication (or information derived from it) or known matter forms part of the common general knowledge in the field of endeavour to which this specification relates.

[0129] Throughout this specification and the claims which follow, unless the context requires otherwise, the word "comprise", and variations such as "comprises" and "comprising", will be understood to imply the inclusion of a stated integer or step or group of integers or steps but not the exclusion of any other integer or step or group of integers or steps.

THE CLAIMS DEFINING THE INVENTION ARE AS FOLLOWS:

1. A computer implemented method of detecting vulnerabilities in a pre-existing source code listing, said source code listing having a listed sequence of expressions, each expression including a set of operands and operators to transform values of the operands, said listed sequence of expressions having an inherent control flow indicative of the run-time execution of the expressions and an inherent data flow indicative of the run-time transformations of operand values, said source code listing further having routine calls, said routine calls including arguments with which to invoke a routine, said source code listing being stored in a computer-readable medium, said computer implemented method comprising the acts of:

executing computer instructions to analyse the source code listing to create computer models of the operands, said models each including a corresponding initial set of information to represent the range of values for said operand, and said models being transformed, in response to analysis of the source code, to have a transformed range of values to correspond to the operand transformations expressed in the source code listing, said models being stored in computer memory, and wherein each model specifies pre-determined characteristics about and possible values for each operand as a result of said source code expressions;

executing computer instructions to use said operand models to create models of said arguments to routine calls, said argument models being stored in computer memory;

executing computer instructions to use said argument models in conjunction with pre-specified criteria for the corresponding routine calls to determine whether the routine calls possess vulnerabilities as a consequence of the arguments and known routine behaviour; and

generating a report that identifies the vulnerabilities said report being viewable by a developer-user, so the developer-user may address the vulnerabilities identified in the report by modifying the source code listing if necessary.

2. The computer implemented method of claim 1, using a database having computer-readable information about a predefined set of source code routine calls, said information

2004232058 28 Apr 2010

specifying one or more conditions that present a vulnerability during execution of the source code routine call, wherein the act of using the argument models in conjunction with pre-specified criteria for the corresponding routine calls to determine whether the routine calls possess vulnerabilities as a consequence of the arguments and known routine behaviour comprises the act of using the data base to retrieve information for a corresponding routine call to check for the condition to see whether the routine call presents vulnerability.

3. The computer implemented method of claim 1 wherein the report identifies the location in the source code listing where the vulnerability occurred.

4. A computer implemented method of detecting vulnerabilities in a pre-existing source code listing, said source code listing having a listed sequence of expressions, each expression including a set of operands and operators to transform values of the operands, said listed sequence of expressions having an inherent control flow indicative of the run-time execution of the expressions and an inherent data flow indicative of the run-time transformations of operand values, said source code listing further having routine calls, said routine calls including arguments with which to invoke a routine said source code listing being stored in a computer-readable medium, said computer implemented method comprising the acts of:

executing computer instructions to analyse the source code listing to create computer models of arguments to routine calls in the source code listing, said argument models being stored in computer memory, said argument models each including a corresponding initial set of information to represent the range of values for said argument, and said argument models being transformed, in response to analysis of the source code, to have a transformed range of values to correspond to the transformations expressed in the source code listing, said models being stored in computer memory, and wherein each argument model specifies pre-determined characteristics about and possible values for each argument as a result of said source code expressions;

executing computer instructions to use said argument models in conjunction with pre-specified criteria for the corresponding routine calls to determine whether the routine

calls possess vulnerabilities as a consequence of the arguments and the routine behaviour; and

generating a report that identifies the vulnerabilities said report being viewable by a developer-user, so the developer-user may address the vulnerabilities identified in the report by modifying the source code listing if necessary.

5. The computer implemented method of claim 4, using a database having computer-readable information about a predefined set of source code routine calls, said information specifying one or more conditions that present a vulnerability during execution of the source code routine call, wherein the act of using the argument models in conjunction with pre-specified criteria for the corresponding routine calls to determine whether the routine calls possess vulnerabilities as a consequence of the arguments and the routine behaviour comprises the act of using the data base to retrieve information for a corresponding routine call to check for the condition to see whether the routine call presents a vulnerability.

6. The computer implemented method of claim 4 wherein the report identifies the location in the source code listing where the vulnerability occurred.

7. A computer implemented utility for detecting vulnerabilities in a pre-existing source code listing, said source code listing having a listed sequence of expressions, each expression including a set of operands and operators to transform values of the operands, said listed sequence of expressions having an inherent control flow indicative of the run-time execution of the expressions and an inherent data flow indicative of the run-time transformations of operand values, said source code listing further having routine calls, said routine calls including arguments with which to invoke a routine, said source code listing being stored in a computer-readable medium, said utility comprising a computer-readable medium encoded with:

executable instructions for analysing the source code listing to create computer models of the operands, said models each including a corresponding initial set of information to represent the range of values for said operand, and said models being transformable, in response to analysis of the source code, to have a transformed range of

values to correspond to the operand transformations expressed in the source code listing, said models, storable in a computer memory, and wherein each model specifies pre-determined characteristics about and possible values for each operand as a result of said source code expressions;

executable instructions for using the operand models to create models of arguments to routine calls in the source code listing, said argument models being stored in computer memory; and

executable instructions for using the argument models in conjunction with pre-specified criteria for the corresponding routine calls to determine whether the routine calls possess vulnerabilities as a consequence of the arguments and known routine behaviour; and

executable instructions for generating a report that identifies the vulnerabilities said report being viewable by a developer-user, so the developer-user may address the vulnerabilities identified in the report by modifying the source code listing if necessary.

8. The computer implemented utility of claim 7, using a data base having computer readable information about a predefined set of source code routine calls, said information specifying one or more conditions that present a vulnerability during execution of the source code routine call, wherein the executable instructions for using the argument models in conjunction with pre-specified criteria for the corresponding routine calls to determine whether the routine calls possess vulnerabilities as a consequence of the arguments and known routine behaviour includes executable instructions for using the database to retrieve information for a corresponding routine call to check for the specified condition to see whether the routine call presents a vulnerability.

9. The computer implemented utility of claim 7 wherein the report identifies the location in the source code listing where the vulnerability occurred.

10. A computer implemented method of detecting vulnerabilities in a pre-existing source code listing, said source code listing having a listed sequence of expressions, each expression including a set of operands and operators to transform values of the operands,

said listed sequence of expressions having an inherent control flow indicative of the run-time execution of the expressions and an inherent data flow indicative of the run-time transformations of operand values, said source code listing further having routine calls, said routine calls including arguments with which to invoke a routine, said arguments including expression-references and operand-references to computer files, said source code listing being stored in a computer-readable medium, said computer implemented method comprising the acts of:

executing computer instructions to analyse the source code listing to create computer models of said control flow to indicate the run-time sequence in which routine calls will be invoked and to create computer models of said arguments for the routine calls using a flow insensitive analysis, wherein said control flow models include a control flow graph, and wherein each of said models of arguments is stored in computer memory and specifies pre-determined characteristics about and a range of possible values for the corresponding argument as a result of said source code expressions;

executing computer instructions to use said computer models of said control flow in order to determine a run-time sequence of execution of a pair of routine calls by traversing the control flow graph backwards, said pair of routine calls having a first routine call and second routine call in which execution of the first routine call precedes execution of said second routine call;

executing computer instructions to determine whether a second routine to be executed has a second argument with a corresponding modelled range of possible of values that includes a reference to a file that is also within a corresponding modelled range of possible values for a first argument of the first routine to be executed, so that a possibility of the first and second arguments referring to the same file is determined even when said expression-references and operand-references to computer files for said first and said second arguments are lexically dissimilar;

executing computer instructions to identify said sequence as a race condition vulnerability; and

generating a report that is viewable by a user and that identifies the race condition vulnerabilities, so the user may modify the source code listing to address the vulnerability if desired.

11. The method of claim 10 further including the act of executing computer instructions to analyse the source code listing to create computer models of said data flow to indicate the run-time transformations of operand values and including the act of using data flow models to resolve the expression-references and operand-references to computer files in the first and second routine calls to detect whether both routines refer to the same computer file.

12. A system for detecting vulnerabilities in a pre-existing source code listing, said source code listing having a listed sequence of expressions, each expression including a set of operands and operators to transform values of the operands, said listed sequence of expressions having an inherent control flow indicative of the run-time execution of the expressions and an inherent data flow indicative of the run-time transformations of operand values, said source code listing further having routine calls, said routine calls including arguments with which to invoke a routine, said arguments including expression-references and operand-references to computer files, said source code listing being stored in a computer-readable medium, said system comprising:

computer-executable instructions on a computer-readable medium to analyse the source code listing to create computer models of said control flow to indicate the run-time sequence in which routine calls will be invoked and to create computer models of said arguments for the routine calls using a flow insensitive analysis, wherein said control flow models include a control flow graph, and wherein each of said models of arguments is stored in computer memory and specifies pre-determined characteristics about and a range of possible values for the corresponding argument as a result of said source code expressions;

computer-executable instructions on a computer-readable medium to use said computer models of said control flow to determine a run-time sequence of execution of a pair of routine calls by traversing the control flow graph backwards, said pair of routine calls having a first routine call and second routine call in which execution of the first routine call precedes execution of said second routine call;

computer-executable instructions on a computer-readable medium to determine whether a second routine to be executed has a second argument with a corresponding

modelled range of possible of values that includes a reference to a file that is also within a corresponding modelled range of possible values for a first argument of the first routine to be executed, so that a possibility of the first and second arguments referring to the same file is determined even when said expression-references and operand-references to computer files for said first and said second arguments are lexically dissimilar;

computer-executable instructions on a computer-readable medium to identify said sequence as a race condition vulnerability; and

computer-executable instructions on a computer-readable medium to generate a report that is viewable by a user and that identifies the race condition vulnerabilities, so the user may modify the source code listing to address the vulnerability if desired.

13. The system of claim 12 further including computer-executable instructions on a computer-readable medium to analyse the source code listing to create computer models of said data flow to indicate the run-time transformations of operand values and to use the data flow models to resolve the expression-references and operand-references to computer files in the first and second routine calls to detect whether both routines refer to the same computer file.

14. A computer implemented method of detecting vulnerabilities in a pre-existing source code listing, said source code listing having a listed sequence of expressions, each expression including a set of operands and operators to transform values of the operands, said listed sequence of expressions having an inherent control flow indicative of the run-time execution of the expressions and an inherent data flow indicative of the run-time transformations of operand values, said source code listing being expressed in multiple programming languages, said source code listing further having routine calls including arguments with which to invoke a routine, said source code listing being stored in a computer-readable medium, said computer implemented method comprising the acts of:

executing computer instructions to create a single intermediate representation of said source code listing regardless of programming language by parsing said source code listing;

executing computer instructions to provide a database having computer-readable records associated with pre-identified routines, each record specifying an argument vulnerability condition for arguments of said corresponding pre-identified routine, that, if satisfied, presents a vulnerability during execution of said routine;

executing computer instructions to statically analyse said intermediate representation of said source code listing to create computer models of the operands, said models representing expected transformation of the operands from run time execution of a computer program created by compilation of said source code listing, said models being stored in computer memory;

executing computer instructions said operand models to create models of said arguments to routine calls, said argument models being stored in computer memory;

executing computer instructions to retrieve, from said database, a record corresponding to routine calls represented in said intermediate representation;

executing computer instructions to compare said argument models with said condition specified in the retrieved record to determine whether the routine call possesses vulnerabilities as a consequence of the arguments; and

generating a report that identifies detected vulnerabilities, said report being viewable by a developer-user, so the developer-user may address the vulnerabilities identified in the report by modifying the source code listing if necessary.

15. The computer implemented method of claim 14 wherein the report identifies the location in the source code listing where the vulnerability occurred.

16. A method of detecting privilege escalation vulnerabilities in a pre-existing source code listing, said source code listing having a listed sequence of expressions, each expression including a set of operands and operators to transform values of the operands, said source code listing further having routine calls, said routine calls including arguments with which to invoke a routine, said source code listing being stored in computer readable medium having computer executable instructions, wherein a privilege escalation vulnerability is an uncontrolled escalation of system privileges that allows unauthorized access to system resources, the method comprising:

providing a list specifying routines that potentially cause privilege escalation vulnerabilities;

providing pre-specified ranges of values for arguments of routines in the list that cause privilege escalation vulnerabilities;

analysing the source code listing to identify calls to routines specified in the list;

analysing the source code listing to semantically analyse arguments of the identified routine calls to determine routine calls that possess privilege escalation vulnerabilities using the pre-specified ranges of values;

wherein semantically analysing the arguments of the identified routine calls comprises analysing the source code listing to create computer models of the arguments, each model specifying a range of values that each corresponding argument can take when the source code listing is executed; and

generating a report that identifies the vulnerabilities.

17. The method of claim 16, wherein analysing the source code listing to create computer models of the arguments comprises:

analysing the source code listing to create computer models of said operands, each of said operand models specifying a range of values of each corresponding operand as a result of operand transformations expressed in the source code listing; and

using the operand models to create the argument models.

18. A computer implemented method of detecting vulnerabilities in a pre-existing source code listing substantially as hereinbefore described with reference to the accompanying drawings.

19. A system for detecting vulnerabilities in a pre-existing source code listing substantially as hereinbefore described with reference to the accompanying drawings.

20. A method of detecting privilege escalation vulnerabilities in a pre-existing source code listing substantially as hereinbefore described with reference to the accompanying drawings.

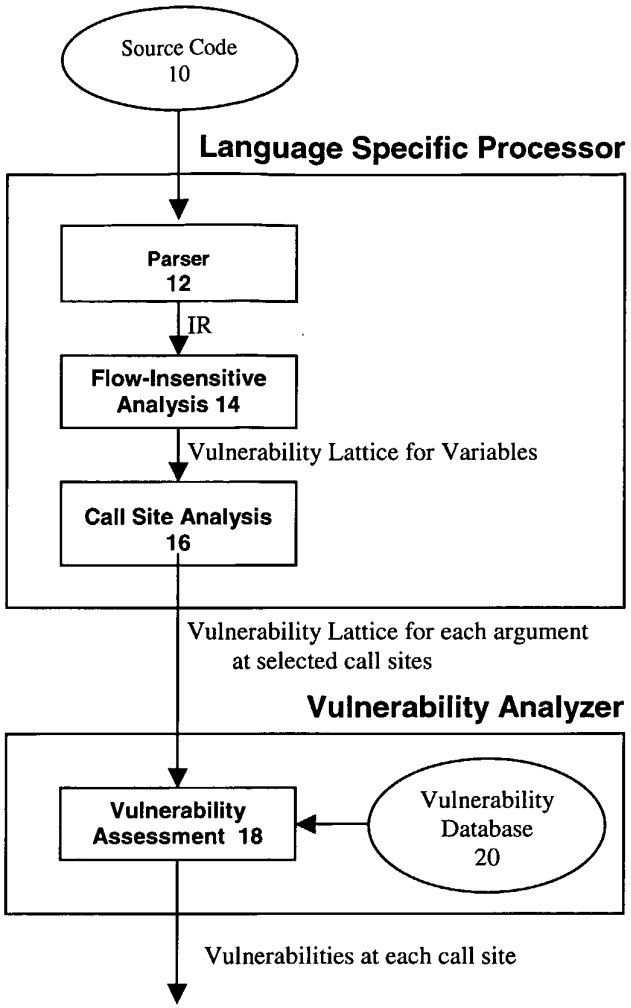

Source Code
10
Language Specific Processor
Parser
12
IR
Flow-Insensitive
Analysis 14
Vulnerability Lattice for Variables
Call Site Analysis
16
Vulnerability Lattice for each argument
at selected call sites
Vulnerability Analyzer
Vulnerability
Assessment 18
Vulnerability
Database
20
Vulnerabilities at each call site

FIGURE 1

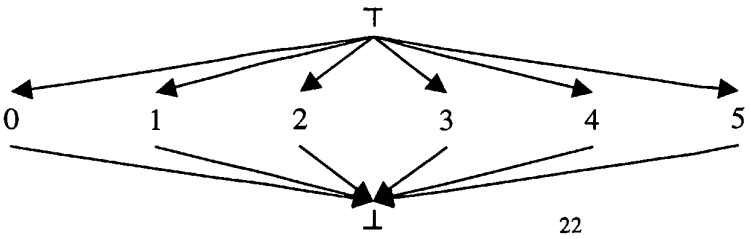


FIGURE 2

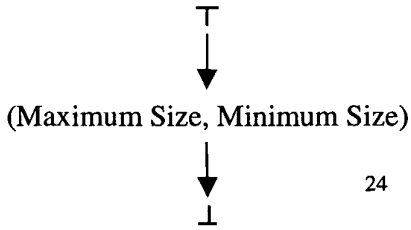


FIGURE 3

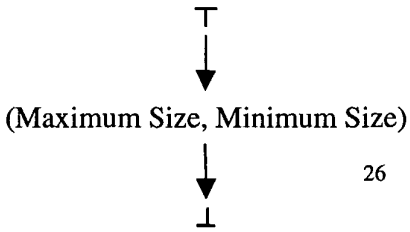


FIGURE 4

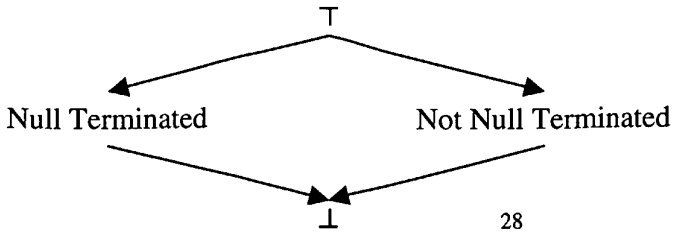


FIGURE 5

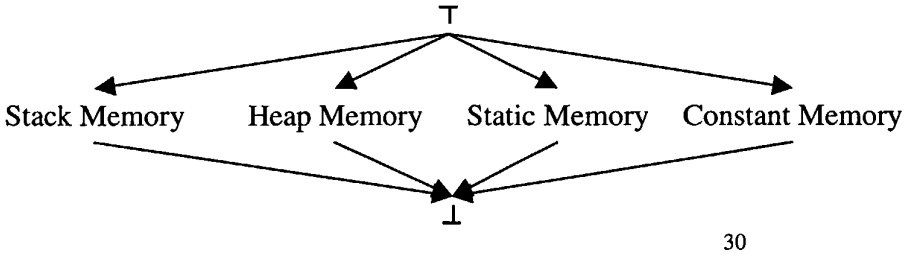


FIGURE 6

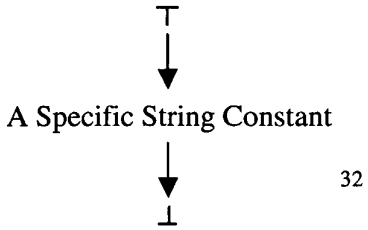


FIGURE 7

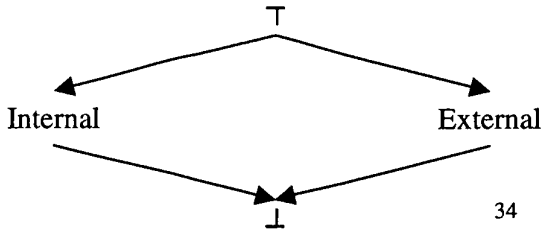


FIGURE 8

FIGURE 9

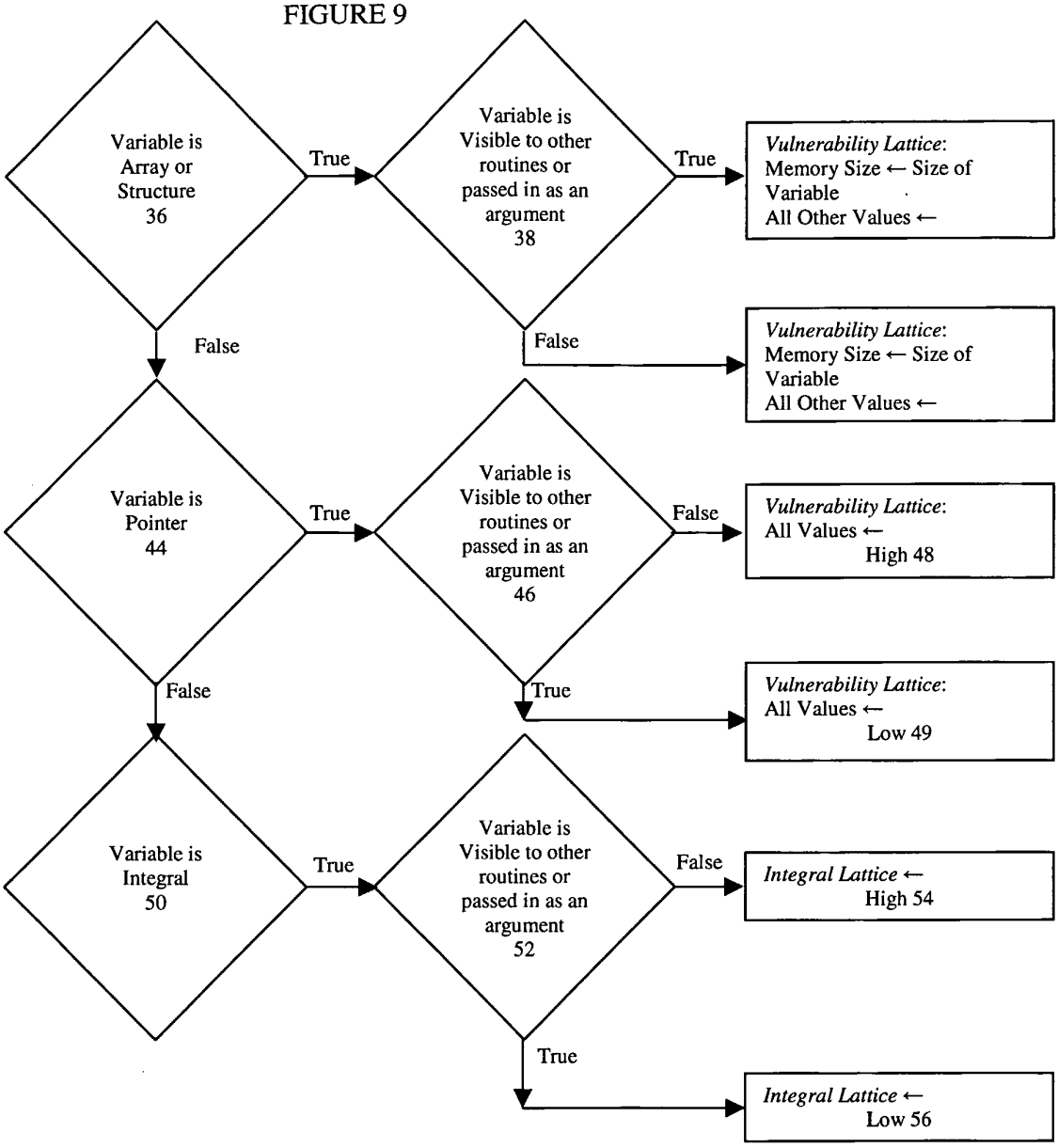

Variable is Array or Structure 36
True
Variable is Visible to other routines or passed in as an argument 38
True
Vulnerability Lattice: Memory Size ← Size of Variable All Other Values ←
False
False
Vulnerability Lattice: Memory Size ← Size of Variable All Other Values ←
Variable is Pointer 44
True
Variable is Visible to other routines or passed in as an argument 46
False
Vulnerability Lattice: All Values ← High 48
False
True
Vulnerability Lattice: All Values ← Low 49
Variable is Integral 50
True
Variable is Visible to other routines or passed in as an argument 52
False
Integral Lattice ← High 54
True
Integral Lattice ← Low 56

FIGURE 10A

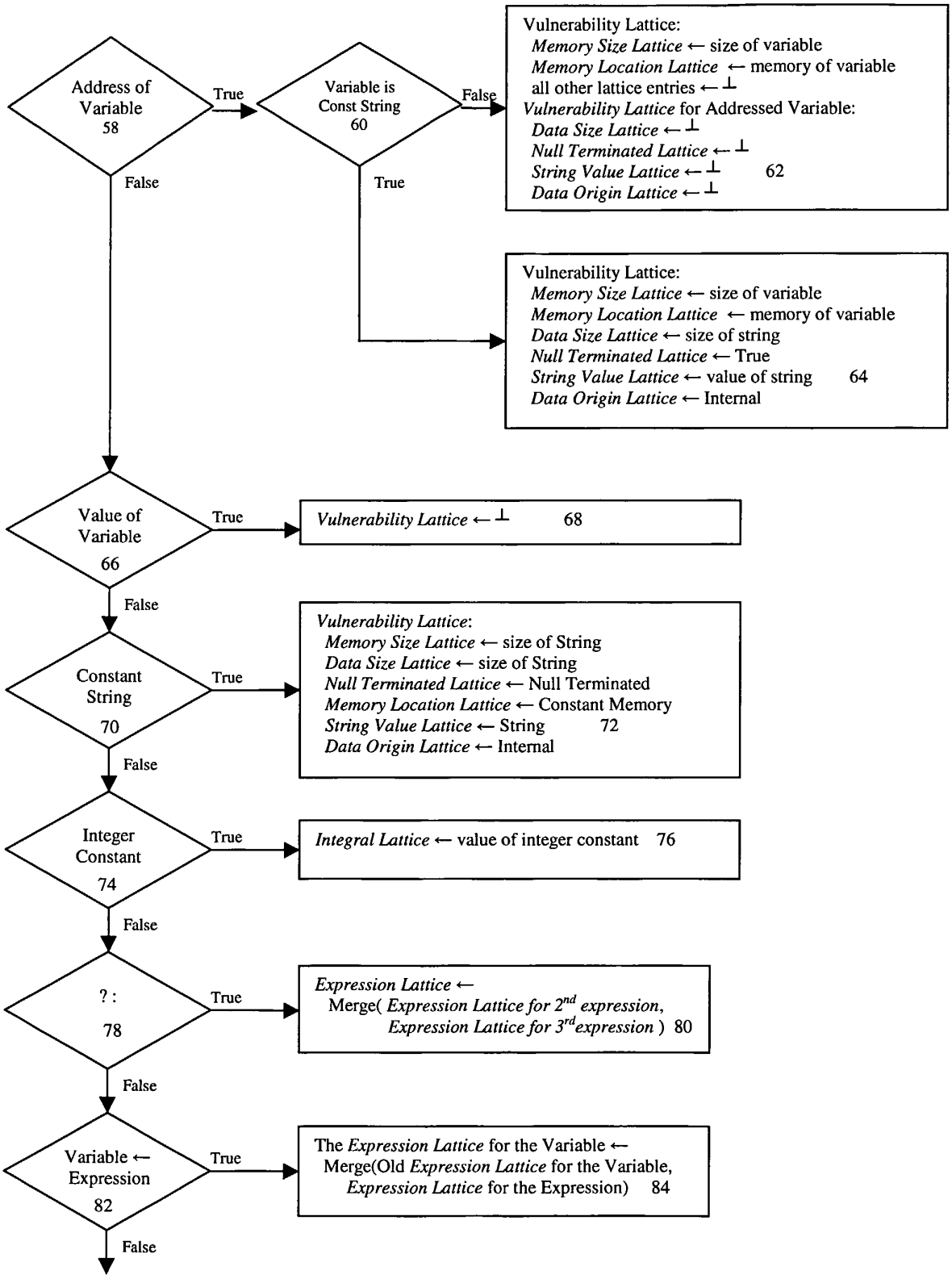

Address of Variable 58
True
Variable is Const String 60
False
True
False
Vulnerability Lattice:
Memory Size Lattice ← size of variable
Memory Location Lattice ← memory of variable
all other lattice entries ← ⊥
Vulnerability Lattice for Addressed Variable:
Data Size Lattice ← ⊥
Null Terminated Lattice ← ⊥
String Value Lattice ← ⊥ 62
Data Origin Lattice ← ⊥
Vulnerability Lattice:
Memory Size Lattice ← size of variable
Memory Location Lattice ← memory of variable
Data Size Lattice ← size of string
Null Terminated Lattice ← True
String Value Lattice ← value of string 64
Data Origin Lattice ← Internal
Value of Variable 66
True
Vulnerability Lattice ← ⊥ 68
False
Constant String 70
True
Vulnerability Lattice:
Memory Size Lattice ← size of String
Data Size Lattice ← size of String
Null Terminated Lattice ← Null Terminated
Memory Location Lattice ← Constant Memory
String Value Lattice ← String 72
Data Origin Lattice ← Internal
False
Integer Constant 74
True
Integral Lattice ← value of integer constant 76
False
? : 78
True
Expression Lattice ←
Merge(Expression Lattice for 2nd expression,
Expression Lattice for 3rd expression) 80
False
Variable ← Expression 82
True
The Expression Lattice for the Variable ←
Merge(Old Expression Lattice for the Variable,
Expression Lattice for the Expression) 84
False

Figure 10B

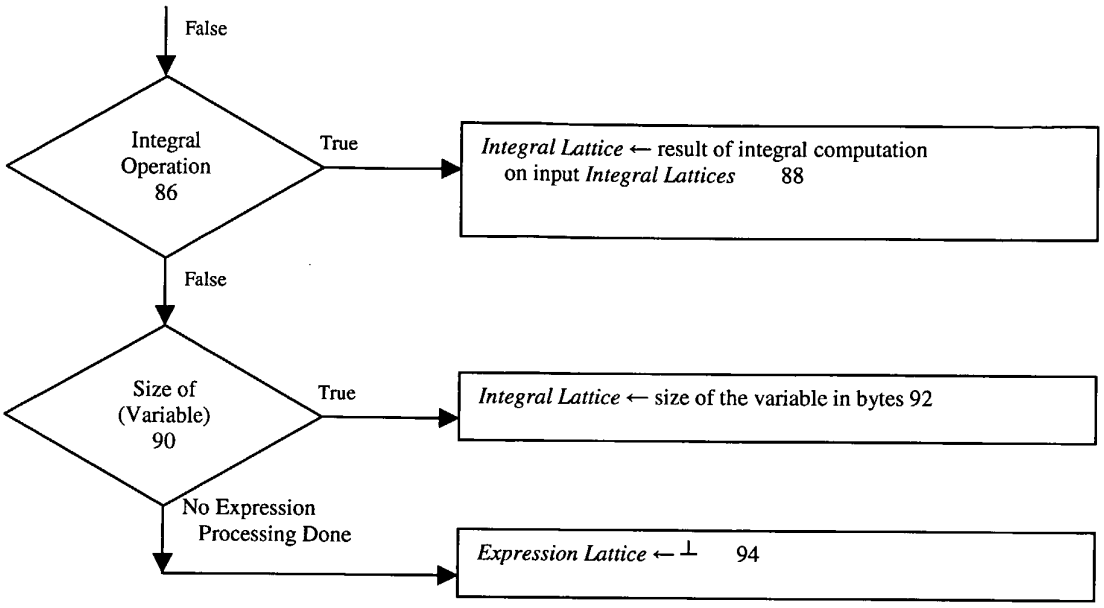

False
Integral Operation 86
True
Integral Lattice ← result of integral computation on input Integral Lattices 88
False
Size of (Variable) 90
True
Integral Lattice ← size of the variable in bytes 92
No Expression Processing Done
Expression Lattice ← ⊥ 94

FIGURE 11A

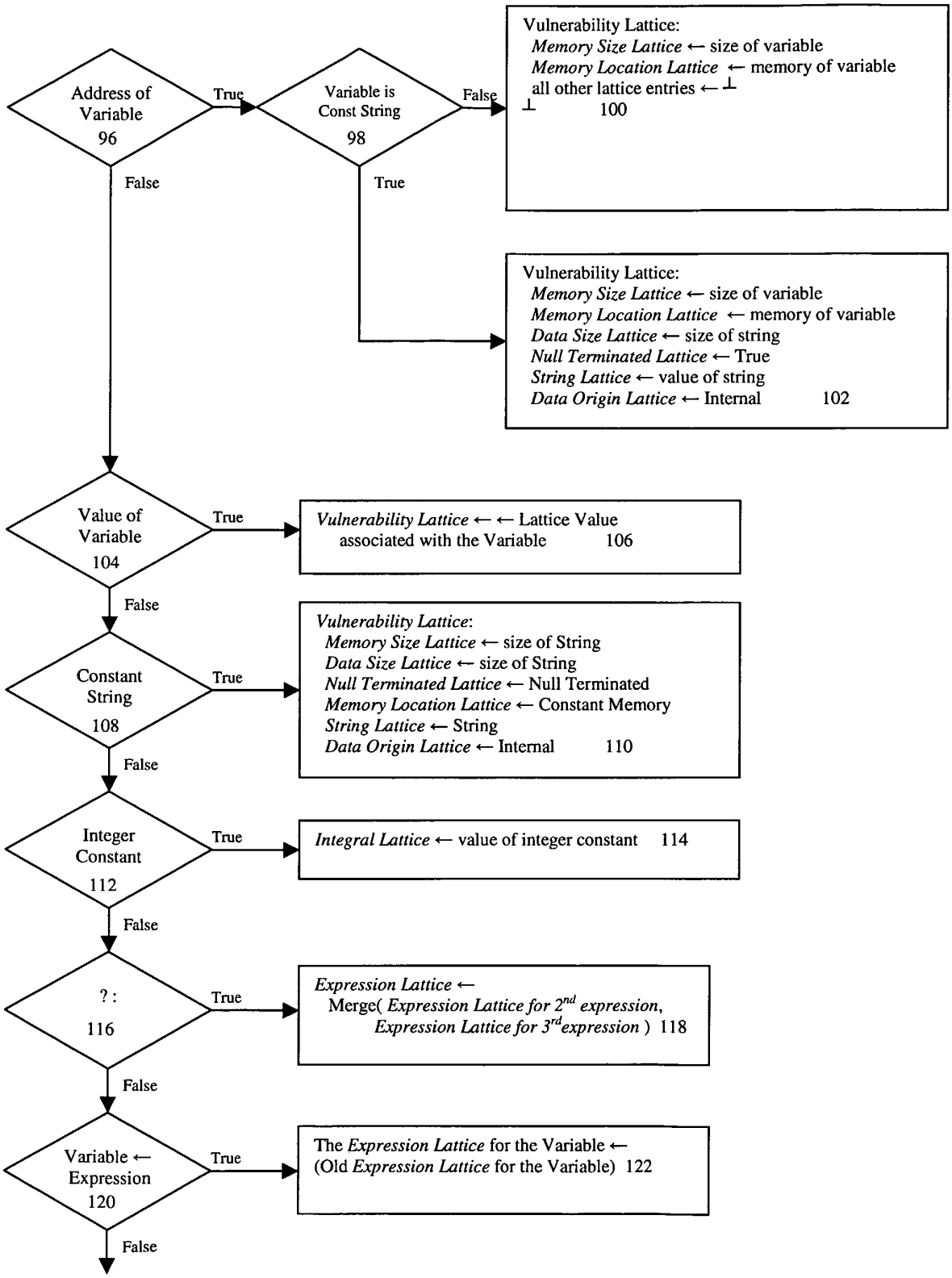
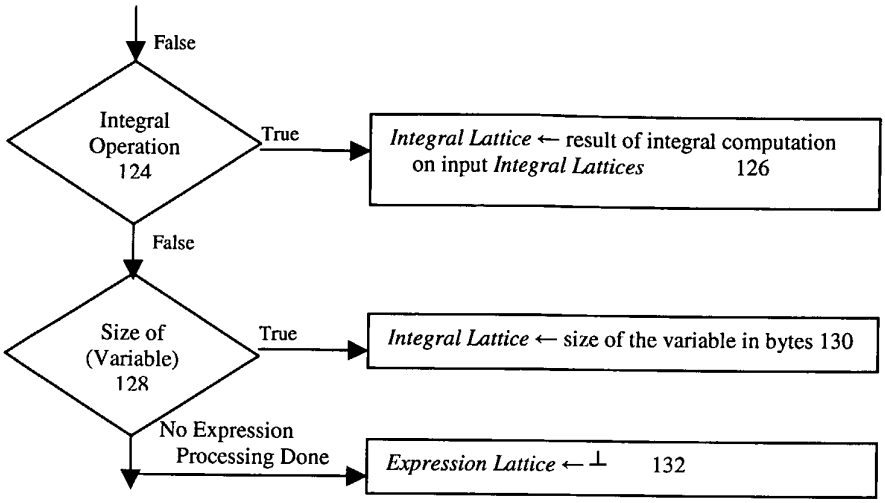

Address of Variable 96
True
Variable is Const String 98
False
Vulnerability Lattice:
Memory Size Lattice ← size of variable
Memory Location Lattice ← memory of variable
all other lattice entries ← ⊥
⊥ 100
False
True
Vulnerability Lattice:
Memory Size Lattice ← size of variable
Memory Location Lattice ← memory of variable
Data Size Lattice ← size of string
Null Terminated Lattice ← True
String Lattice ← value of string
Data Origin Lattice ← Internal 102
Value of Variable 104
True
Vulnerability Lattice ← ← Lattice Value associated with the Variable 106
False
Constant String 108
True
Vulnerability Lattice:
Memory Size Lattice ← size of String
Data Size Lattice ← size of String
Null Terminated Lattice ← Null Terminated
Memory Location Lattice ← Constant Memory
String Lattice ← String
Data Origin Lattice ← Internal 110
False
Integer Constant 112
True
Integral Lattice ← value of integer constant 114
False
? : 116
True
Expression Lattice ←
Merge(Expression Lattice for 2nd expression,
Expression Lattice for 3rd expression) 118
False
Variable ← Expression 120
True
The Expression Lattice for the Variable ←
(Old Expression Lattice for the Variable) 122
False

Figure 11B


False
Integral Operation 124
True
Integral Lattice ← result of integral computation on input Integral Lattices 126
False
Size of (Variable) 128
True
Integral Lattice ← size of the variable in bytes 130
No Expression Processing Done
Expression Lattice ← ⊥ 132

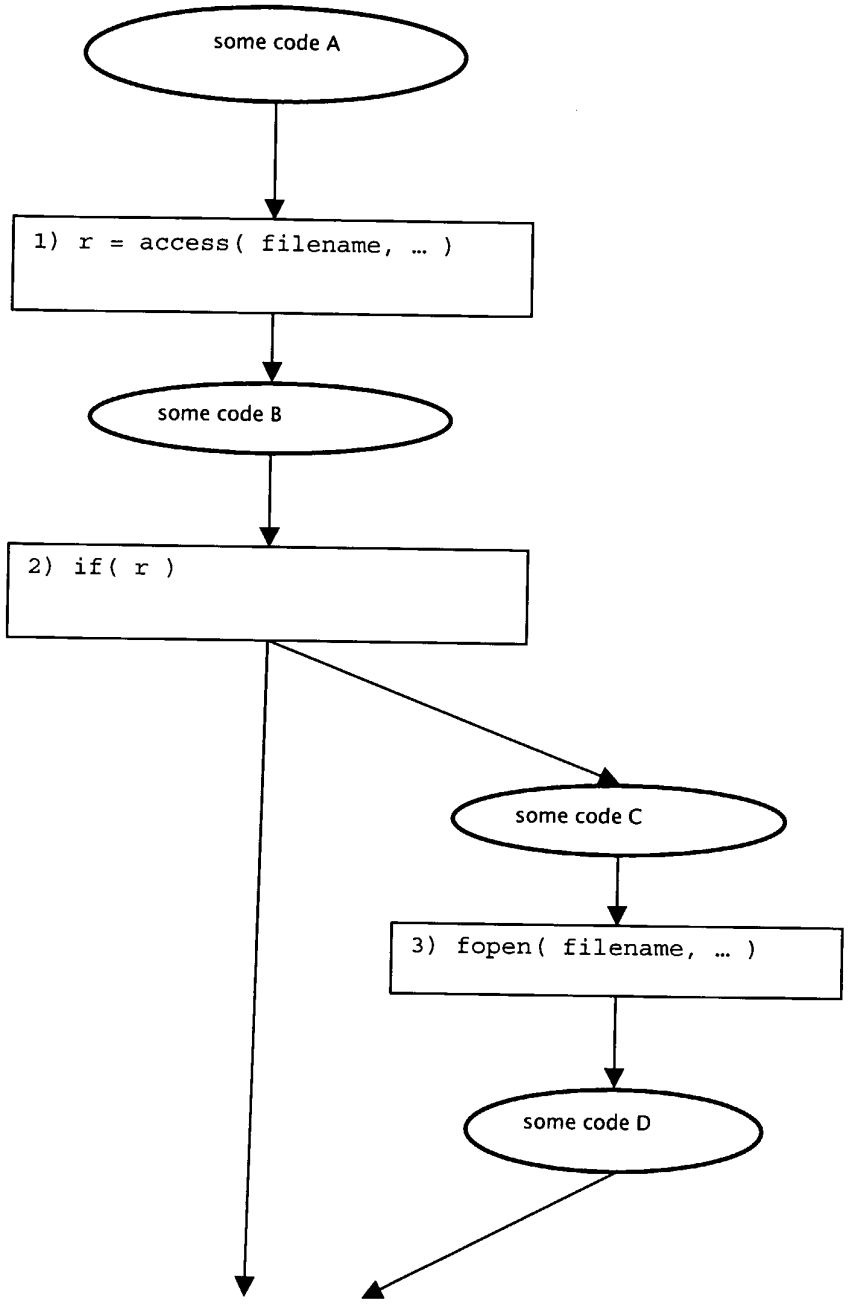

some code A
1) r = access(filename, …)
some code B
2) if(r)
some code C
3) fopen(filename, …)
some code D

FIGURE 12

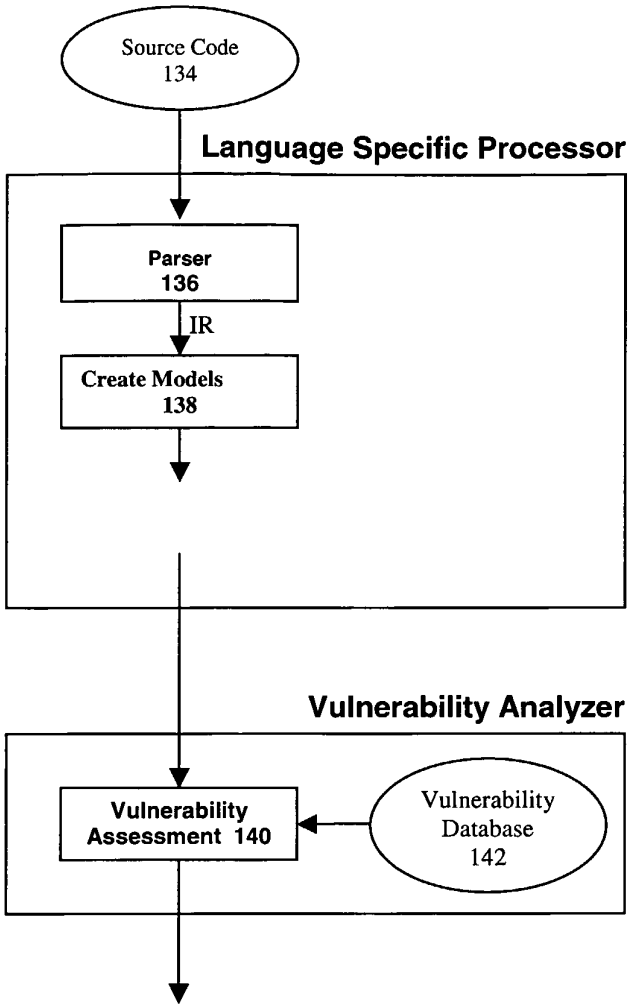

Source Code
134
Language Specific Processor
Parser
136
IR
Create Models
138
Vulnerability Analyzer
Vulnerability
Assessment 140
Vulnerability
Database
142

FIGURE 13