VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector

Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin, *Fellow, IEEE*

Abstract—Automatically detecting software vulnerabilities is an important problem that has attracted much attention. However, existing vulnerability detectors still cannot achieve the vulnerability *detection capability* and *locating precision* that would warrant their adoption for real-world use. In this paper, we present Vulnerability Deep learning-based Locator (VulDeeLocator), a deep learning-based fine-grained vulnerability detector, for C programs with source code. VulDeeLocator advances the state-of-the-art by simultaneously achieving a high detection capability and a high locating precision. When applied to three real-world software products, VulDeeLocator detects four vulnerabilities that are not reported in the National Vulnerability Database (NVD); among these four vulnerabilities, three are not known to exist in these products until now, but the other one has been "silently" patched by the vendor when releasing newer versions of the vulnerable product. The core innovations underlying VulDeeLocator are (i) the leverage of intermediate code to accommodate semantic information that cannot be conveyed by source code-based representations, and (ii) the concept of *granularity refinement* for precisely pinning down locations of vulnerabilities.

Index Terms—Vulnerability detection, security, deep learning, program analysis, program representation.

1 Introduction

Software vulnerabilities are a major cause of cyber attacks. Unfortunately, vulnerabilities are prevalent as evidenced by the steady increase of vulnerabilities reported by the Common Vulnerabilities and Exposures (CVE) [1]. One important approach towards eliminating vulnerabilities is to design *vulnerability detectors* to detect (and patch) them. An ideal vulnerability detector should simultaneously achieve a high *detection capability* and a high *locating precision* (i.e., precisely pinning down the vulnerable lines of code).

A popular family of vulnerability detectors is based on *static analysis*. These detectors can be divided into *code similarity-based* ones and *pattern-based* ones. Code similarity-based detectors [2], [3], [4], [5] can detect vulnerabilities caused by code cloning, and can achieve a high locating precision when they indeed detect vulnerabilities. However, they incur high false-negatives (i.e., low detection capability) when applied to detect vulnerabilities that are not caused by code cloning. Pattern-based detectors can be further divided into *rule-based* ones and *machine learning-based* ones. Rule-based detectors [6], [7], [8], [9], [10], [11] can identify the vulnerable lines of code when they indeed *correctly* detect vulnerabilities, but often incur a *low* detection capability (because of their high false-positives and high false-negatives). Moreover, they require human analysts to define vulnerability detection rules. Machine learning-based detectors use vulnerability patterns for detection, where the patterns are learned from analyst-defined feature representation of vulnerable programs [12], [13], [14], [15], [16]. However, these detectors cannot achieve a high locating precision because they operate at a coarse granularity, typically at the function level [12].

The recent development in *machine learning-based* vulnerability detection is to use deep learning [17], [18] and operate at a finer, or "program slice", level. These detectors also can relieve the problem of *manual-feature definition*, which has received further attention recently [19], [20], [21]. However, these detectors still offer *inadequate detection capability* and *inadequate locating precision*.

To see their *inadequate detection capability*, we observe that the state-of-the-art detector [18], despite its improvement upon [17], reportedly achieves an F1-measure of 86.0%, a false-positive rate of 10.1%, and a false-negative rate of 12.2% (see Table 4 in Section 6). This inadequacy can be attributed to the detector's incapability in (i) capturing the relations between semantically-related statements and (ii) taking advantage of control flows and the variable *define-use* relations. To see (i), we observe that programs often contain many user-defined and/or system header files (e.g., `.h`) for defining *types* and *macros*. Analyzing source code alone, as is done in [17], [18], cannot associate the *uses* of types and macros in program files (e.g., `.c`) to their

Corresponding author: Deqing Zou.

- *Z. Li is with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Security Engineering Research Center, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China, and also with School of Cyber Security and Computer, Hebei University, Baoding, 071002, China. E-mail: lizhenhbu@gmail.com*
- *D. Zou is with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Security Engineering Research Center, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China, and also with Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen 518057, China. E-mail: deqingzou@hust.edu.cn*
- *S. Xu is with the Department of Computer Science, University of Texas at San Antonio, San Antonio, TX 78249, USA. E-mail: shxu@cs.utsa.edu.*
- *Z. Chen, Y. Zhu, and H. Jin are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Security Engineering Research Center, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {zhaoxaunchen, yokisir, hjin}@hust.edu.cn*

definitions in header files. This is not surprising because source code analysis tools may not even be able to correctly identify the relations between the definition and the use of macros and global variables in a same program file, as what will be shown in Section 6.6. To see (ii), we observe that source code is not in the Static Single Assignment (SSA) form, which assures that each variable is defined-and-then-used and is assigned exactly once [22]. That is, source code-based representations do not fully expose control flows and/or variable *define-use* relations [23], and some semantic information cannot be captured by vulnerability detectors that leverage source code-based representations [17], [18].

To see their *inadequate locating precision*, we observe that although they operate on program slices (which are finer-grained than functions), a program slice can have many lines of code. For example, according to the dataset published in [18], 78.7% of their program slices have at least 10 lines of code and 47.8% of them have at least 20 lines, indicating a low locating precision. That is, coarse-grained vulnerability detection is merely a pre-step for vulnerability assessment because it cannot precisely pinpoint vulnerabilities [24].

The preceding inadequacies of state-of-the-art vulnerability detectors motivates the need of detectors that can achieve a high detection capability and a high locating precision simultaneously.

Our contributions. In this paper we make two contributions. First, we propose Vulnerability Deep learning-based Locator (VulDeeLocator), a deep learning-based fine-grained vulnerability detector, for C programs with source code. VulDeeLocator can *simultaneously* achieve a high detection capability and a high locating precision. When compared with the state-of-the-art detector [18], VulDeeLocator offers (i) an 11.0%, 9.6%, and 8.2% improvement in the vulnerability detection F1-measure, false-positive rate, and false-negative rate, respectively; and (ii) a 3.9X improvement in the vulnerability locating precision. When applied to three real-world software products, it detects four vulnerabilities that are *not* reported in the National Vulnerability Database (NVD) [25]. Among these four vulnerabilities, three are not known to exist in these products until now and have been notified to the vendors, and the other one has been "silently" patched by the vendor when releasing newer versions of the vulnerable product. VulDeeLocator has two innovations:

- It leverages program *intermediate code* to define program slices for vulnerability detection. Such slices can accommodate semantic information that cannot be conveyed by source code-based representations, such as the aforementioned (i) relations between the definitions of types and macros and their uses and (ii) control flows and variable define-use relations.
- It leverages the idea of *granularity refinement* introduced in this paper to make the granularity of detector outputs (e.g., 3 lines of code) finer than the granularity of detector inputs (e.g., 32 lines of code).

To the best of our knowledge, we are the first to use intermediate code to design machine learning-based vulnerability detectors, despite that intermediate code has been leveraged in rule-based vulnerability detectors [8], [9]. Moreover, we introduce new guiding principles for vulnerability candidate representations and new requirements for fine-grained vulnerability detectors. These guiding principles and requirements would guide the design of tailored fine-grained vulnerability detectors. We demonstrate the feasibility of granularity refinement via a novel variant of the Bidirectional Recurrent Neural Network (BRNN), dubbed "BRNN for vulnerability detection and locating" (BRNN-vdl).

Second, we prepare a dataset in the Lower Level Virtual Machine (LLVM) intermediate code with accompanying program source code. This dataset is motivated by the need of evaluating the effectiveness of VulDeeLocator; it contains 119,782 vulnerability candidates in intermediate code, among which 30,201 are vulnerable and 89,581 are not vulnerable. It is not trivial to prepare this dataset because we need user-defined and system header files for generating intermediate code. In order for other researchers to use the dataset, we have made it available at https://github.com/VulDeeLocator/VulDeeLocator. We will publish the source code used in our experiments on the same website.

Paper organization. Section 2 discusses the basic ideas and definitions underlying VulDeeLocator. Section 3 presents an overview of VulDeeLocator. Section 4 describes how VulDeeLocator leverages intermediate code and Section 5 describes how VulDeeLocator pinpoints vulnerabilities. Section 6 presents our experiments and results. Section 7 discusses limitations of the present study. Section 8 reviews the related prior work. Section 9 concludes the present paper.

2 Basic Ideas and Definitions

2.1 Basic Ideas

The basic idea underlying VulDeeLocator is to extract some *tokens* (e.g., identifiers, operators, constants, and keywords) from program source code according to a given set of vulnerability syntax characteristics, and then leverage the intermediate code of the same program to accommodate the statements in the intermediate code that are semantically related to those tokens. These statements are encoded into vectors (which are then used to train a neural network) or are the input to the trained neural network for vulnerability detection. The output in the testing phase is finer-grained (i.e., shorter or smaller) than the corresponding input. Figure 1 illustrates these basic ideas, showing that in the testing phase an input of d' intermediate code statements leads to a *refined* output of two source code statements indicating where the vulnerability is.

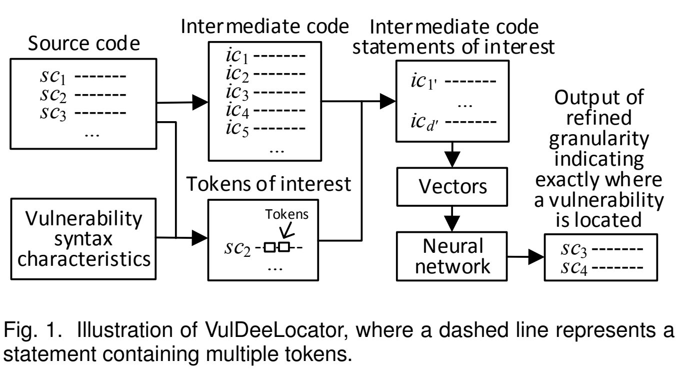

Fig. 1. Illustration of VulDeeLocator, where a dashed line represents a statement containing multiple tokens.

The idea of leveraging intermediate code deserves more explanation. Figure 2 describes a buffer overflow vulnerability candidate related to pointer "*data*" (Line 2). However, the candidate cannot accommodate the control flow of conditional operator "*N<m?N:99*" (Line 14), despite that this kind of semantic information is important for vulnerability detection (because control flow can expose vulnerabilities [26]). Source code-based representations cannot convey this kind of semantic information because source code is not in the SSA form [22], which assures that each variable is assigned exactly once. These limitations of source code-based representations can be overcome by intermediate code.

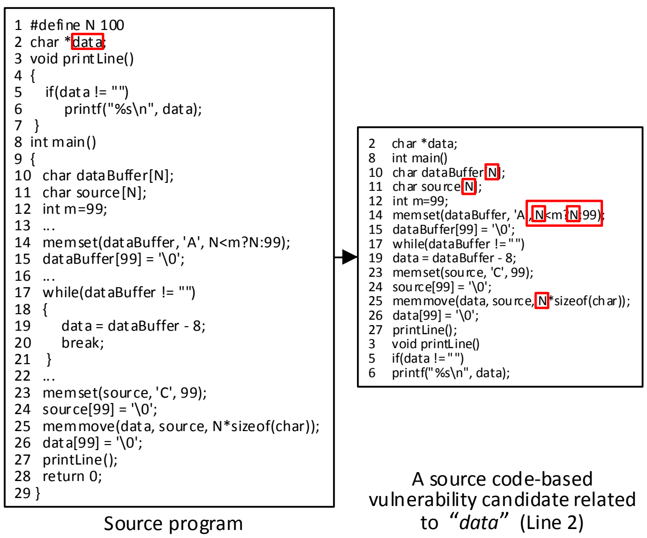

Fig. 2. An example showing that source code-based representation cannot fully expose the semantic information corresponding to "*data*" (Line 2 in the source program) that may be related to a vulnerability.

Figure 2 also shows another example, where the macro definition identifier "*N*" (highlighted with boxes) cannot be identified as "*100*" when using source code-based representation to derive the vulnerability candidate related to "*data*". This is because source code parsing may not associate the use of macros and global variables to their definitions in the same program file, let alone associating the use of types and macros in a program file (e.g., `.c`) to their definitions in another file (i.e., header file). As we will see later, this kind of semantic information can be captured by intermediate code.

2.2 Formal Definitions

In order to make the aforementioned ideas precise, we need formal definitions. We start with reviewing:

Definition 1 (source program). A source program P is a set of program files $p_1, \ldots, p_n$, denoted by $P = \{p_1, \ldots, p_n\}$. A program file p_i ($1 \le i \le n$) consists of a set of functions and possibly some *outside* type and/or macro definitions (i.e., these type and/or macro definitions are not specified inside, but are specified outside, those functions). We denote these functions and outside type and macro definitions by $f_{i,1}, \ldots, f_{i,m_i}$ and therefore denote a program file by $p_i = \{f_{i,1}, \ldots, f_{i,m_i}\}$. A function or outside type and/or macro definition $f_{i,j}$ ($1 \le j \le m_i$) is a sequence of statements $s_{i,j,1}, \ldots, s_{i,j,r_{i,j}}$, denoted by $f_{i,j} = (s_{i,j,1}, \ldots, s_{i,j,r_{i,j}})$. A statement $s_{i,j,k}$ ($1 \le k \le r_{i,j}$) is a sequence of tokens, denoted by $s_{i,j,k} = (t_{i,j,k,1}, \ldots, t_{i,j,k,\xi_{i,j,k}})$, where $t_{i,j,k,l}$ ($1 \le l \le \xi_{i,j,k}$) is a token (e.g., identifier, operator, constant, and keyword).

It is intuitive that vulnerabilities exhibit some syntax characteristics that can be leveraged to identify some pieces of code (i.e., program slices) as *initial candidates* for vulnerability detection [17], [18]. Vulnerability syntax characteristics can be represented via some appropriate data structures (e.g., Abstract Syntax Trees or ASTs), which allow one to extract pieces of source code that match these characteristics. Regardless of the representation of such characteristics, we can use $H = \{h_1, ..., h_\eta\}$ to represent a set of η vulnerability syntax characteristics. We can use a characteristic h_q ($1 \le q \le \eta$) to extract pieces of source code as syntax-based vulnerability candidates, which are the starting point for vulnerability detection, leading to:

Definition 2. (source code- and Syntax-based Vulnerability Candidate or sSyVC [18]) Consider a source program $P = \{p_1, \ldots, p_n\}$ where a program file $p_i = \{f_{i,1}, \ldots, f_{i,m_i}\}$, a function or outside type and/or macro definition $f_{i,j} = \{s_{i,j,1}, \ldots, s_{i,j,r_{i,j}}\}$, and a statement $s_{i,j,k} = (t_{i,j,k,1}, \ldots, t_{i,j,k,\xi_{i,j,k,z}})$. Given a set of vulnerability syntax characteristics $H = \{h_1, \ldots, h_\eta\}$, a sSyVC $y_{i,j,k,z}$ is one or multiple consecutive tokens in statement $s_{i,j,k}$ that match some vulnerability syntax characteristic h_q ($1 \le q \le \eta$), denoted by $y_{i,j,k,z} = (t_{i,j,k,u}, \ldots, t_{i,j,k,v})$ where $1 \le u \le v \le \xi_{i,j,k,z}$.

Since we propose leveraging program intermediate code to capture semantic information, we need to define:

Definition 3 (intermediate code). Given a source program $P = \{p_1, \ldots, p_n\}$ where program file $p_i = \{f_{i,1}, \ldots, f_{i,m_i}\}$, the *intermediate code* of source program P, denoted by P', is a set of intermediate program files denoted by $P' = \{p'_1, \ldots, p'_n\}$, where p'_i is the intermediate code of program file p_i and correspondingly consists of a set of functions and possibly some *outside* type and/or macro definitions (i.e., these type and/or macro definitions are not specified within, but are specified outside, those functions in intermediate-code program file p'_i), denoted by $f'_{i,1}, \ldots, f'_{i,m_i}$. We denote by $p'_i = \left\{f'_{i,1}, \ldots, f'_{i,m_i}\right\}$, where $f'_{i,j}$ ($1 \le j \le m_i$) is the intermediate code of a function or outside type and/or macro definition $f_{i,j}$, denoted by a sequence of statements $f'_{i,j} = (s'_{i,j,1}, \ldots, s'_{i,j,r'_{i,j}})$.

Given sSyVCs extracted from a program and the intermediate code of the same program, we can now define:

Definition 4. (intermediate code- and Semantics-based Vulnerability Candidate or iSeVC) Consider a source program $P = \{p_1, \ldots, p_n\}$, its intermediate code $P' = \{p'_1, \ldots, p'_n\}$, and a sSyVC $y_{i,j,k,z}$ of P. Denote by $y'_{i,j,k,z}$ the intermediate code of sSyVC $y_{i,j,k,z}$ in p'_i. The iSeVC corresponding to sSyVC $y_{i,j,k,z}$ is a sequence of statements $s'_{a_1,b_1,c_1}, \ldots, s'_{a_{\rho_{i,j,k,z}},b_{\psi_{i,j,k,z}},c_{\varpi_{i,j,k,z}}}$ in intermediate code P' of source program P; these statements are data or control dependent [27] on $y'_{i,j,k,z}$, denoted by $e_{i,j,k,z} = (s'_{a_1,b_1,c_1}, \ldots, s'_{a_{\rho_{i,j,k,z}},b_{\psi_{i,j,k,z}},c_{\varpi_{i,j,k,z}}})$. That is, the iSeVC corresponding to sSyVC $y_{i,j,k,z}$ is a program slice of $y'_{i,j,k,z}$ in the intermediate code of program P.

3 Overview of VulDeeLocator

Figure 3 highlights the structure of VulDeeLocator, which can be instantiated with specific intermediate code representations and deep learning models. The input to VulDeeLocator is the source code of training programs for learning a neural network or target programs for vulnerability detection. More specifically, the learning-phase input includes source code of C programs, which may or may not be vulnerable. The source code of C programs should satisfy the following: (i) they can be compiled into (platform-independent) intermediate code, such as the LLVM intermediate code [28], [22]; and (ii) the vulnerable programs are accompanied by descriptions on the locations of their vulnerabilities, which will be leveraged to locate vulnerabilities in target programs.

At a high level, VulDeeLocator has two components. The first component leverages intermediate code representation of training programs and target programs as follows:

- Step I: Extracting sSyVCs from the source code, namely pieces of code that bear some vulnerability syntax characteristic(s).
- Step II: Generating iSeVCs from the intermediate code according to sSyVCs.

The second component uses the intermediate code-based representation to detect and locate vulnerabilities as follows:

- Step III: Labeling iSeVCs extracted from training programs as vulnerable or not and vulnerability locations.
- Step IV: Training a neural network model from the vector representations of the iSeVCs and their labels.
- Step V: Using the trained neural network model to detect and locate vulnerabilities in target programs.

Note that the *learning* phase corresponds to Steps I-IV and the *testing* (i.e., detection) phase corresponds to Steps I, II, and V.

4 Intermediate Code-based Vulnerability Candidate Representation

4.1 Guiding Principles for Vulnerability Candidate Representation

It is intuitive that vulnerability detectors should accommodate program semantic information, highlighting the importance of identifying effective vulnerability candidate representations. For this purpose, we propose using the following principles to guide the identification of effective vulnerability candidate representation.

- **Principle 1: Accommodating semantically-related program statements across files**. Some files may be dependent on others because, for example, a variable used or referred in one file may be defined in another file. Effective vulnerability candidate representations should accommodate this define-use relation.
- **Principle 2: Accommodating semantically-related program statements across functions**. Semantically-related statements may go beyond boundaries of functions, meaning that effective vulnerability candidate representations should accommodate, and further preserve the order of, those semantically-related statements, even if they belong to different functions.

4.2 Extracting sSyVCs

As defined above, a sSyVC is a piece of code that is extracted from a program according to some vulnerability syntax characteristic(s). There may be many approaches to obtaining vulnerability syntax characteristics for vulnerability detection. As a concrete example, we leverage the syntax characteristics of known vulnerabilities and represent these characteristics via Abstract Syntax Trees (ASTs) of the program source code (more precisely, attributes of the nodes on ASTs). This will ease the extraction of sSyVCs according to vulnerability syntax characteristics. We define the following four kinds of vulnerability syntax characteristics, which are mentioned here because they will be referred in our examples.

- Library/API Function Call (FC): This vulnerability syntax characteristic is that the type of a node on the AST is function call, the function name matches a library/API function name, and at least one argument of the function call is a variable.
- Array Definition (AD): This vulnerability syntax characteristic is that the type of a node on the AST is variable declaration and the code corresponding to the node contains characters '[' and ']'.
- Pointer Definition (PD): This vulnerability syntax characteristic is that the type of a node on the AST is variable declaration and the code corresponding to the node contains character '$*$'.
- Arithmetic Expression (AE): This vulnerability syntax characteristic is that the type of a node on the AST is assignment expression and the node has at least one variable at the right-hand side of the assignment expression.

Given the source code of a program, one can generate its AST(s), from which sSyVCs can be extracted by identifying the nodes whose type and code match some vulnerability syntax characteristics. We reiterate that these syntax characteristics themselves are far from adequate in detecting vulnerabilities because they cannot accommodate the due semantic information that is related to vulnerabilities.

Figure 4(a) is an example showing the sSyVCs (highlighted by boxes) in a program: sSyVCs related to the FC-kind vulnerability syntax characteristics include "*printf*" (Line 6), "*memset*" (Lines 14 and 23), and "*memmove*" (Line 25); sSyVCs related to the AD-kind vulnerability syntax characteristics include "*dataBuffer*" (Line 10) and "*source*" (Line 11); sSyVCs related to the PD-kind vulnerability syntax characteristics include "*data*" (Line 2); and sSyVCs related to the AE-kind vulnerability syntax characteristics include "*data=dataBuffer-8*" (Line 19).

4.3 Generating iSeVCs

In order to generate iSeVCs, we propose focusing on the LLVM intermediate code, namely the LLVM Intermediate

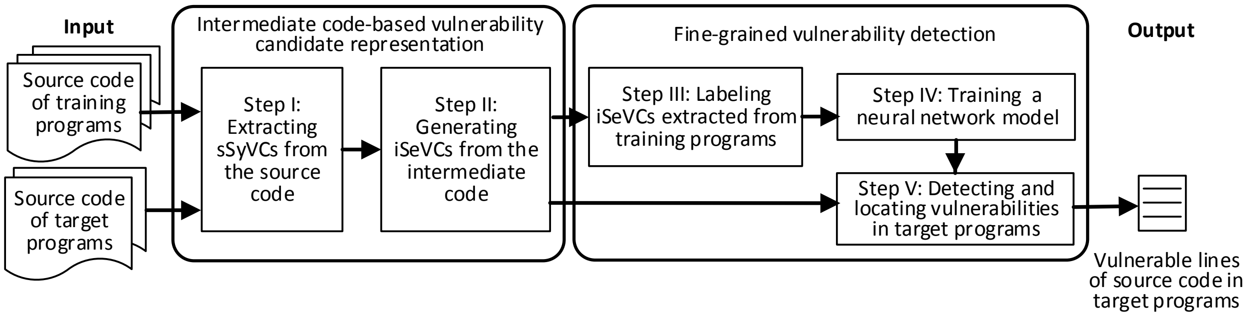

Fig. 3. Overview of VulDeeLocator with two components: *intermediate code-based vulnerability candidate representation* and *fine-grained vulnerability detection*. The *learning* phase consists of Steps I-IV and the *testing* (i.e., detection) phase consists of Steps I, II, and V.

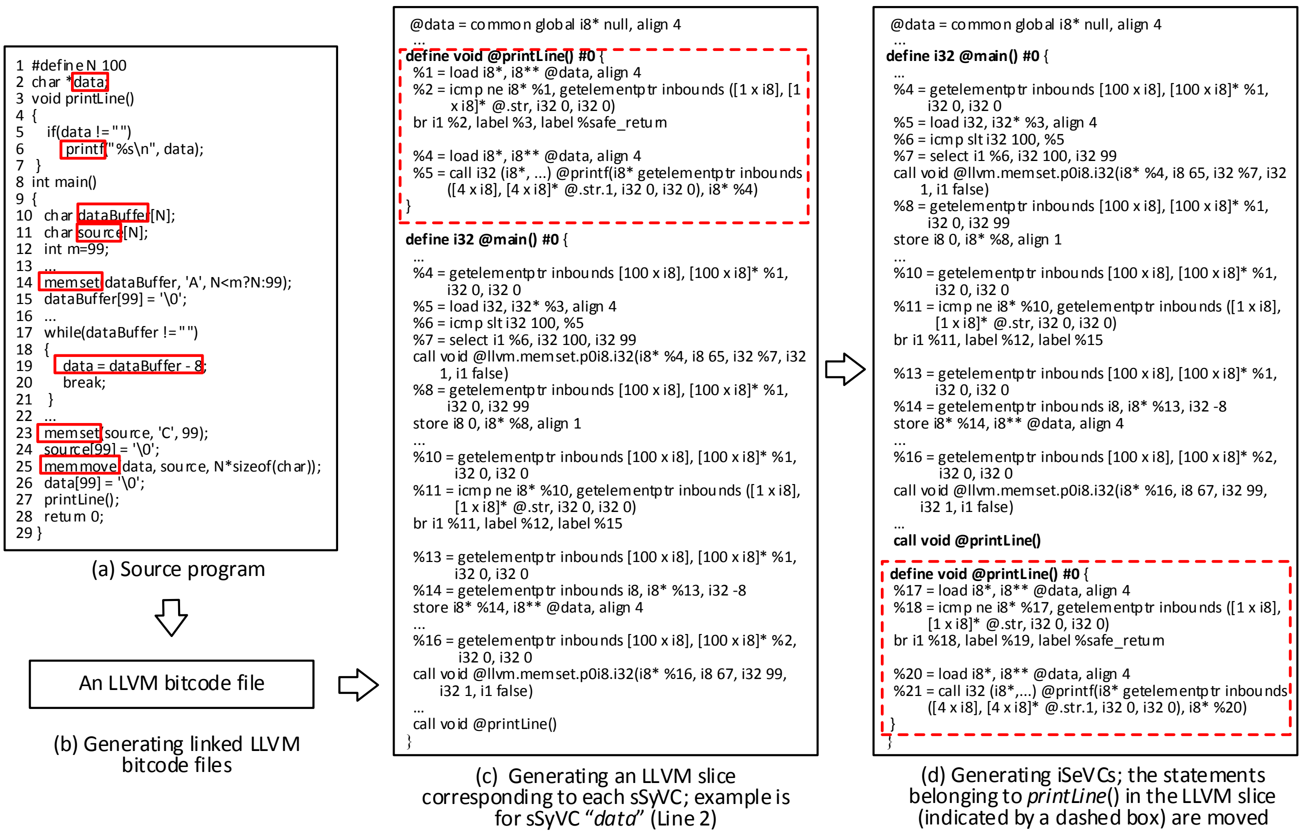

Fig. 4. (a): An example showing the sSyVCs (highlighted by boxes) that are extracted from the program in Figure 2; (b)-(d): An example showing the generation of iSeVCs from the sSyVC "*data*" (Line 2).

Representation (LLVM IR) [28], because it is widely used for C programs. We propose Algorithm 1 for generating iSeVCs from sSyVCs and the LLVM intermediate code; this algorithm can be adapted to accommodate other intermediate code. Corresponding to the afore-presented Principles, Algorithm 1 has three components: generating linked LLVM bitcode files; generating LLVM slices corresponding to sSyVCs; and generating iSeVCs.

Generating linked LLVM bitcode files (enforcing Principle 1). This component generates one or multiple linked LLVM bitcode files from source programs as follows (Lines 4-12): (i) use the *Clang* compiler [29] to generate LLVM bitcode files; (ii) link the LLVM bitcode files according to their dependency relationships, leading to one or multiple linked LLVM bitcode files. Figure 4(b) illustrates the idea using the example sSyVC "*data*" in Line 2 of the source program described in Figure 4(a).

Generating LLVM slices corresponding to sSyVCs and generating iSeVCs together (enforcing Principle 2). For each sSyVC extracted from Step I, one can generate an LLVM slice corresponding to the sSyVC as follows (Lines 13-20):

- Generate the dependence graph by computing the control dependencies and data dependencies from the linked LLVM bitcode file.
- Slice the dependence graph according to each sSyVC [23] such that (i) in an LLVM slice, each local variable is represented as a numeric value with prefix "%", and (ii) for each function in the LLVM slice, the numeric value for the first local variable is "1", and the numeric value is increased by 1 for each local variable thereafter.

As an example, we mention that tools like *dg* [30] can generate LLVM slices corresponding to sSyVCs. Figure 4(c)

Algorithm 1 Generating iSeVCs from the intermediate code

Input: A source program $P = \{p_1, \ldots, p_n\}$; a set $Y = \{y_{i,j,k,z}\}$ of sSyVCs
Output: The set of iSeVCs E

1: $E \leftarrow \emptyset$; {the set of iSeVCs}
2: $P' \leftarrow \emptyset$; {the set of LLVM bitcode files corresponding to source program files in P}
3: $B \leftarrow \emptyset$; {the set of linked LLVM bitcode files for P}
4: **for** each $p_i \in P$ **do**
5: Use *Clang* to compile p_i to an LLVM bitcode file p'_i;
6: $P' \leftarrow P' \cup \{p'_i\}$;
7: **end for**
8: Group the LLVM bitcode files in P' by dependency relationships;
9: **for** each group G_μ **do**
10: Link the LLVM bitcode files in G_μ to an LLVM bitcode file b'_μ;
11: $B \leftarrow B \cup \{b'_\mu\}$;
12: **end for**
13: **for** each $y_{i,j,k,z} \in Y$ **do**
14: **for** each $b'_\mu \in B$ **do**
15: **if** the bitcode file corresponding to p_i is linked to b'_μ **then**
16: Generate the LLVM slice $e_{i,j,k,z}$ corresponding to $y_{i,j,k,z}$ from b'_μ;
17: $E \leftarrow E \cup \{e_{i,j,k,z}\}$;
18: **end if**
19: **end for**
20: **end for**
21: **for** each $e_{i,j,k,z} \in E$ **do**
22: **for** each function f'_γ called by function f'_α **do**
23: The statements in $e_{i,j,k,z}$ of f'_γ are appended to the statement (in f'_α) calling function f'_γ;
24: Modify each numeric value with prefix "%" in the appended statements to a new numeric value that has not been used in f'_α;
25: **end for**
26: **end for**
27: **return** E;

illustrates that the LLVM slice corresponding to sSyVC "*data*".

Finally, iSeVCs (i.e., program statements that are semantically related to sSyVCs) are generated as follows (Lines 21-26). For each function f'_γ that is called by function f'_α, the statements in the LLVM slice of function f'_γ are appended to the statement (in function f'_α) that calls function f'_γ. This is to preserve the order of statements that possibly belong to different functions but are related to each other (because of some control dependency and/or data dependency).

Figure 4(d) illustrates that the statements in the LLVM slice of function *printLine*, which is highlighted by the dashed box, are appended to the statement "*call void @printLine()*" in the calling function *main*. In order to avoid assigning the same numeric value to different local variables in the LLVM slices of different functions, each numeric value with prefix "%" (i.e., local variable) in the appended statements is modified to a new numeric value that has not been used in the calling function. In the example shown in Figure 4(d), the local variable "%1" in the LLVM slice of function *printLine* is modified to "%17" because "16" is the last assigned numeric value in function *main*, which is shown in Figure 4(c).

Remark. The inadequacy of source code-based representations in conveying semantic information, which is shown in the two examples discussed in Section 2.1, is overcome by the use of intermediate code. Specifically, the iSeVC shown in Figure 4(d) involves statements "*%6=icmp slt i32 100, %5*" and "*%7=select i1 %6, i32 100, i32 99*" in the intermediate code; these statements expose the control flow of the conditional operator that cannot be conveyed by source code-based representations. Moreover, the iSeVC shown in Figure 4(d) can identify that "*N*" in "*N<m?N:99*" (Line 14) takes the value "*100*" because of the corresponding intermediate code "*%6=icmp slt i32 100, %5*" and "*%7=select i1 %6, i32 100, i32 99*".

5 Fine-grained Vulnerability Detection

5.1 Requirements for Fine-grained Vulnerability Detectors

We propose the following three requirements for neural network models that aim to detect and locate vulnerabilities. These requirements are centered at the novel idea of *granularity refinement*.

- **Requirement 1: Granularity refinement**. In order to pin down vulnerabilities, the granularity of the output of a neural network vulnerability detector should be finer than that of the input.
- **Requirement 2: Easy mapping**. It should be easy to map the output of a neural network (at a refined granularity) back to the iSeVCs to pinpoint vulnerabilities. The output should be a sequence of tokens, where one or multiple consecutive tokens correspond to a same line of code in the intermediate code. These lines of intermediate code can be easily mapped back to iSeVCs, and therefore the vulnerable lines of code in source programs.
- **Requirement 3: Attention taking**. Some parts of an iSeVC may be more important than others and thus should be paid more attention by a neural network; i.e., it should take advantage of the "attention" (i.e., more important parts) highlighted in an input.

5.2 Labeling iSeVCs

We label each iSeVC from training programs as follows: If an iSeVC contains a known vulnerability, the iSeVC is labeled with the line number(s) of the vulnerability in the iSeVC (i.e. location of the vulnerability), denoted by $x_1, \ldots, x_\zeta$ where x_ϵ ($1 \le \epsilon \le \zeta$) is a line number corresponding to the vulnerability; otherwise, the iSeVC is labeled as "0" (i.e., containing no vulnerability). Since a vulnerability dataset should provide the locations (e.g., line numbers) of vulnerabilities in the intermediate code of a source program, these line numbers of vulnerabilities in the source program need to be mapped to the line numbers in the intermediate code, which can be done simply by leveraging a textual LLVM file that comes with debugging information [31].

5.3 Training a Neural Network Model

Each iSeVC needs to be encoded into a vector, which is used as an input to a neural network. In order to make iSeVCs independent of user-defined function names while capturing program semantic information, this step maps user-defined function names to symbolic names (e.g., "FUN1", "FUN2") in a one-to-one fashion. It is worth mentioning that iSeVCs are already independent of local variable names because the latter are replaced with symbolic names in the intermediate code. Then, a *word embedding* method can be used to encode iSeVCs into vectors. Since the lengths of the resulting vectors (representing iSeVCs) can be different and a neural network takes input vectors of a fixed-length θ, these vectors may need to be adjusted as follows: If a vector is shorter than θ, zeroes are padded to the end of the vector; if a vector is longer than θ, the vector is truncated to length θ to make the sSyVC appear in the middle of the resulting vector [18]. Finally, the vectors are used as the input to a neural network that satisfies the aforementioned Requirements 1-3 (i.e., granularity refinement, easy mapping, and attention taking). In what follows, we elaborate the neural network BRNN-vdl we propose, which satisfies the aforementioned Requirements 1-3.

5.3.1 BRNNs achieve easy mapping

One may suggest to use Recurrent Neural Networks (RNNs), such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) [32], [33], to achieve *easy mapping*. This seems reasonable because RNNs are effective in coping with sequential data and the output at each time step corresponds to a token in an iSeVC, which makes it easy to map the output back to iSeVCs. However, unidirectional RNNs are not sufficient because a statement in a program may be affected by some *preceding* and/or *subsequent* statements in the program. Nevertheless, BRNNs, such as Bidirectional LSTM (BLSTM) and Bidirectional GRU (BGRU) can indeed achieve *easy mapping*, while accommodating *preceding* and *subsequent* statements.

BRNNs cannot achieve the other two properties because their output granularity is the same as, rather than refining, the input granularity because they treat every part of an input equally. For vulnerability detection, some parts (i.e., vulnerable lines of code) of an iSeVC may be more important than the other parts of the iSeVC and should be paid more *attention* by neural networks.

5.3.2 BRNN-vdl: A novel variant of BRNN further achieving attention taking and granularity refinement

Figure 5 highlights the structure of BRNN-vdl, which extends the standard BRNN with three extra layers that formulate the "vdl" part to achieve granularity refinement and attention taking. The input to BRNN-vdl includes (i) the vectors that represent the iSeVCs, and (ii) a vulnerability location matrix that represents the locations of vulnerabilities in each vector. The learning phase outputs a BRNN-vdl with fine-tuned parameters. In what follows, we briefly review BRNN and then describe the three extra layers in BRNN-vdl we introduce.

Overview of the BRNN component in BRNN-vdl. As shown in Figure 5, the *standard BRNN* has (i) a number of BRNN layers, which connect the RNN cells (e.g., LSTM and GRU) in both forward and backward directions, (ii) a dense layer, which reduces the number of dimensions of the vectors received from the BRNN layers, and (iii) an activation layer, which uses an activation function to generate the output at a time step. In the context of the present paper, the input is the vectors representing the labelled iSeVCs. Each time step corresponds to a token in an iSeVC. At time step τ, where $1 \leq \tau \leq \lambda$ and λ is the number of tokens in

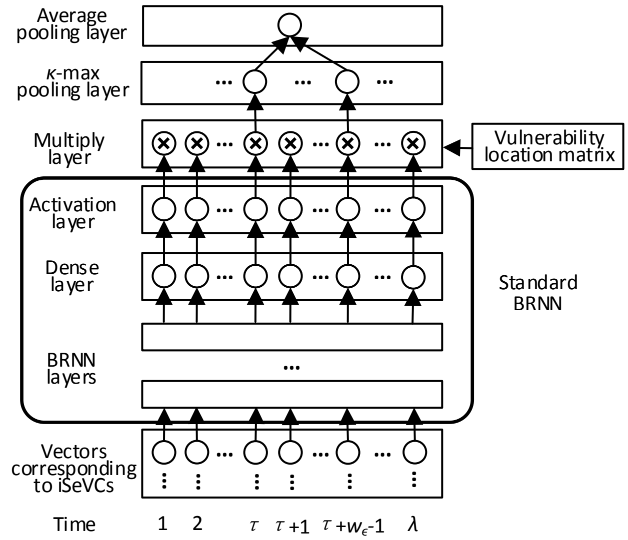

Fig. 5. BRNN-vdl extends BRNN with three extra layers (i.e., the *multiply*, κ*-max pooling*, and *average pooling* layers) that formulate the "vdl" part to achieve three desired properties.

each iSeVC, the output of the BRNN layers for iSeVC $e_{i,j,k,z}$, denoted by $g_\tau(e_{i,j,k,z})$, is

$$g_\tau(e_{i,j,k,z}) = \phi(g_{\tau-1}(e_{i,j,k,z}), g_{\tau+1}(e_{i,j,k,z}), e_{i,j,k,z}, \boldsymbol{\omega}, \boldsymbol{\beta}), \tag{1}$$

where $\boldsymbol{\omega}$ is a weight vector, $\boldsymbol{\beta}$ is a bias vector, $g_{\tau-1}(e_{i,j,k,z})$ and $g_{\tau+1}(e_{i,j,k,z})$ are respectively the output of the BRNN layers at time steps $\tau - 1$ and $\tau + 1$, and function ϕ indicates that the output of BRNN layers is represented by its parameters that include $g_{\tau-1}(e_{i,j,k,z})$, $g_{\tau+1}(e_{i,j,k,z})$, $e_{i,j,k,z}$, $\boldsymbol{\omega}$, and $\boldsymbol{\beta}$. How these parameters exactly interact with each other depends on the RNN cells, such as LSTM and GRU (see [32], [33] for more information). For iSeVC $e_{i,j,k,z}$, the output vector of the standard BRNN $\boldsymbol{A_{i,j,k,z}}$ (i.e., the output vector of the activation layer) is denoted by

$$\boldsymbol{A_{i,j,k,z}} = (g_1(e_{i,j,k,z}), \ldots, g_\lambda(e_{i,j,k,z})). \tag{2}$$

The multiply layer achieves *attention taking*. For the iSeVCs that are vulnerable, the multiply layer is meant to select the outputs of the tokens that correspond to the vulnerable lines of code. These selected outputs will be used in the subsequent layers and the back propagation process of BRNN-vdl because they would help locate vulnerabilities with a higher precision (than not using this multiply layer). For the iSeVCs that are not vulnerable, the multiply layer is meant to select *all* outputs of the tokens and use them in the subsequent layers and the back propagation process of BRNN-vdl, because these tokens are equally important as far as the learning phase is concerned.

Formally, for iSeVC $e_{i,j,k,z}$, the multiply layer multiplies the output vector of the activation layer $\boldsymbol{A_{i,j,k,z}}$ with the vulnerability location matrix $\boldsymbol{L_{i,j,k,z}}$. The output vector of the multiply layer $\boldsymbol{M_{i,j,k,z}}$ is denoted by

$$\boldsymbol{M_{i,j,k,z}} = \boldsymbol{A_{i,j,k,z}} \boldsymbol{L_{i,j,k,z}}, \tag{3}$$

where $\boldsymbol{L_{i,j,k,z}}$ is a diagonal matrix with $\boldsymbol{L_{i,j,k,z}} = diag(\alpha_1, \alpha_2, \ldots, \alpha_\lambda)$. For a vulnerable iSeVC, let us denote by x'_ϵ the location of the first token in the vulnerable line x_ϵ for some $1 \leq \epsilon \leq \zeta$, and by w_ϵ the number of tokens in x_ϵ.

The value of α_φ ($1 \le \varphi \le \lambda$) is determined as follows: For the iSeVCs that are vulnerable, if $\varphi \in \{x'_\epsilon, \ldots, x'_\epsilon + w_\epsilon - 1\}$, then we set $\alpha_\varphi = 1$; otherwise, we set $\alpha_\varphi = 0$. For the iSeVCs that are not vulnerable, we set $\alpha_\varphi = 1$ for $1 \le \varphi \le \lambda$. This design choice can be justified as follows: For each iSeVC that is not vulnerable, all tokens should be equally treated because no line of code is vulnerable. However, a vulnerable iSeVC contains (i) one or multiple vulnerable lines of code, which should be highlighted for vulnerability locating purposes, and (ii) possibly a large number of lines of code that are not vulnerable, which only provide the context for vulnerability detection. If all tokens in a vulnerable iSeVC are equally treated, a false-negative can occur because most lines of code are not vulnerable.

The κ-max pooling layer and the average pooling layer together achieve *granularity refinement*. The *κ-max pooling* layer is meant to select the κ largest values among the elements in the output vector of the multiply layer $\boldsymbol{M_{i,j,k,z}}$. The *average pooling* layer is meant to compute the average of the outputs of the κ-max pooling layer. Intuitively, these two layers together achieve granularity refinement because (i) they further select the outputs of the multiply layer to obtain the output corresponding to each iSeVC, which is used for back propagation, and (ii) they take into account both the maximum and the average.

Formally, for an iSeVC $e_{i,j,k,z}$, the output of average pooling layer $o_{i,j,k,z}$ is defined as

$$o_{i,j,k,z} = ave(max_\kappa(\boldsymbol{M_{i,j,k,z}})), \quad (4)$$

where function max_κ returns the κ largest elements in the vector, and function ave returns the average of the κ largest elements. After conducting iterative forward and backward propagations, the training process converges to a BRNN-vdl with fine-tuned parameters, which encodes vulnerability patterns in the training data.

5.4 Detecting and Locating Vulnerabilities

Figure 6 highlights using the learned BRNN-vdl to detect and locate vulnerabilities in target programs. The input is the vectors representing the iSeVCs extracted from the target programs. For generating outputs, we first compute the average of the κ largest values for the tokens in each line. Then, we extract the lines whose output is larger than the threshold ϑ, leading to vulnerable iSeVCs and lines of code. Finally, we map these vulnerable lines of code to the vulnerable lines of source code as the output of the detection phase.

6 Experiments and Results

Our experiments use a machine with a NVIDIA GeForce GTX 1080 GPU and an Intel Xeon E5-1620 CPU operating at 3.50GHz.

6.1 Research Questions

We gear our experiments towards answering the following four Research Questions (RQs):

- RQ1: Can intermediate code-based vulnerability candidate representation be leveraged to achieve a substantially higher vulnerability detection capability?

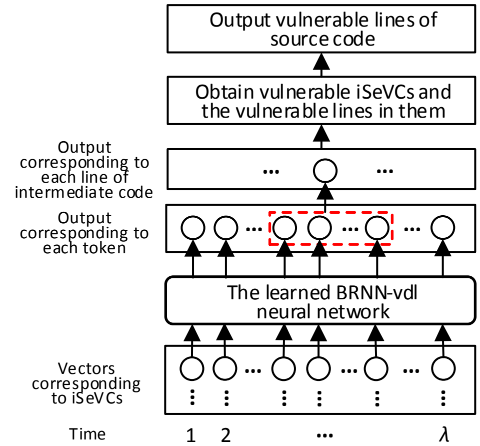

Fig. 6. Using the learned BRNN-vdl to detect vulnerabilities in target programs, where the dashed box highlights the tokens extracted from a line of code.

- RQ2: Can BRNN-vdl achieve a substantially higher vulnerability locating precision than BRNN?
- RQ3: How effective and precise is VulDeeLocator in detecting and locating vulnerabilities of target programs with known ground truth?
- RQ4: How effective and precise is VulDeeLocator in detecting and locating vulnerabilities of real-world software products for which we do not know whether they contain vulnerabilities or not?

6.2 Evaluation Metrics

We propose using five standard metrics to measure the detection capability of a vulnerability detector (see, e.g., [34]). Let TP denote the number of vulnerable samples that are detected as vulnerable (i.e., true-positives), FP denote the number of samples that are not vulnerable but are detected as vulnerable (i.e., false-positives), TN denote the number of samples that are not vulnerable and are not detected as vulnerable (i.e., true-negatives), FN denote the number of vulnerable samples that are not detected as vulnerable (i.e., false-negatives). The five metrics are: (i) false-positive rate $FPR = \frac{\text{FP}}{\text{FP+TN}}$; (ii) false-negative rate $FNR = \frac{\text{FN}}{\text{TP+FN}}$; (iii) accuracy $A = \frac{\text{TP+TN}}{\text{TP+FP+TN+FN}}$; (iv) precision $P = \frac{\text{TP}}{\text{TP+FP}}$; (v) F1-measure $F1 = \frac{2 \cdot P \cdot (1-FNR)}{P+(1-FNR)}$, or the overall effectiveness.

In order to evaluate the locating precision of a vulnerability detector, we propose using the standard Intersection over Union (IoU) metric with IoU $= \frac{|\text{U} \cap \text{V}|}{|\text{U} \cup \text{V}|}$, where U is the set of truly vulnerable lines of code and V is the set of detected vulnerable lines of code [35]. Figure 7 illustrates the meaning of IoU with respect to one iSeVC; as highlighted by boxes, U contains 4 statements, V contains 3 statements, $U \cap V$ contains 2 statements (i.e., $|U \cap V| = 2$), and $U \cup V$ contains 5 statements (i.e., $|U \cup V| = 5$), leading to IoU $= 2/5$. Intuitively, IoU reflects the degree at which the detected vulnerable statements overlap with the truly vulnerable statements. The closer the IoU is to 1, the higher the locating precision.

6.3 Preparing the Input to VulDeeLocator

We collect the source code of C programs from two vulnerability sources: the NVD [25] and the Software Assur-

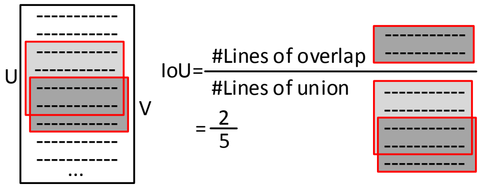

Fig. 7. An example illustrating the meaning of IoU, where a dashed line represents a program statement.

ance Reference Dataset (SARD) [36]. The programs collected from the NVD are accompanied by their *diff* files, which describe the difference between the programs before and after patching the vulnerabilities in question. The programs collected from the SARD are accompanied by labels, which indicate whether they are vulnerable or not. Note that SARD contains production, synthetic, and academic programs (i.e., test cases). We filter out the programs that cannot be compiled into the LLVM intermediate code. We also filter out those programs whose length is less than 500 lines of code, which are not so useful for our purposes because they are mainly simple synthetic programs that contain limited functionalities.

For training purpose, we collect 10,246 programs that may or may not be vulnerable, including 382 programs from the NVD and 9,864 from the SARD. The training set contains 11 types of vulnerabilities: CWE-20, CWE-78, CWE-119, CWE-121, CWE-122, CWE-124, CWE-126, CWE-127, CWE-134, CWE-189, and CWE-399, where each type is uniquely identified by a Common Weakness Enumeration IDentifier (CWE ID) [37]. For testing purpose, we randomly collect 2,561 programs from the SARD as the target programs with known ground truth, meaning an 80:20 ratio of training vs. testing data. The 2,561 programs involve (i) 2,038 programs containing 7 (of the 11) types of vulnerabilities mentioned above, and (ii) 523 programs containing 5 types of vulnerabilities (i.e., CWE-194, CWE-195, CWE-197, CWE-590, and CWE-690), which are however not contained in any of the training programs.

6.4 Intermediate Code-based Vulnerability Candidate

Extracting sSyVCs. In order to extract sSyVCs from the source code, we use *Clang* [29] to generate ASTs from a source program. Then, we traverse the ASTs to generate sSyVCs. For obtaining vulnerability syntax characteristics, we leverage the C vulnerability rules of the commercial tool Checkmarx [7] because we found that the syntax characteristics from these rules have a good coverage over known vulnerabilities. This leads to the four kinds of vulnerability syntax characteristics mentioned above, namely Library/API Function Call (FC), Array Definition (AD), Pointer Definition (PD), and Arithmetic Expression (AE). These characteristics cover 9,884 (i.e., 98.3%) of the vulnerable programs collected from the NVD and the SARD.

Take vulnerabilities of the FC-kind (i.e., Library/API function call) as an example, the syntax characteristic is that the type of a node on the AST in question is function call, the function name matches a library/API function name, and at least one argument of the function call is a variable. Given the ASTs of a source program, the matching algorithm proceeds as follows: (i) we first traverse the ASTs to identify the nodes whose type is "CxCursor_CallExpr" (meaning a function call); (ii) identify the nodes whose token matches a library/API function name (e.g., memset); (iii) traverse the children of the node corresponding to a library/API function call to identify the nodes whose type is "CxCursor_DeclRefExpr" (meaning a variable argument); and (iv) a library/API function call together with its variable arguments is extracted as a sSyVC. In total, we extract 119,782 sSyVCs, including 29,782 sSyVCs of the FC-kind, 29,273 sSyVCs of the AD-kind, 38,063 sSyVCs of the PD-kind, and 22,664 sSyVCs of the AE-kind from the training programs.

Generating iSeVCs. We use tool *dg* [30] to generate LLVM-based intermediate code slices corresponding to given source code sSyVCs as follows. For each given source code sSyVC, the corresponding iSeVC is a slice of intermediate code statements consisting of: (i) the intermediate code statements corresponding to the given source code sSyVC; and (ii) any intermediate code statement that has a data-dependence or control-dependence with any of the variables that are used or defined in the given sSyVC. Note that the statements in (ii) may belong to different functions or files than the one to which the given sSyVC belongs. Take the sSyVC "*data*" (Line 2) in Figure 4(a) as an example, the LLVM-based intermediate code slices (i.e., LLVM slices) corresponding to "*data*" generated by *dg* is described in Figure 4(c). Then, we use Algorithm 1 to generate iSeVCs. In total, we extract 119,782 iSeVCs from training programs. The second column of Table 1 summarizes the number of iSeVCs in each kind of sSyVCs.

TABLE 1
The total number (#) of iSeVCs, vulnerable iSeVCs, and non-vulnerable iSeVCs (i.e., not vulnerable) extracted from the training programs.

Kind of sSyVCs	#iSeVCs	#vul. iSeVCs	#non-vul. iSeVCs
FC-kind	29,782	6,606	23,176
AD-kind	29,273	8,381	20,892
PD-kind	38,063	11,089	26,974
AE-kind	22,664	4,125	18,539
Total	119,782	30,201	89,581

6.5 Fine-grained Vulnerability Detection

Labeling iSeVCs. For the iSeVCs extracted from programs collected from NVD, we focus on the vulnerabilities that are accompanied by *diff* files that involve *line deletion* or *line movement* because they allow us to pin down the locations of these vulnerabilities (i.e., statements prefixed by "-" in a *diff* file). If (i) an iSeVC contains some intermediate code that corresponds to one or multiple statements prefixed by "-" in the *diff* file and (ii) the program in question contains a known vulnerability, then the iSeVC is labeled as the line number(s) of the vulnerability in the intermediate code; otherwise, the iSeVC is labeled as "0" (i.e., not vulnerable).

For iSeVCs extracted from programs collected from the SARD, if an iSeVC contains some intermediate code that corresponds to one or multiple vulnerable statements in the source program, then the iSeVC is labeled as the line number(s) of the vulnerability in the intermediate code; otherwise, the iSeVC is labeled as "0" (i.e., not vulnerable). The third and fourth columns of Table 1 respectively list the numbers of vulnerable iSeVCs and non-vulnerable iSeVCs in each kind of sSyVCs.

Training, detecting and locating. In order to use neural networks, we need to encode iSeVCs into vectors. For this purpose, we first divide each iSeVC into a sequence of tokens via lexical analysis (e.g., “call”, “void”, “@”, “FUN1”, “(”, and “)”), and then transform each token to a fixed-length vector via *word2vec* tool [38]. Finally, a token-level vector for each iSeVC is obtained by concatenating the token-level vectors in sequence. Each token is encoded into a vector of length 30, and each iSeVC is represented by a vector of length θ=27,000, which means that the first 900 tokens of an iSeVC are considered.

We implement the BRNN-vdl in Python using TensorFlow [39] together with Keras [40]. We use a 5-fold cross validation to train the BRNN-vdl and choose the parameter values that lead to the highest F1-measure. We implement two instances of BRNN: one is BLSTM, which leads to “VulDeeLocator-BLSTM”; the other is BGRU, which leads to “VulDeeLocator-BGRU”. Take VulDeeLocator-BGRU as an example, the trained hyper-parameters are: output dimension is 512; the number of hidden layers is 2; the number of hidden nodes at each layer is 900; batch size is 16; minibatch stochastic gradient descent together with ADAMAX [41] is used; learning rate is 0.002; dropout is 0.4; the number of epochs is 10; and $\kappa = 1$.

For detecting vulnerabilities in target programs, we first compute the average of the κ largest values among the tokens in each line of intermediate code. Then, we extract the lines whose output is larger than threshold ϑ (e.g., 0.5). These lines of code are the vulnerable ones, and are mapped back to the vulnerable lines of source code as the output of the testing (i.e., detection) phase.

6.6 Experiments for Answering RQ1

In order to evaluate the advantages of intermediate code-based vulnerability candidate representation over source code-based one, we conduct experiments with the following two vulnerability candidate representations:

- *source code- and Semantics-based Vulnerability Candidate* (sSeVC): A sSeVC is a sequence of source code statements that have some data-dependence or control-dependence with a sSyVC (i.e., source code- and Syntax-based Vulnerability Candidate, as defined in Section 2.2) and can be obtained by using a source code static analysis tool (e.g., *Joern* [26]).
- iSeVC: An iSeVC is a sequence of intermediate code statements that have some data-dependence or control-dependence with a sSyVC. Compared with source code-based representation, iSeVC is in the static single assignment (i.e., SSA) form, which assures that each variable is defined-and-then-used and is assigned exactly once.

We report the experimental results with VulDeeLocator-BGRU, while noting that experimental results with VulDeeLocator-BLSTM are similar. Table 2 summarizes the comparison. We observe that iSeVCs lead to better results than sSeVCs, including a 1.7% improvement in false-positive rate, a 28.8% improvement in false-negative rate, a 9.9% improvement in accuracy, a 3.2% improvement in precision, and a 18.3% improvement in F1-measure. This can be attributed to the two advantages of intermediate code-based representation: (i) intermediate code is in the SSA form, which can expose more information about control-flows and the define-use relations between variables; (ii) intermediate code-based vulnerability candidates can capture more semantic information (e.g. the relations between the definitions of types or macros and their uses), which however may not be identified by sSeVCs. This is justified by the following two examples.

TABLE 2
Vulnerability detection capability of VulDeeLocator-BGRU using two different kinds of vulnerability candidate representations, indicating that intermediate code-based representation is more effective than source code-based representation.

Vulnerability candidate	Representation	FPR (%)	FNR (%)	A (%)	P (%)	F1 (%)
sSeVC	Source code-based	2.2	32.8	86.1	94.9	78.7
iSeVC	Intermediate code-based	0.5	4.0	96.0	98.1	97.0

```
1 ...
2 #define COMMAND_ARG2 "ls "
3 #define COMMAND_ARG3 data
4 #define EXECL execl
5 ...
6 void CWE78_OS_Command_Injection__char_console_execl_12_bad()
7 {
8    char * data;
9    char dataBuffer[100] = COMMAND_ARG2;
10   data = dataBuffer;
11   ...
12   size_t dataLen = strlen(data);
13   ...
14   if (fgets(data+dataLen, (int)(100-dataLen), stdin) != NULL)
15   {
16       dataLen = strlen(data);
17       if (dataLen > 0 && data[dataLen-1] == '\n')
18       {
19           data[dataLen-1] = '\0';
20       }
21   }
22   ...
23   EXECL(COMMAND_INT_PATH, COMMAND_INT_PATH, COMMAND_ARG1,
             COMMAND_ARG2, COMMAND_ARG3, NULL);
24 }
25 ...
```

(a) Test case 244486

```
1 ...
2 static char * CWE127_Buffer_Underread__malloc_char_memcpy_45_badData;
3 ...
4 static void badSink()
5 {
6   char * data = CWE127_Buffer_Underread__malloc_char_memcpy_45_badData;
7   char dest[100];
8   memset(dest, 'C', 100-1);
9   dest[100-1] = '\0';
10  memcpy(dest, data, 100*sizeof(char));
11  dest[100-1] = '\0';
12  ...
13 }
14 void CWE127_Buffer_Underread__malloc_char_memcpy_45_bad()
15 {
16  char * data;
17  data = NULL;
18  char * dataBuffer = (char *)malloc(100*sizeof(char));
19  ...
20  memset(dataBuffer, 'A', 100-1);
21  dataBuffer[100-1] = '\0';
22  data = dataBuffer - 8;
23  CWE127_Buffer_Underread__malloc_char_memcpy_45_badData = data;
24  badSink();
25 }
26 ...
```

(b) Test case 234895

Fig. 8. Two examples of vulnerabilities that are missed by VulDeeLocator-BGRU trained from sSeVCs.

Figure 8(a) shows one example, which is dubbed *test case 244486* because it is derived from the SARD dataset [36]. This example contains an OS command injection vulnerability because the input is received from the console and is used without validating it (vulnerable Line 23). Consider sSyVC “*data*” in Line 8. Since the source code parsing (e.g. when using *Joern* [26]) cannot deal with macro definitions, “COMMAND_ARG3” in Line 23 cannot be identified as "*data*". As a consequence, the corresponding sSeVC fails to identify the vulnerable statement in Line 23. This explains the false-negative. On the other hand, the iSeVCs can identify the "COMMAND_ARG3" in Line 23 as "*data*" after compilation. This explains why the resulting model can detect the vulnerability.

Figure 8(b) shows another example, dubbed *test case 234895*. The example contains a buffer under-read vulnerability because the copy from a memory location may be located before the source buffer (vulnerable Line 10). Consider sSyVC "*data*" in Line 6. Since the source code parsing (e.g. when using *Joern* [26]) cannot deal with global variables, the sSeVC corresponding to sSyVC "*data*" in Line 6 does not contain the statements that are semantically related to sSyVC "*data*" via the global variable *CWE127_Buffer_Underread__malloc_char_memcpy_45_badData* in function *CWE127_Buffer_Underread__malloc_char_memcpy_45_bad*. The root cause of the vulnerability is that the data pointer points to a memory address that is different from the allocated memory buffer (Line 22), which is defined in function *CWE127_Buffer_Underread__malloc_char_memcpy_45_bad*. This explains why the vulnerability is missed. However, the model learned from iSeVCs can identify and accommodate these statements because they are semantically related to the global variable. In summary, we answer RQ1 with the following:

Insight 1.
VulDeeLocator leveraging intermediate code-based representation is substantially more effective than VulDeeLocator using source code-based representation, owing to the aforementioned two advantages of intermediate code-based representation.

6.7 Experiments for Answering RQ2

In order to see the capability of BRNN-vdl in locating vulnerabilities, we conduct experiments to compare BRNN-vdl and BRNN while using two types of vulnerability candidates (i.e., source code-based sSeVCs vs. intermediate code-based iSeVCs as specified in Section 6.6). In what follows, we report the experimental results of using BGRU to instantiate BRNN, while noting that similar results are observed when using BLSTM to instantiate BRNN.

TABLE 3
Comparing BRNN-vdl with BRNN (more specifically, BGRU-vdl vs. BGRU), where IoU is averaged over the IoUs measured between the detected vulnerable code and the ground-truth vulnerable code in the test data and |V| is the average number of detected vulnerable lines of source code.

Vulnerability candidate	Model	FPR (%)	FNR (%)	A (%)	P (%)	F1 (%)	IoU (%)	|V|
sSeVC	BRNN -vdl	2.2	32.8	86.1	94.9	78.7	29.9	3.4
	BRNN	8.4	28.1	84.1	84.1	77.5	7.4	14.8
iSeVC	BRNN -vdl	0.5	4.0	96.0	98.1	97.0	32.7	2.2
	BRNN	2.3	5.4	97.0	92.0	93.3	10.1	19.9

Table 3 presents the comparison. For locating vulnerabilities, BRNN-vdl achieves, on average, a 22.6% higher IoU than BRNN because the number of detected vulnerable lines of code is 2.8 for BRNN-vdl and 17.4 for BRNN on average. This can be explained by the fact that BRNN preserves the input granularity in its output, while BRNN-vdl reduces the input lines of code to much smaller lines of code in its output. This "granularity refinement" is accomplished by the "vdl" part. In terms of vulnerability detection capability, BRNN-vdl is better than BRNN, with a 6.2% lower false-positive rate and a 1.2% higher F1-measure at the price of a 4.7% higher false-negative rate when using sSeVCs as vulnerability candidates, and with a 1.8% lower false-positive rate, a 1.4% false-negative rate, and a 3.7% higher F1-measure when using iSeVCs as vulnerability candidates. This means that "vdl" can somewhat improve the vulnerability detection capability. This leads to:

Insight 2. BRNN-vdl achieves a substantially higher vulnerability locating precision and a somewhat higher vulnerability detection capability than BRNN.

6.8 Experiments for Answering RQ3

In order to answer RQ3, we compare two instances of VulDeeLocator and some state-of-the-art vulnerability detectors in terms of their capabilities in detecting and locating vulnerabilities in target programs with known ground truth.

6.8.1 Comparing VulDeeLocators with two instances of BRNN (i.e., BLSTM vs. BGRU)

Table 4 highlights the experimental results. We observe that VulDeeLocator-BGRU simultaneously achieves a 3.8% lower false-negative rate, a 1.8% higher F1-measure, and a 5.6% higher IoU than VulDeeLocator-BLSTM. The higher locating precision may be attributed to the fact that BGRU uses fewer parameters, possibly making it easier to "refine" the output. The training time and detection time of VulDeeLocator-BLSTM are respectively 51,696 seconds and 1,389 seconds, which are much longer than the training time (32,713 seconds) and detection time (1,325 seconds) of VulDeeLocator-BGRU. This is because BGRU uses fewer parameters and would converge faster.

TABLE 4
Effectiveness of VulDeeLocator-BLSTM, VulDeeLocator-BGRU, and state-of-the-art vulnerability detectors, where IoU is averaged over the IoUs measured between the detected vulnerable code and the ground-truth vulnerable code in the test data and |V| is the average number of detected vulnerable lines of source code.

Method	FPR (%)	FNR (%)	A (%)	P (%)	F1 (%)	IoU (%)	|V|
VulDeeLocator with two instances of BRNN							
VulDeeLocator -BLSTM	0.5	7.8	97.7	98.5	95.2	27.1	2.1
VulDeeLocator -BGRU	0.5	4.0	96.0	98.1	97.0	32.7	2.2
State-of-the-art vulnerability detectors							
Flawfinder	10.5	83.7	61.6	48.9	24.5	38.4	7.3
Checkmarx	72.8	54.4	39.1	27.9	34.6	23.8	4.1
Fortify	37.0	54.0	56.5	43.4	44.6	30.9	2.4
VulDeePecker	7.9	49.4	51.0	91.9	65.2	9.0	14.3
SySeVR	10.1	12.2	89.0	84.2	86.0	8.4	16.7

VulDeeLocator-BGRU detects all of the vulnerabilities in the 2,484 (of the 2,561) target programs, despite that 5 types of detected vulnerabilities did *not* appear in the training data. This justifies that pattern-based detector indeed can detect *some*, but not all (as elaborated below), vulnerabilities that are not in the training data. However, there are indeed false-negatives, which can be caused by inadequate coverage of vulnerability syntax characteristics and vulnerability types in the training data. For example, 19 false-negatives of VulDeeLocator-BGRU are not covered by the four kinds of vulnerability syntax characteristics, highlighting the importance of identifying as-complete-as-possible vulnerability syntax characteristics. Moreover, another 56 false-negatives of VulDeeLocator-BGRU are caused by the fact that their corresponding vulnerability types do not appear in the training data.

Insight 3. The coverage of vulnerability syntax characteristics and the coverage of vulnerability types in the training data are key to lowering false-negatives.

6.8.2 Comparing with state-of-the-art pattern-based vulnerability detectors

For source code- and rule-based vulnerability detectors, we consider the open source tool Flawfinder [6] and the commercial product Checkmarx [7]. For intermediate code- and rule-based vulnerability detectors, we consider the commercial product Fortify [8]. For deep learning-based vulnerability detectors, we consider VulDeePecker [17], which is designed to detect vulnerabilities related to library/API function calls, and SySeVR [18], which is designed to detect multiple types of vulnerabilities. The implementations of these two tools are obtained from their authors (via private communications). We choose these systems for comparison because they are the state-of-the-art and/or available to us.

Table 4 summarizes the comparison. We make the following observations. **(i)** Source code- and rule-based vulnerability detector Flawfinder incurs prohibitively high false-negative rate, which can be attributed to the inadequacy of its parser and patterns [42]. **(ii)** Source code- and rule-based vulnerability detector Checkmarx incurs prohibitively high false-positive rate and false-negative rate, which can be attributed to the inadequacy of the rules defined by human experts. This justifies why we only use the Checkmarx rules to extract sSyVCs as a starting point for vulnerability detection. **(iii)** Intermediate code- and rule-based vulnerability detector Fortify incurs very high false-positive rate and false-negative rate, suggesting that rules based on intermediate code can indeed accommodate more useful information than rules based on source code. **(iv)** Deep learning-based detector VulDeePecker is much less effective than deep learning-based SySeVR because the former can only cope with the class of vulnerabilities related to library/API function calls [17], but SySeVR can cope with multiple classes of vulnerabilities [18]. **(v)** VulDeeLocator-BGRU achieves respectively an 11.0%, 9.6%, and 8.2% improvement over SySeVR in F1-measure, false-positive rate, and false-negative rate, because the former can accommodate more semantic information conveyed by intermediate code. **(vi)** Rule-based vulnerability detectors (i.e., Flawfinder, Checkmarx, and Fortify) achieve an IoU of 31.0% on average, but their low overall effectiveness (F1-measure) hinders their usefulness. **(vii)** IoUs of VulDeeLocator-BLSTM (27.1%) and VulDeeLocator-BGRU (32.7%) are much higher than IoUs of VulDeePecker (9.0%) and SySeVR (8.4%), because the average number of detected vulnerable lines of code is 2.1 for VulDeeLocator-BLSTM, 2.2 for VulDeeLocator-BGRU, 14.3 for VulDeePecker, and 16.7 for SySeVR. The higher vulnerability locating precision can be attributed to the vdl-part of BRNN-vdl. In summary, we draw:

Insight 4. VulDeeLocator is more effective than the state-of-the-art pattern-based vulnerability detectors in detecting and locating vulnerabilities. In particular, VulDeeLocator-BGRU achieves a 3.9X higher locating precision than the state-of-the-art vulnerability detector SySeVR.

6.9 Experiments for Answering RQ4

In order to answer RQ4, we apply VulDeeLocator-BGRU, which is the most effective instance of VulDeeLocator, to detect vulnerabilities in several versions of 3 software products (i.e., Linux kernel, FFmpeg, and Libav). Since we do not know whether these products contain vulnerabilities or not (i.e., the ground truth is not known), we select 200 program files of these software products as the test data, and manually examine and confirm the vulnerabilities detected from them. VulDeeLocator detects 16 vulnerabilities from these 200 program files, including 5 false positives. Among the 11 true positives, 7 vulnerabilities correspond to known vulnerabilities as shown in Table 5, but the other 4 vulnerabilities are not reported in the NVD as shown in Table 6. The average IoU, where average is over all of the detected vulnerabilities (including the 5 false positives), is 30.2%. For each vulnerability, the average number of detected vulnerable lines of code is 3.9. Among these 11 vulnerabilities, VulDeePecker [17] missed 9 vulnerabilities (which is not surprising because these 9 vulnerabilities are not related to library/API function call) and SySeVR [18] missed 7 vulnerabilities; moreover, the average number of detected vulnerable lines of code for VulDeePecker (40.5) and SySeVR (48.4) is much larger than the average number of detected vulnerable lines of code for VulDeeLocator (3.9).

TABLE 5
The 7 vulnerabilities detected by VulDeeLocator-BGRU (from 200 program files) that are reported in the NVD, where |V| is the number of detected vulnerable lines of source code.

Target product	CVE ID	Vulnerable file	IoU (%)	|V|	Kind of sSyVC
FFmpeg 0.8.2	CVE-2011-3973	.../cavsdec.c	40.0	4	PD
FFmpeg 0.9.4	CVE-2011-3934	.../vp3.c	50.0	2	PD
Linux kernel 2.6.29	CVE-2009-1527	.../ptrace.c	25.0	3	PD
	CVE-2011-1598	.../bcm.c	25.0	4	PD
	CVE-2011-3637	.../task_mmu.c	50.0	2	PD
Linux kernel 3.7.9	CVE-2014-3122	.../rmap.c	25.0	3	FC
	CVE-2014-5471	.../rock.c	31.3	4	FC

TABLE 6
The 4 vulnerabilities detected by VulDeeLocator-BGRU (from 200 program files) that are not reported in the NVD, where |V| is the number of detected vulnerable lines of source code.

Target product	Vulnerable file	IoU (%)	|V|	Kind of sSyVC	1st patched version
Libav 10.3	libavcodec/anonymized.c	33.3	3	PD	–
	libavcodec/anonymized.c	28.6	2	PD	–
	libavcodec/anonymized.c	25.0	2	PD	–
Libav 9.10	libavformat/matroskadec.c	50.0	3	PD	Libav 9.18

Figure 9 presents an example of vulnerability that is detected by VulDeeLocator-BGRU but missed by VulDeePecker [17] and SySeVR [18] in software product FFmpeg 0.9.4. This vulnerability corresponds to CVE-2011-3934 and is related to *double free* because the statement in Line 11 can cause a double release of variable "$s \rightarrow current\_frame$". Consider sSyVC "$s$" in Line 3. Since the source code parsing (e.g. when using *Joern* [26]) cannot deal with macro definitions, "$copy\_fields(s, s1, golden\_frame, current\_frame)$" in Line 11 cannot be identified as "$memcpy(\&s \rightarrow golden\_frame$, $\&s1 \rightarrow golden\_frame$, $(char*)\&s \rightarrow current\_frame$ - $(char*)\&s \rightarrow golden\_frame)$", meaning that the source code-based representation cannot identify the vulnerable statement in Line 11. This explains the false-negative of VulDeePecker and SySeVR. On the other hand, the iSeVC can convert the statement in Line 11 to "$memcpy(\&s \rightarrow golden\_frame$, $\&s1 \rightarrow golden\_frame$, $(char*)\&s \rightarrow current\_frame$ - $(char*)\&s \rightarrow golden\_frame)$;". This explains why VulDeeLocator can detect the vulnerability.

Table 6 highlights the four vulnerabilities detected by VulDeeLocator-BGRU that are not reported in the NVD. These vulnerabilities are caused by improper use of pointers and allow remote attackers to wage denial-of-service attacks. Specifically, the first vulnerability is caused by the lack of checking on memory allocation; the second and third ones are caused by the lack of properly validating the reduction factor or block length; the fourth one is caused by use-after-free. Among the four vulnerabilities mentioned above, three in Libav 10.3 are not known to exist until now and are confirmed by our manual examination. For ethical exposure purposes, we anonymize their vulnerable files but have reported the details to the vendor. The other vulnerability in Libav 9.10 is a use-after-free vulnerability related to a pointer named "$tracks$"; this vulnerability is not reported in the NVD but has been "silently" patched by the vendor when releasing newer versions of the product (with the first patched version being Libav 9.18). In summary, we draw:

Insight 5. VulDeeLocator can detect and pinpoint vulnerabilities in real-world software products.

```
1 static int vp3_update_thread_context(AVCodecContext *dst, const
  AVCodecContext *src)
2 {
3   Vp3DecodeContext *s = dst->priv_data, *s1 = src->priv_data;
4   ...
5   #define copy_fields(to, from, start_field, end_field) memcpy(&to->start_field,
    &from->start_field, (char*)&to->end_field - (char*)&to->start_field)
6
7   if (!s1->current_frame.data[0]
8       ||s->width != s1->width
9       ||s->height!= s1->height) {
10        if (s != s1)
11            copy_fields(s, s1, golden_frame, current_frame);
12      return -1;
13  }
14   if (s != s1) {
15       // init tables if the first frame hasn't been decoded
16       if (!s->current_frame.data[0]) {
17         ...
18       }
19      ...
20  #undef copy_fields
21  }
22   update_frames(dst);
23   return 0;
24 }
```

Fig. 9. The vulnerability detected by VulDeeLocator but not detected by any of VulDeePecker and SySeVR.

7 Limitations

This study has several limitations. **First**, the design of VulDeeLocator focuses on detecting vulnerabilities in C source programs because (i) we want to demonstrate the feasibility of VulDeeLocator and (ii) the tools we leverage happen to support C. Extending VulDeeLocator to accommodate other programming languages is an interesting future work. **Second**, VulDeeLocator requires to compile program source code into intermediate code, and cannot be used when a program source code cannot be compiled. **Third**, the four kinds of vulnerability syntax characteristics used by VulDeeLocator can cover 98.3% of vulnerable programs collected from NVD and SARD. This 98.3% coverage should be used with caution because (i) for the NVD data, we only use the lines of code that are deleted or moved in a diff file as the location of a vulnerability (i.e., we did not consider those vulnerabilities whose diff files only involve line additions), and (ii) the SARD data may not be representative of real-world software products. It is an open problem to identify more complete vulnerability syntax characteristics. **Fourth**, our case study uses BRNN-vdl to instantiate VulDeeLocator to demonstrate feasibility. Tailored neural networks need to be designed for vulnerability detection purposes. **Fifth**, we can partly explain the effectiveness of VulDeeLocator, but much more research needs to be done in this direction of *explainability*.

8 Related Work

Prior work on static vulnerability detection. The present study belongs to static vulnerability detection, which includes *code similarity-based* methods and *pattern-based* methods. Code similarity-based methods [2], [3], [4], [5] can achieve a high locating precision when they indeed detect vulnerabilities, but have a high false-negative rate because many vulnerabilities are not caused by code cloning [17]. Pattern-based vulnerability detection methods can be further divided into *rule-based* ones and *machine learning-based* ones. Rule-based methods use analyst-generated rules to detect vulnerabilities, including (i) open source tools (e.g., Flawfinder [6]) and commercial tools (e.g., Checkmarx [7]), which operate on program source code, and (ii) Fortify and Coverity [8], [9], which operate on intermediate code. These tools have high false-positives or false-negatives [42]. Machine learning-based methods aim to detect vulnerabilities using patterns learned from analyst-defined feature representations of vulnerabilities [12], [13], [14], [15], [16], [24] or "raw" feature representations via deep learning [19], [20], [21], [17], [18], [43]. These methods detect vulnerabilities at coarse granularities (e.g., programs [14], components [13], functions [12], [19], [21], [43], and code gadgets [17], [18]).

Among the detectors mentioned above, VulDeePecker [17] and SySeVR [18] are closely related to ours. However, these two detectors operate on program slices and source code-based vulnerability candidates. To the best of our knowledge, VulDeeLocator is the *first* deep learning-based detector that uses intermediate code-based vulnerability candidates to accommodate semantic information and uses *granularity refinement* to achieve a high locating precision.

Prior work on dynamic vulnerability detection. Dynamic vulnerability detection, including dynamic symbolic execution [44], [45] and fuzzing [46], [47], is complementary to static vulnerability detection and is often used to detect vulnerabilities in binary code. These methods explore program execution paths to identify the inputs that make the program exhibit unsafe operations (e.g., crashing). These methods can determine whether or not a program has a bug or vulnerability, but cannot precisely locate the vulnerability. In contrast, VulDeeLocator not only can detect vulnerabilities but also can pin down the locations of vulnerabilities, partly owing to the availability of program source code and intermediate code.

Prior work on bug detection. Since vulnerabilities can be seen as a special kind of bugs [48], we briefly review prior studies on bug detectors. Similar to vulnerability detection, there are two detection methods: static vs. dynamic. Static methods often use information retrieval techniques together with bug reports to detect bugs in source code (e.g., [49], [50]). Dynamic methods include: spectrum-based methods [51], which examine pass-and-fail execution traces to determine whether or not a line of source code has a bug; mutation-based methods [52], [53], which consider whether or not the execution of a line of code affects the result of a test case. However, bug detection methods cannot be used to detect vulnerabilities because (i) bugs are not necessarily vulnerabilities and (ii) bug detection methods often rely on bug reports or test cases.

9 Conclusion

We presented VulDeeLocator, the first deep learning-based fine-grained vulnerability detector that can simultaneously achieve a high detection capability and a high locating precision. It achieves these by leveraging intermediate code to capture semantic information that cannot be conveyed by source code-based representations and the new idea of granularity refinement. As one application, VulDeeLocator detected four vulnerabilities that were *not* reported in the NVD. The limitations of the present study offer interesting open problems for future research.

Acknowledgment

We thank the reviewers for their constructive comments, which have guided us in improving the paper.The authors from Huazhong University of Science and Technology are supported in part by the National Key Research and Development (R&D) Plan of China under Grant No. 2017YFB0802205, in part by the National Natural Science Foundation of China under Grant No. 61672249 and No. 61802106, in part by the Shenzhen Fundamental Research Program under Grant No. JCYJ20170413114215614. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the funding agencies in any sense.

References

[1] *CVE*, http://cve.mitre.org/.

[2] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: a scalable approach for vulnerable code clone discovery," in *Proceedings of 2017 IEEE Symposium on Security and Privacy, San Jose, CA, USA*, 2017, pp. 595–614.

[3] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: finding unpatched code clones in entire OS distributions," in *Proceedings of 2012 IEEE Symposium on Security and Privacy, San Francisco, California, USA*, 2012, pp. 48–62.

[4] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium*, 2010, pp. 447–456.

[5] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications, Los Angeles, CA, USA*, 2016, pp. 201–213.

[6] *Flawfinder*, http://www.dwheeler.com/flawfinder.

[7] *Checkmarx*, https://www.checkmarx.com/.

[8] *HP Fortify*, https://www.hpfod.com/.

[9] *Coverity*, https://scan.coverity.com/.

[10] H. Liang, L. Wang, D. Wu, and J. Xu, "MLSA: a static bugs analysis tool based on LLVM IR," in *Proceedings of the 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, Shanghai, China*, 2016, pp. 407–412.

[11] Z. Fang, Q. Liu, Y. Zhang, K. Wang, Z. Wang, and Q. Wu, "A static technique for detecting input validation vulnerabilities in android apps," *SCIENCE CHINA Information Sciences*, vol. 60, no. 5, pp. 052 111:1–052 111:16, 2017.

[12] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th Annual Computer Security Applications Conference, Orlando, FL, USA*, 2012, pp. 359–368.

[13] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of 2007 ACM Conference on Computer and Communications Security, Alexandria, Virginia, USA*, 2007, pp. 529–540.

[14] G. Grieco, G. L. Grinblat, L. C. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proceedings of the 6th ACM on Conference on Data and Application Security and Privacy, New Orleans, LA, USA*, 2016, pp. 85–96.

[15] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: exposing missing checks in source code for vulnerability discovery," in *Proceedings of 2013 ACM SIGSAC Conference on Computer and Communications Security, Berlin, Germany*, 2013, pp. 499–510.

[16] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *Proceedings of 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA*, 2015, pp. 797–812.

[17] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: a deep learning-based system for vulnerability detection," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium, San Diego, California, USA*, 2018.

[18] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: a framework for using deep learning to detect software vulnerabilities," *CoRR*, vol. abs/1807.06756, 2018.

[19] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "POSTER: vulnerability discovery with function representation learning from unlabeled projects," in *Proceedings of 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA*, 2017, pp. 2539–2541.

[20] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning," *CoRR*, vol. abs/1807.04320, 2018.

[21] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, M. W. McConley, J. M. Opper, S. P. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," *CoRR*, vol. abs/1803.04497, 2018.

[22] S. S. Muchnick, *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[23] M. Chalupa, "Slicing of LLVM bitcode," Master's thesis, Masaryk University, 2016.

[24] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "Leopard: identifying vulnerable code for vulnerability assessment through program metrics," in *Proceedings of the 41st International Conference on Software Engineering, Montreal, QC, Canada*, 2019, pp. 60–71.

[25] *NVD*, https://nvd.nist.gov/.

[26] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of 2014 IEEE Symposium on Security and Privacy, Berkeley, CA, USA*, 2014, pp. 590–604.

[27] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 3, 1995.

[28] C. Lattner and V. S. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization, San Jose, CA, USA*, 2004, pp. 75–88.

[29] *Clang*, http://clang.llvm.org/.

[30] *dg*, https://github.com/mchalupa/dg.

[31] *Source Level Debugging with LLVM*, http://llvm.org/docs/SourceLevelDebugging.html.

[32] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[33] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: encoder-decoder approaches," in *Proceedings of SSST@EMNLP 2014, 8th Workshop on Syntax, Semantics and Structure in Statistical Translation, Doha, Qatar*, 2014, pp. 103–111.

[34] M. Pendleton, R. Garcia-Lebron, J. Cho, and S. Xu, "A survey on systems security metrics," *ACM Comput. Surv.*, vol. 49, no. 4, pp. 62:1–62:35, 2017.

[35] *Intersection over Union*, https://en.wikipedia.org/wiki/Jaccard_index?tdsourcetag=s_pctim_aiomsg.

[36] *Software Assurance Reference Dataset*, https://samate.nist.gov/SRD/index.php.

[37] *Common Weakness Enumeration*, http://cwe.mitre.org/.

[38] *word2vec*, http://radimrehurek.com/gensim/models/word2vec.html.

[39] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: a system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, Savannah, GA, USA*, 2016, pp. 265–283.

[40] *Keras*, https://github.com/fchollet/keras.

[41] D. P. Kingma and J. Ba, "Adam: a method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[42] F. Yamaguchi, "Pattern-based vulnerability discovery," Ph.D. dissertation, University of Göttingen, 2015.

[43] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. Y. de Vel, and P. Montague, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Trans. Industrial Informatics*, vol. 14, no. 7, pp. 3289–3297, 2018.

[44] E. Alatawi, H. Søndergaard, and T. Miller, "Leveraging abstract interpretation for efficient dynamic symbolic execution," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, Urbana, IL, USA*, 2017, pp. 619–624.

[45] M. Christakis, P. Müller, and V. Wüstholz, "Guiding dynamic symbolic execution toward unverified program executions," in *Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA*, 2016, pp. 144–155.

[46] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: augmenting fuzzing through selective symbolic execution," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium, San Diego, California, USA*, 2016.

[47] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: path sensitive fuzzing," in *Proceedings of 2018 IEEE Symposium on Security and Privacy, San Francisco, California, USA*, 2018, pp. 679–696.

[48] X. Sun, X. Peng, K. Zhang, Y. Liu, and Y. Cai, "How security bugs are fixed and what can be improved: an empirical study with mozilla," *SCIENCE CHINA Information Sciences*, vol. 62, no. 1, pp. 19 102:1–19 102:3, 2019.

[49] Y. Xiao, J. Keung, Q. Mi, and K. E. Bennin, "Bug localization with semantic and structural features using convolutional neural network and cascade forest," in *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering, Christchurch, New Zealand*, 2018, pp. 101–111.

[50] F. Thung, T. B. Le, P. S. Kochhar, and D. Lo, "BugLocalizer: integrated tool support for bug localization," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China*, 2014, pp. 767–770.

[51] X. Xie, T. Y. Chen, F. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 4, pp. 31:1–31:40, 2013.

[52] M. Papadakis and Y. L. Traon, "Metallaxis-FL: mutation-based fault localization," *Software Testing, Verification & Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[53] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: mutating faulty programs for fault localization," in *Proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation, Cleveland, Ohio, USA*, 2014, pp. 153–162.