Which2learn: A Vulnerability Dataset Complexity Measurement Method for Data-Driven Detectors

Huozhu Wang
Institute of Information Engineering, Chinese Academy of Science School of Cyber Security, University of Chinese Academy Sciences
Beijing, China
wanghuozhu@iie.ac.cn

Ziyuan Zhu*
Institute of Information Engineering, Chinese Academy of Science School of Cyber Security, University of Chinese Academy Sciences
Beijing, China
zhuziyuan@iie.ac.cn

Dan Meng
Institute of Information Engineering, Chinese Academy of Science School of Cyber Security, University of Chinese Academy Sciences
Beijing, China
mengdan@iie.ac.cn

2023 IEEE Symposium on Computers and Communications (ISCC) | 979-8-3503-0048-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISCC58397.2023.10218000

***Abstract*—The increasing number of software vulnerabilities on complex programs has posed potential threats to cyberspace security. Recently, many data-driven methods have been proposed to detect such a large number of vulnerabilities. However, most of these data-driven detectors mainly focus on developing different models to improve the classification performance, ignoring the important research question that prior knowledge is learned from vulnerability datasets when training models.**

In this work, we propose a novel method to determine which dataset is relatively high-complexity for a data-driven detector to learn prior knowledge. Our method is called *Which2learn* for short. Our dataset complexity measure method is based on the sample's Program Dependence Graph. Experiments show that our dataset measurement method can improve the state-of-the-art GNN-based model's F1-score by about 9.5% in popular memory-related vulnerability detection. Moreover, our dataset measurement method can be easily extended to select training samples in most graph embedding machine learning tasks.

***Index Terms*—Cybersecurity, Software vulnerability detection, Data-driven method, Graph Neural Network.**

I. Introduction

Vulnerabilities detection is the critical activity in the arms race between attackers and defenders in cyberspace [1]. Attackers may develop an exploit [2] before defenders fix the undetected vulnerability. Traditional vulnerability detection methods can usually be divided into two categories: static detection methods and dynamic detection methods. Among them, static detection methods have been widely used and usually achieve better coverage in a limited time than dynamic detection methods. Traditional static detection methods (e.g., Cppcheck, Checkmarx, Clang Static Analyzer, Infer, Fortify, Coverity, Flawfinder, ITS4, RATS, CodeQL and SVF [3]) have shown their success in detecting memory corruption vulnerabilities of well-defined patterns.

However, traditional static vulnerability detection methods rely on security experts' prior knowledge to define all kinds of detection patterns for different types of vulnerabilities, which is a hands-on and resource-intensive process. To improve efficiency, many data-driven methods have been proposed to focus on learning the patterns of vulnerable code examples automatically [4] - [20]. These existing data-driven models usually perform well on synthetic vulnerability datasets [9]–[11]. Unfortunately, these models usually do not perform well in real-world complex vulnerability datasets [8], [17]. The reason is that the synthetic vulnerability datasets (e.g., Juliet Test Suite [9]) contain vulnerabilities that security experts added manually. Therefore, synthetic vulnerabilities are relatively easy to detect, as they are usually added in the form of syntactic code changes to a single statement (e.g., NULL pointer dereference in a simple variable). Specifically, Fig. 1 shows the vulnerability datasets categories in the previous data-driven detectors. The vulnerability datasets common selecting process in previous studies [4], [6], [7], [15], [16], [18], [20], [23] is shown as step ③ in Fig. 1. Although some previous studies get relatively high F1-score in some selected Open-Source Software (OSS) samples, these selected OSS samples' amount is often different, relatively small, and lack representativeness [25]. Moreover, Chakraborty et al. [21] found that if they directly use a previous pre-trained model to detect their created real-world vulnerabilities datasets (i.e., Chromium and Debian), the F1-score drops by about 73% on average. Even if they retrain these models with their datasets, the F1-score drops by about 54% from the reported results.

Challenges. Therefore, the state-of-art vulnerability datasets in data-driven detectors are suffering from the following challenges: (1) In a data-driven vulnerability detector training task, the gap between synthetic and real-world vulnerability datasets is unclear in current studies. The gap can cause the model's F1-score to decrease significantly. (2) The gap measurement method is lacking in the current studies. (3) The method to improve a detector model's F1-score by reducing the gap is also lacking.

To alleviate these challenges, we first systematically study existing vulnerability datasets in the current literature. And find the main gap between synthetic and real-world datasets is the sample complexity (the detail about the gap can be seen in Section II). Secondly, to accurately measure a sample

*Corresponding author.

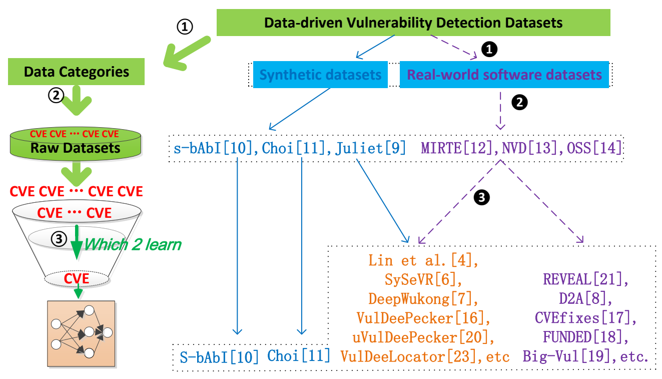


Fig. 1. Popular publicly available datasets in data-driven vulnerability detection. Depending on how these code samples are generated, we classify them into two categories: (1) synthetic vulnerability datasets; (2) real-world software vulnerability datasets.

complexity, we propose a novel quantitative method based on Program Dependence Graph (PDG) at the LLVM IR layer which is more fine-grained than the context captured from the source code in the previous studies [4], [6], [7], [15], [16], [18], [20].

Our Contributions. In summary, the main contributions of our work are as follows:

(1) We systematically study existing vulnerability datasets in the current literature. And find the main gap between synthetic and real-world datasets is the sample complexity.

(2) We propose new quantitative and fine-grained evaluation metrics (C_S metrics) to measure the Complexity of a sample based on its Program Dependence Graph (PDG) at the LLVM IR level.

(3) Based on our proposed C_S metrics, we implement our measure method called *Which2learn*. Then conduct a comprehensive measurement experiment in 2200 real-world CVE C/C++ cases and 64099 synthetic C/C++ cases. These cases generate 1133041 fine-grained SPDGs (Subgraphs of Program Dependence Graph). Experiments on these SPDGs show that our sample complexity measure method can improve the state-of-the-art GNN-based model's F1-score by about 9.5% in popular memory-related vulnerability detection. To the best of our knowledge, our method is the first work to measure a vulnerable sample's complexity. Moreover, our sample complexity measurement method can be easily extended to select training samples in most graph embedding machine learning tasks.

Paper Organization. The rest of the paper is organized as follows: Section II discusses key issues and challenges in current vulnerability datasets. Section III presents our proposed new metrics to measure a vulnerability dataset accurately. Section IV elaborates on our datasets and experimental setting details. We present detailed results on the evaluations of our proposed approach in section V. The related work is discussed in section VI. The conclusion is presented in section VII.

II. Background And Motivation

In this section, we systematically study vulnerability datasets in the current literature [8] - [21]. Then, we reveal several key issues and challenges when using these vulnerability datasets.

A. Existing datasets

Fig. 1 shows popular publicly available vulnerability datasets in data-driven vulnerability detectors. Juliet Test Suite [9], S-bAbI [10], and Choi et. al [11] are synthetic datasets that were generated from predefined vulnerable patterns. Synthetic datasets were originally designed for evaluating traditional static and dynamic analysis-based vulnerability detection tools. Compared to real-world software vulnerability datasets (like Listing 2), the main limitation of synthetic datasets (like Listing 1) is the lack of sample complexity. A relatively complex sample may introduce a vulnerability when programmers miss a corner case [26].

```
// CWE467_Use_of_sizeof_on_Pointer_Type__struct_44.c
-  data = (twoints *)malloc(sizeof(data));
+  data = (twoints *)malloc(sizeof(*data));
   data->a = 1;
   func_ptr(data);
```

Listing 1. A code snippet of a synthetic vulnerability sample in Juliet [9].

B. Motivation Example

```
1    // linux/include/linux/usb/composite.h
2       struct usb_configuration {
3          /*...*/
4          struct usb_function *interface[MAX_CONFIG_INTERFACES];
5       };
6
7    // linux/drivers/usb/gadget/composite.c
8       /*...*/
9       struct usb_configuration *os_desc_cfg;
10      os_desc_cfg = cdev->os_desc_config;
11      /*...*/
12      interface = w_value & 0xFF;
13      /* POTENTIAL FLAW: The retrieved function pointer may be null. */
14   +  if (interface >= MAX_CONFIG_INTERFACES ||
15   +     !os_desc_cfg->interface[interface])
16   +     break;
17      buf[6] = w_index;
18      count = count_ext_prop(os_desc_cfg, interface);
19
```

Listing 2. The patch for CVE-2022-25258 [22] in the Linux kernel.

Consider a function-pointer example in Listing 2, the function-pointer ($usb\_function = os\_desc\_cfg \rightarrow interface[interface]$) may be NULL when dereferencing the structure pointer os_desc_cfg at line 15. Although the fix (from line 14 to line 16) is very simple, detecting the vulnerability itself requires in-depth reasoning about the complex context of deeply nested code. A model trained with relatively simple examples like Listing 1 in synthetic datasets (e.g., [9]–[11]) may fail to detect complex vulnerabilities like Listing 2 in real-world software. We describe the cause with two aspects: (a) the complexity of sample statements; (b) the complexity of dependency edges between statements.

(a) Complexity of sample statements.

We describe the complexity of sample statements using a real-world vulnerability. As shown in the top part of Listing 3, most data-driven source code-based detectors consider two different types of the same name variable (e.g., $interface$) as the same symbolic representation (e.g., *VAR2*) in the word embedding process. But variable $interface$ in red is a function pointer array, and variable $interface$ in green is an array index. So the variable type information is lost after embedding the source code. As a result, the detector can not determine whether the symbol *VAR2* is an array or index. In our experiment, we find no previous data-driven vulnerability detector could detect this vulnerable sample.

(b) Complexity of dependency edges between statements. The number of control-dependency and/or data-dependency edges in real-world samples is much larger than in synthetic samples. For example, the number of dependency edges in LLVM IR graph generated from a real-world sample as Listing 2 is 26317 (the statistical tool is SVF [3]), but the number of dependency edges in LLVM IR graph generated from a synthetic sample as Listing 1 is 12. When a GNN-based model is trained with synthetic samples, then test with real-world samples (as Fig. 3 shows), the GNN-based model may fail to update nodes' information along a long path consisting of adjacency edges.

```
1   /* The patch code snippet of CVE-2022-25258 in the Linux kernel. */
2   os_desc_cfg->interface[interface]
3
4   /* Source code symbolic representations in embedding preprocess. */
5   VAR1->VAR2[VAR2]
6
7   /* LLVM IR. */
8   %360=getelementptr %struct.usb_configuration, %struct.usb_configuration*
        %330, i64 0, i32 14, i64 %359
9   %361=load %struct.usb_function*, %struct.usb_function** %360, align 8
10
11  /* LLVM IR symbolic representations in embedding preprocess. */
12  %INT1=getelementptr %struct.VAR1, %struct.VAR1* %INT2, i64 INT3, i32 INT4,
        i64 %INT5
13  %INT6=load %struct.VAR2*, %struct.VAR2** %INT1
14
```

Listing 3. Comparsion of different code embedding in CVE-2022-25258.

While the above show two examples of function pointers, there are many complex samples with different programming idioms in real-world vulnerability datasets, such as callback functions, virtual functions, and dynamic linking [40]. Here, we can observe that sample complexity is the main gap between synthetic and real-world samples. Then, we introduce our proposed novel quantitative method to measure sample complexity in section III.

III. Proposed Method

To tackle the above challenge mentioned in Section 2, we propose a novel method to measure the complexity of a sample at the LLVM IR level (Low Level Virtual Machine Intermediate Representation). Compared with source code-based embedding, LLVM IR mainly has the following two advantages in the vulnerability dataset measurement task:

(1) LLVM IR can provide more variable-type information. The variable-type information can provide more context information when embedding. For example, as shown in line 2 of Listing 3, variable $interface$ in red is a function pointer array, and variable $interface$ in green is an array index. Line 8 shows the corresponding LLVM IR, the integer 14 in red means that variable $interface$ is the fifteenth member of struct *usb_configuration*, and the integer 359 in green indicates the array index $interface$ in green. So we can use LLVM IR to distinguish the same name variable in the source code. In other words, if we feed LLVM IR into the model, the model will learn more prior knowledge from LLVM IR symbolic representations than the source code symbolic representations.

(2) As subsection II-B mentioned, LLVM IR can show a more fine-grained control-dependency and/or data-dependency relation. The benefit is the measurement of the complexity of dependency relations will be more accurate.

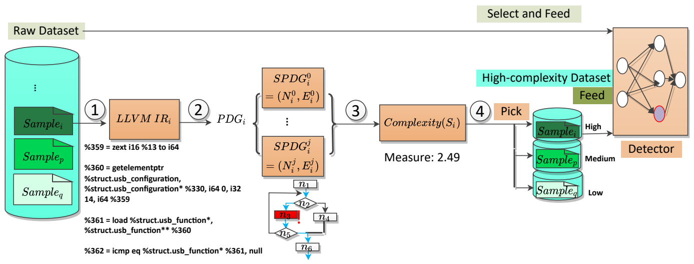

Select and Feed
Raw Dataset
$Sample_i$
$Sample_p$
$Sample_q$
1
$LLVM\ IR_i$
2
PDG_i
$SPDG_i^0 = (N_i^0, E_i^0)$
$SPDG_i^j = (N_i^j, E_i^j)$
3
$Complexity(S_i)$
Measure: 2.49
4
Pick
High-complexity Dataset
Feed
High
Medium
Low
Detector
%359 = zext i16 %13 to i64
%360 = getelementptr %struct.usb_configuration, %struct.usb_configuration* %330, i64 0, i32 14, i64 %359
%361 = load %struct.usb_function*, %struct.usb_function** %360
%362 = icmp eq %struct.usb_function* %361, null
n_1
n_2
n_3
n_4
n_5
n_6

Fig. 2. Overview of *Which2learn*. The $SPDG_i^j$ means the j-th Subgraph of $sample_i$ Program Dependence Graph.

With respect to the data-driven vulnerability detection task, our novel dataset measure methods at the LLVM IR level can be used for dataset selection. Our proposed method can help researchers determine which dataset is relatively high-complexity for a data-driven detector to learn prior knowledge. Our method is called *Which2learn* for short.

A. Overview of Which2learn

Fig. 2 highlights the overview of *Which2learn*. Let's first introduce the input and output of our method. The input of our proposed method is the raw dataset for data-driven vulnerability detectors. The raw dataset is a collection of source code samples, which may be vulnerable or non-vulnerable.

The output of our proposed method is a relatively high-complexity dataset. A relatively high-complexity dataset can be used to train a robust detector for real-world complex vulnerabilities. Next, we will show our method from input to output.

B. Detail Pipeline of Which2learn

We use the following algorithm and definitions to make our method description precise.

(a) Definition and Algorithm

Definition 1 (Vulnerability Dataset). To train a vulnerability detection model, we need a dataset defined as a set D:

$$D = \{S_i \mid CWEID_{S_i} \in (\{0\} \cup CWE_{1000}), i \in \{1, ..., |D|\}\} \quad (1)$$

where $|D|$ is the number of samples in the dataset D. And S_i is a non-vulnerable or vulnerable sample. Similar to previous works [16], the CWEID (Common Weakness Enumeration Identifier) of S_i is an integer in the CWE_{1000} view [24].

Definition 2 (Dataset Complexity). To alleviate the challenge mentioned in Section I, we define a new metric called $Complexity(D)$ denotes the complexity of a dataset with respect to the vulnerability detection task. As shown in Eq. 2, given a vulnerability dataset, we define the dataset complexity $Complexity(D)$ is the arithmetic average of each sample complexity C_{S_i}.

$$Complexity(D) = \frac{1}{|D|}\sum_{i=1}^{|D|} C_{S_i} \quad (2)$$

where $|D|$ is the number of samples in dataset D. Then we introduce detailed steps of how we measure a sample's complexity with algorithm 1.

(b) Measuring a sample complexity (C_{S_i}).

As shown in Fig. 2, the proposed method to measure a sample complexity can be divided into three steps: step ①, step ② and step ③.

Algorithm 1: Sample complexity $Complexity(S_i)$

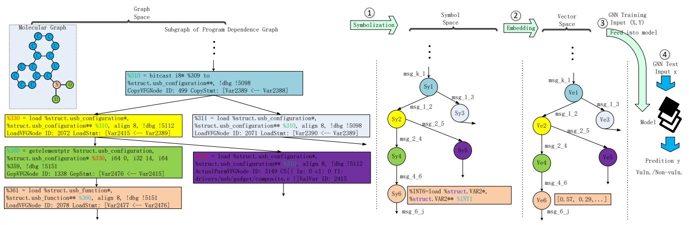
```
Input  : $S_i$ (The i-th sample in Dataset $D$)
Output: $C_{S_i}$ ( Complexity of $S_i$ )
1 $IR_i \leftarrow S_i$ /* Step ① in Fig. 2 */
2 $PDG_i \leftarrow IR_i$ /* Step ② in Fig. 2 */
3 $C_{S_i} \leftarrow 0$ /* Line: 3 -> 8: Step ③ in Fig. 2 */
4 while $SPDG_i^j$ in $PDG_i$ do
5     $C_{N_i^j} \leftarrow NodesComplexity(SPDG_i^j)$ /* Eq. 4 */
6     $C_{E_i^j} \leftarrow EdgesComplexity(SPDG_i^j)$ /* Eq. 5 */
7     $C_{SPDG_i^j} \leftarrow C_{N_i^j} * C_{E_i^j}$
8     $C_{S_i} \leftarrow Max(C_{S_i}, C_{SPDG_i^j})$
```

Step ① in Fig. 2: Generating LLVM IR_i.

We compile a sample S_i to convert source code to LLVM IR IR_i with Clang.

Step ② in Fig. 2: Generating $SPDG_i^j$.

To capture the vulnerable context in sample S_i, we need to generate Program Dependence Graphs PDG_i [27] from IR_i. For getting a more fine-grained vulnerable context, we partition the large full Program Dependence Graph PDG_i into many smaller subgraphs. We define the j-th subgraph as $SPDG_i^j$. For example, Fig. 3 shows a part of $SPDG_i^j$ generated from the example in Listing 3.



```
%310 = bitcast i8* %309 to
%struct.usb_configuration**, !dbg !5098
CopyVFGNode ID: 499 CopyStmt: [Var2389 <-- Var2388]

%330 = load %struct.usb_configuration*,
%struct.usb_configuration** %310, align 8, !dbg !5112
LoadVFGNode ID: 2072 LoadStmt: [Var2415 <-- Var2389]

%311 = load %struct.usb_configuration*,
%struct.usb_configuration** %310, align 8, !dbg !5098
LoadVFGNode ID: 2071 LoadStmt: [Var2390 <-- Var2389]

%360 = getelementptr %struct.usb_configuration,
%struct.usb_configuration* %330, i64 0, i32 14, i64
%359, !dbg !5151
GepVFGNode ID: 1338 GepStmt: [Var2476 <-- Var2415]

%330 = load %struct.usb_configuration*,
%struct.usb_configuration** %310, align 8, !dbg !5112
ActualParmVFGNode ID: 3149 CS[{ ln: 0 cl: 0 fl:
drivers/usb/gadget/composite.c }]ValVar ID: 2415

%361 = load %struct.usb_function*,
%struct.usb_function** %360, align 8, !dbg !5151
LoadVFGNode ID: 2078 LoadStmt: [Var2477 <-- Var2476]

%INT6=load %struct.VAR2*,
%struct.VAR2** %INT1
```

Fig. 3. For space reasons, the left shows a part of $SPDG$ (Subgraph of Program Dependence Graph) generated for the example from listing 3 in CVE-2022-25258. The right shows the message-passing process in a neural network. The msg_i_j means the message propagated from $node_i$ to $node_j$ in a neural network.

Step ③ in Fig. 2: $SPDG_i^j$ Complexity Measurement.

To measure $SPDG_i^j$ complexity, we divide $SPDG_i^j$ complexity into two parts: the complexity of nodes features and the complexity of edges connections. Because the two parts are two orthogonal aspects of a graph $G = (N, E)$, we consider the product of the two parts' complexity as the $SPDG_i^j$ complexity shown in Eq. 3.

$$C_{SPDG_i^j} = C_{N_i^j} \; * \; C_{E_i^j} \tag{3}$$

where $C_{SPDG_i^j}$ denotes the complexity of a graph $SPDG_i^j$, $C_{N_i^j}$ denotes the average complexity of all nodes features in the graph $SPDG_i^j$, $C_{E_i^j}$ denotes the average complexity of all edges connections in the graph $SPDG_i^j$. Like the space complexity of an algorithm, we consider the maximum value of $SPDG_i^j$ complexity as a sample's complexity.

(c) Nodes features complexity (*NodesComplexity()*).

Given a graph $SPDG_i^j$, we define nodes features complexity $C_{N_i^j}$ is the arithmetic average of each nodes complexity C_{n_k}. So $C_{N_i^j}$ can be measured by Eq. 4.

$$C_{N_i^j} = \frac{1}{|N_i^j|} \sum_{k=1}^{|N_i^j|} C_{n_k} \tag{4}$$

where $|N_i^j|$ denotes the total amount of nodes in $SPDG_i^j$. Most data-driven detectors usually embed source code statements space to vector space, but we embed LLVM IR space to vector space for a more fine-grained measurement.

Since the LLVM instructions are RISC-like three-address instructions, each instruction is expected to define a destination operand. We treat the complexity of a node feature as the number of source operands. For example, as shown in Listing 3, the *getelementptr* instruction has 4 source operands (i.e., $\%330, 0, 14,$ and $\%359$), we maintain the complexity of the *getelementptr* node as 4. As another example in Listing 3, the *load* instruction has 2 source operands (i.e., $\%360$ and 8), we maintain the complexity of the *load* node as 2.

(d) Edges connections complexity (*EdgesComplexity()*).

In the classical spectral graph theory [29], the l_2 norm of eigenvalues is the sum of the squares of the difference between adjacent nodes in a graph. The larger the l_2 norm means the more complex the edges difference in a graph. So we consider the eigenvalues' l_2 norm as $C_{E_i^j}$ shown in Eq. 5.

$$C_{E_i^j} = \|\overrightarrow{\lambda_{L(SPDG_i^j)}}\|_2 \tag{5}$$

where $L(SPDG_i^j)$ is Laplacian matrix of graph $SPDG_i^j$.

IV. Experimental Setups

In this section, we describe experimental setups including evaluation datasets, evaluation metrics, and parameter settings.

A. Dataset

To analyze the impact of the dataset's complexity on data-driven models, we design our experiments on two kinds of datasets: (a) synthetic vulnerability datasets and (b) real-world vulnerability datasets.

(a) **Synthetic vulnerability dataset.** Software Assurance Reference Dataset (SARD) [9] is the most popular synthetic vulnerability dataset, which contains a large number of known vulnerability types. It is widely used to evaluate the performance of data-driven detectors in the previous study, e.g., [4], [6], [7], [15], [16], [18], [20]. We use the latest version (version 1.3) [9], which contains 64,099 C/C++ test cases.

(b) **Real-world vulnerability datasets.** Previous work (i.e., CVEfixes [17]) provided the largest raw open-source real-world vulnerability datasets up to 9 June 2021. We add the latest CVE cases to the dataset. Then we get a real-world dataset of 2200 real-world CVE cases from 361 C/C++ projects. Without loss of generality, we evaluate various open-source projects, including system software (e.g., Linux kernel) and user applications (e.g., FFmpeg). We adopt the method proposed by Zhou et al. [30] to label $SPDGs$ in samples in these Open-Source Software datasets.

In total, the statistics SPDGs of the above vulnerable cases in our evaluation dataset are shown in Table I. Like Lipp et al [31], we divide our datasets into eight abstract root groups [35]. Specifically, according to the CWE_{1000} tree view [24], root vulnerability type R-CWE-664 includes CWE-664 (Improper Control of a Resource Through its Lifetime), CWE-121 (Stack-based Buffer Overflow), CWE-122 (Heap-based Buffer Overflow) and other children vulnerability types. In each group, we utilize SMOTE [36] to alleviate the data imbalance problem during training.

TABLE I
THE STATISTICS OF OUR EVALUATION DATASETS. ROOT VULNERABILITY CLASS INCLUDES PILLAR [35] AND ITS CHILDREN CLASSES. OSS MEANS OPEN SOURCE SOFTWARE C/C++ PROJECT. # NON-VULNS MEANS THE TOTAL NUMBER OF NON-VULNERABLE $SPDGs$. # VULNS MEANS THE TOTAL NUMBER OF VULNERABLE $SPDGs$.

Root class	OSS (Real-world)		SRAD (Synthetic)	
R-CWE-ID	**# Non-vulns.**	**# Vulns.**	**# Non-vulns.**	**# Vulns.**
R-CWE-284	12076	5391	2512	1473
R-CWE-664	119503	75209	351879	51557
R-CWE-682	56247	10608	72953	10072
R-CWE-691	12640	9247	2753	1923
R-CWE-693	1162	895	4226	1479
R-CWE-703	2096	1247	3501	2537
R-CWE-707	83965	19704	117030	10479
R-CWE-710	65379	3481	16902	2915
Total	353068	125782	571756	82435

B. Design and Implementation

We first calculate the above real-world datasets' complexity to find relatively complex vulnerable/non-vulnerable samples. Then analyze the impact of the dataset's complexity on data-driven detectors, we design two kinds of experiments as follows:

(a) **Feed complex samples to well-trained models from the previous studies.** To the best of our knowledge, state-of-the-art data-driven vulnerability detectors can usually be summarized into two categories: Sequence-based and Graph-based. As sequence-based detection models (including transformer-based), the state-of-the-art sequence-based data-driven detector is VulDeeLocator [23]. So we feed relatively complex samples into well-trained models from VulDeeLocator [23]. We denote this scenario as $Scenario\_0\ \_seq$. As graph-based detection models, the state-of-the-art graph-based data-driven detector is DeepWukong [7]. So we feed relatively complex samples into well-trained models from DeepWukong [7]. We denote this scenario as $Scenario\_0\_graph$. Similar to VulDeeLocator [23] and DeepWukong [7], we apply F1-score as the evaluation metric shown in Eq. 6.

$$F_1 = (\frac{2}{Precision + Recall})^{-1} with \begin{cases} Precision = \text{TP/(TP+FP)} \\ Recall = \text{TP/(TP+FN)} \end{cases} \quad (6)$$

where TP (True Positive) means the total number of vulnerable $SPDGs$ which is classified into vulnerable $SPDGs$. FP (False Positive) means the total number of non-vulnerable $SPDGs$ which is classified into vulnerable $SPDGs$. FN (False Negative) means the total number of vulnerable $SPDGs$ which is classified into non-vulnerable $SPDGs$.

TABLE II
EVALUATION SCENARIOS BETWEEN REAL-WORLD DATASET (OSS) AND SYNTHETIC DATASET (SARD).

Test / Training	Real-world dataset	Synthetic dataset
Real-world dataset	Scenario_1 (F_1^{RR})	Scenario_2 (F_1^{RS})
Synthetic dataset	Scenario_3 (F_1^{SR})	Scenario_4 (F_1^{SS})

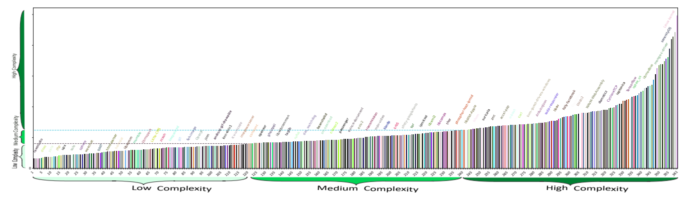


Fig. 4. The y-axis is $Complexity(D)$ that means Complexity of each open-source software vulnerable Dataset. The x-axis is the project index. All projects are taken into account. Every 5th package name appears on the top of the bar for space reasons.

(b) **Feed complex samples to train new models.** Inspired by Lipp et al. [31], we evaluate four different scenarios as shown in Table II These evaluation scenarios allow us to analyze the impact of dataset complexity on models. For a fair comparison, we adopt the same graph neural network (i.e, k-GNNs [34]) as the baseline (i.e., DeepWuKong [7]) for our LLVM instructions node embedding. And adopt the same parameter settings: $epoch$=50, batch size T=64, learning rate α=0.001, and the dimension of the vector representation of each node $vector\_dim$=128. Evaluation experiments of all models are conducted in the same environment: a 64-bit Linux system with Linux kernel 4.15.0 running on an Intel i7-7800X quad-core processor (3.50 GHz) with 16 GB RAM. We trained all the models in this evaluation on 1 Nvidia GTX 1080Ti GPU. Different from DeepWukong [7] adopt Joern [32] to generate *PDG* at the source code level, we adapt SVF [3] to generate *PDG* at LLVM IR level, which is more fine-grained. Moreover, we partition the large full Program Dependence Graph (PDG) into many smaller disconnected subgraphs ($SPDGs$) for a more fine-grained measurement.

V. RESULTS AND ANALYSIS

In this section, we report experimental results and the insight behind them. Firstly, we measure samples' complexity based on our proposed algorithm 1. Then we select high/low-complexity samples to test well-trained models and train new models to analyze the impact of dataset complexity on models.

A. Measurement complexity of datasets

As section IV mentioned, we implement our measure method *Which2learn*. Then we conduct a comprehensive experiment on 361 C/C++ open-source projects containing 125782 vulnerable (positive) $SPDGs$ and 353068 non-vulnerable (negative) $SPDGs$. In this experiment, we consider an open-source software project as a dataset containing positive and negative samples. Fig. 4 shows each dataset complexity $Complexity(D)$ measured by our proposed approach. Among these datasets, we find that the most complex dataset is the Linux kernel, which is the most right bar in Fig. 4. We also can find the least complex dataset is the Rawstudio project [33], which is the most left bar in Fig. 4. Similar to Common Vulnerability Scoring System (CVSS) [42], our proposed dataset's complexity is defined as three ratings (i.e., Low complexity, Medium complexity, and High complexity). Specifically, (1) The high-complexity datasets include operating system-level projects like the Linux kernel [22]. (2) The medium-complexity datasets include large-scale applications like krb5 [41], which is a network security protocol project. (3) The low-complexity datasets include small-scale user-level applications like Rawstudio [33], which is an image manipulation software. Fig. 5 (a) shows the complexity of three samples from the above three rating projects. As algorithm 1 mentioned, we consider the maximum value of $SPDG$ complexity as a sample's complexity rating. For example, as we can see a plus marker (+ [5, 261, 1563]) in Fig. 5 (a), the maximum value of $SPDG$ complexity of sample $composite.c$ is 1563, so we consider sample $composite.c$ is high-complexity. For another example, as we can see a triangle marker ([3, 218, 387]) in Fig. 5 (a), the maximum value of $SPDG$ complexity of sample *rs-filter.c* is 387, so we consider sample *rs-filter.c* is low-complexity.

B. Impact of dataset complexity on data-driven detectors

In this section, we analyze the impact of training dataset complexity on data-driven detectors. We implement three experiments in the following subsections (a), (b), and (c).

(a) **Training dataset size impact on data-driven detectors.**

We can see two phases as shown in Fig. 5 (b). In phase ①, the training dataset is low-complexity. The F1-score increases as the training dataset size increases. When we start adding

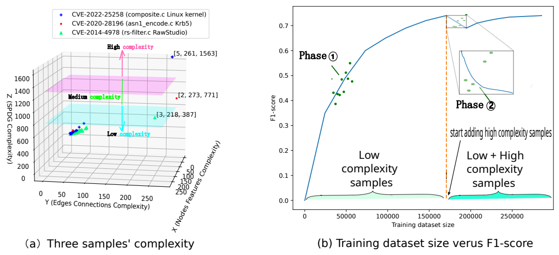


Fig. 5. (a) shows three samples' complexity. (b) shows the impact of dataset size on the model's F1-score.

high-complexity samples to the training dataset (in phase ②), the F1-score decreases for a short period. That's because the model learns little prior knowledge about the high-complexity samples in this short period. Then the model fails to detect these high-complexity samples. It's important to note that having more training data is not the only factor that affects the model's performance (i.e., F1-score). The quality of the training data is also important. The quality includes many aspects, such as sample complexity and label correctness. Based on the gap between the real-world and synthetic vulnerability samples mentioned in the previous sections, we mainly focus on studying the complexity of samples in the following experiment section.

(b) **Feed complex samples to well-trained models from the previous study.**

As we describe in Section IV, we feed our high-complexity real-world dataset (i.e., Open-Source Software) as the test data into two well-trained models (i.e., VulDeeLocator [23] and DeepWukong [7]). All hyperparameters are the same as that of the previous two studies [7], [23]. The test F1-scores are shown in Fig. 6. Comparing Scenario_0_seq and Scenario_0_graph, we can find that the graph-based detector is better than the sequence-based detector. That's because a graph-based detector can capture more context information in Program Dependence Graph (PDG) than a sequence-based detector. Moreover, we find three root vulnerability classes (R-CWE-{664, 703, 710}) can get a relatively high F1-score in Scenario_0_graph and Scenario_0_seq. That's because these three root vulnerability classes and their children's classes usually have a fixed vulnerability pattern. Specifically, R-CWE-664 means improper control of a resource through its lifetime, e.g., Out-of-bounds read/write. R-CWE-703 means an improper check or handling of exceptional conditions, e.g., unchecked return value. R-CWE-710 means improper adherence to coding standards, e.g., NULL pointer dereference.

(c) **Feed complex samples to train new models.**

As described in Section IV, we evaluate four different scenarios as shown in Table II. These evaluation scenarios allow us to analyze the impact of dataset complexity on models. As shown

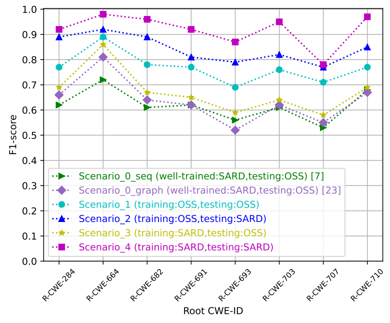


Fig. 6. The model's F1-score in different experimental scenarios. The reported F1-score of every experiment is the average repeated twenty times. Root vulnerability type (e.g., R-CWE-284) includes CWE-284 and its children's vulnerability type (e.g., CWE-273). Training means training data, and testing means test data. SARD is a widely used synthetic dataset, and OSS is a relatively complex real-world Open-Source Software dataset.

in Fig. 6, these four scenarios are Scenario_1, Scenario_2, Scenario_3, and Scenario_4. Among the four scenarios, Scenario_4 gets the highest F1-score because the training and test data are synthetic datasets (i.e., SARD). Training and test tasks are relatively simple to the model. However, Scenario_3 gets the lowest F1-score because the training dataset is a synthetic dataset (i.e., SARD), and the test dataset is a real-world dataset (i.e., OSS). That's because e model trained with relatively simple samples may fail to detect complex samples. Comparing Scenario_1 and Scenario_3, we find that a model trained with relatively complex datasets usually has a relatively high F1-score. Comparing Scenario_0_graph and Scenario_1, we can find that the F1-score in the popular memory-related vulnerability type (i.e., R-CWE-664 shown in axis-x) detection result increases by about 9.5% (from 81.6% to 89.4%) after training with our selected relatively complex datasets. In summary, our proposed method *Which2learn* can be used to select relatively complex training samples for training a better F1-score data-driven detector. Moreover, our dataset measurement method can be easily extended to select training samples in most graph embedding machine learning tasks. For example, our dataset measure method can be used to select relatively complex chemical molecular samples for drug discovery [43]. As Fig. 3 shows, we only need to replace the SPDGs (Subgraphs of Program Dependence Graph) with Molecular Graphes in the graph space. The other steps in our method do not need to change much. Then the molecular graph complexity would be divided into two parts (i.e., nodes features complexity and edges connections complexity).

VI. Related Work

There are several different complexities to describing a program's different features for various tasks. Cognitive Complexity [38] describes source code understandability for comprehension tasks. Cyclomatic Complexity [39] describes the structured testing methodology for software testing. Time Complexity and Space Complexity describe a program's dynamic behavior for algorithm analysis. However, the above complexity metrics are not designed for selecting vulnerable samples for training a vulnerability detector. Our proposed Sample Complexity can be applied in this task and even most graph embedding machine learning tasks.

VII. Conclusion

In this paper, we present *Which2learn*, a novel method to measure a vulnerability dataset's complexity. Then, we apply our measurement method to select relatively complex samples and feed these complex samples to state-of-the-art models. Experiments show that our dataset measurement method can improve the state-of-the-art GNN-based models' F1-score by about 9.5% in popular memory-related vulnerability detection. Moreover, our dataset measurement method can be easily extended to most graph embedding machine learning tasks.

References

[1] Lin, Z., Chen, Y., Xing, X., Li, K, "Your Trash Kernel Bug, My Precious 0-day," Blackhat Europe, pp. 38, 2021.
[2] Vulnerability Details (CVE-2017-0144), [Online]. Available: https://www.cvedetails.com/cve/CVE-2017-0144.
[3] Sui, Y., Xue, J., "SVF: interprocedural static value-flow analysis in LLVM," In CC, pp. 265–266, 2016.
[4] G. Lin, et al., "Software vulnerability discovery via learning knowledge bases," In IEEE TDSC, doi: 10.1109/TDSC.2019.2954088.
[5] Cheng, X., et al., "How About Bug-Triggering? Understanding and Characterizing Learning-Based Vulnerability Detectors," In IEEE TDSC, 2022.
[6] Li, Z., et al., "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities," In IEEE TDSC, pp. 1–1, 2021.
[7] Cheng, X., et al., "DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network," In TOSEM, 2021.
[8] Zheng, Y., et al., "D2A: A dataset built for AI-based vulnerability detection methods using differential analysis," In Proc. ICSE-SEIP, 2021.
[9] Juliet test suite for C/C++ version 1.3, [Online]. Available: https://samate.nist.gov/SARD/test-suites.
[10] Sestili, C., et al., "Towards security defect prediction with AI," CoRR, vol. abs/1808.09897, 2018.
[11] Choi, M., et al., "End-to-end prediction of buffer overruns from raw source code via neural memory networks. In: Proc. IJCAI, 2017.
[12] MITRE, Common vulnerabilities and exposures.[Online]. Available: https://cve.mitre.org/index.html.
[13] NIST, National vulnerability database. Accessed: May.19, 2022. [Online]. Available: https://nvd.nist.gov.
[14] Github, Github Open Source Softwares. Accessed: May.19, 2022. [Online]. Available: https://github.com.
[15] Russell, R., et al, "Automated vulnerability detection in source code using deep representation learning," In Proc. ICMLA, 2018.
[16] Li, Z., et al., "VulDeePecker: A deep learning-based system for vulnerability detection," In NDSS, 2018.
[17] Bhandari, G., et al., "CVEfixes: automated collection of vulnerabilities and their fixes from open-source software," In PROMISE, Athens Greece, pp. 30–39, 2021.
[18] Wang, H., et al., "Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection," In TIFS, 2021.
[19] Fan, J., et al., "A C/C++ code vulnerability dataset with code changes and CVE summaries," In Proc. MSR, pp. 508–512, 2020.
[20] Zou, D., et al., "μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection," In IEEE TDSC, vol. 18, no. 5, pp. 2224–2236, 2021.
[21] Chakraborty, S., et al., "Deep Learning based Vulnerability Detection: Are We There Yet?" In arXiv:2009.07235 [cs], 2020.
[22] CVE-2022-25258, [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-25258.
[23] Li, Z., et al., "VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector," In IEEE TDSC, pp. 1–17, 2021.
[24] CWE 1000 VIEW, https://cwe.mitre.org/data/definitions/1000.html. Last accessed 5 June 2022.
[25] K. N. Afanador, and C. E. Irvine, "Representativeness in the Benchmark for Vulnerability Analysis Tools (B-VAT)," In Proc. CSET, 2020.
[26] M.H. Halstead, "Elements of Software Science," Elsevier North-Holland, 1979.
[27] Ferrante, J., et al., "The program dependence graph and its use in optimization," In ACM Trans. Program. Languages Syst., vol. 9, no. 3, pp. 319–349, 1987.
[28] Branchaud-Charron, F., et al., "Spectral Metric for Dataset Complexity Assessment," In IEEE CVPR. doi:10.1109/CVPR.2019.00333.
[29] Y. Ma and J. Tang, "Graph deep learning," pp. 27-28, 2021.
[30] Zhou, Y., Sharma, A., "Automated identification of security issues from commit messages and bug reports," In: Proc. ESEC/FSE, 2017.
[31] Lipp, S., et al., "An empirical study on the effectiveness of static C code analyzers for vulnerability detection," In: Proc. ISSTA, pp. 544–555, 2022.
[32] Yamaguchi, F., et al., "Modeling and Discovering Vulnerabilities with Code Property Graphs," In: 2014 IEEE S&P, pp. 590–604, 2014.
[33] Rawstudio, https://github.com/rawstudio/rawstudio. Last accessed 9 Sep 2022.
[34] Christopher Morris, et al., "Higher-order graph neural networks," CoRR, abs/1810.02244, 2018.
[35] Pillar CWE, https://cwe.mitre.org/documents/glossary. Last accessed 9 Sep 2022.
[36] Nitesh V., et al., "SMOTE: Synthetic Minority over-Sampling Technique," In: Journal of Artificial Intelligence Research, pp.321–357, 2002.
[37] Sui, Y., et al., "Value-flow-based precise code embedding," In: Proc. ACM Program. Lang., vol. 4, no. OOPSLA, pp. 1–27, 2020.
[38] M. Muñoz Barón, et al., "An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability," in Proc. ESEM, pp. 1–12, Oct. 2020.
[39] D. R. Wallace, et al., "Structured testing:: a testing methodology using the cyclomatic complexity metric," NIST SP 500-235, 1996.
[40] X. Xu, et al. "CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software." USENIX Security Symposium. 2019.
[41] krb5, [Online]. Available: https://github.com/krb5/krb5.
[42] NVD Vulnerability Severity Ratings, [Online]. Available: https://nvd.nist.gov/vuln-metrics/cvss.
[43] Yuan, H., et al., "On Explainability of Graph Neural Networks via Subgraph Explorations," in ICML, pp. 12241–12252, 2021.
[44] Lipp, S., et al., "An empirical study on the effectiveness of static C code analyzers for vulnerability detection," In Proc. ISSTA, pp. 544–555, 2022.