THE UNIVERSITY OF TEXAS AT DALLAS

# DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection


Wenbo Wang
Department of Informatics
New Jersey Institute of Technology

Tien N. Nguyen
Computer Science Department
The University of Texas at Dallas

Shaohua Wang
Department of Informatics
New Jersey Institute of Technology

Yi Li
Department of Informatics
New Jersey Institute of Technology

Jiyuan Zhang
Computer Science Department
University of Illinois Urbana-Champaign

**Aashish Yadavally**
Computer Science Department
The University of Texas at Dallas

# Background

- Vulnerability detection is the task of analyzing a given code example to predict whether it is vulnerable (i.e., possesses vulnerabilities such as Denial of Service, Memory Corruption, etc.), or benign.

# Background

- Vulnerability detection is the task of analyzing a given code example to predict whether it is vulnerable (i.e., possesses vulnerabilities such as Denial of Service, Memory Corruption, etc.), or benign.
- Recent advances in machine and deep learning has prompted a surge in applying these techniques for automated vulnerability detection.

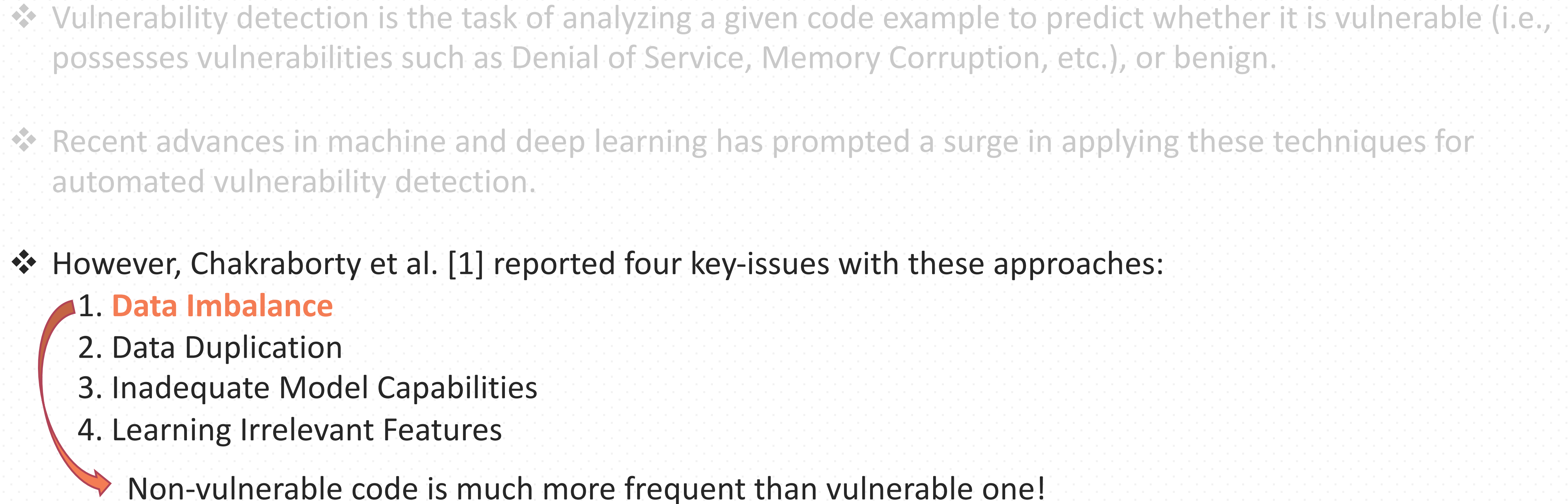
# Background

- Vulnerability detection is the task of analyzing a given code example to predict whether it is vulnerable (i.e., possesses vulnerabilities such as Denial of Service, Memory Corruption, etc.), or benign.

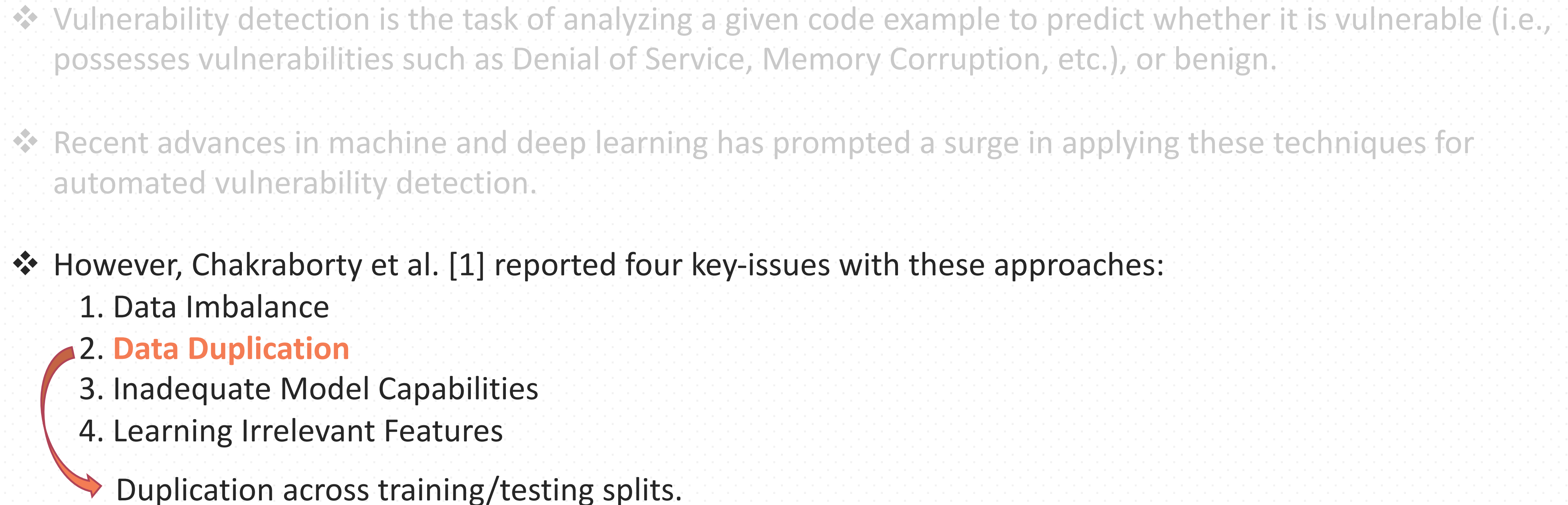
- Recent advances in machine and deep learning has prompted a surge in applying these techniques for automated vulnerability detection.

- However, Chakraborty et al. [1] reported four key-issues with these approaches:
  1. Data Imbalance
  2. Data Duplication
  3. Inadequate Model Capabilities
  4. Learning Irrelevant Features

# Background



# Background



# Background

- Vulnerability detection is the task of analyzing a given code example to predict whether it is vulnerable (i.e., possesses vulnerabilities such as Denial of Service, Memory Corruption, etc.), or benign.

- Recent advances in machine and deep learning has prompted a surge in applying these techniques for automated vulnerability detection.

- However, Chakraborty et al. [1] reported four key-issues with these approaches:
  1. Data Imbalance
  2. Data Duplication
  3. **Inadequate Model Capabilities**
  4. Learning Irrelevant Features

→ Treat code as sequence of tokens and DO NOT consider semantic dependencies..

# Background

- Vulnerability detection is the task of analyzing a given code example to predict whether it is vulnerable (i.e., possesses vulnerabilities such as Denial of Service, Memory Corruption, etc.), or benign.

- Recent advances in machine and deep learning has prompted a surge in applying these techniques for automated vulnerability detection.

- However, Chakraborty et al. [1] reported four key-issues with these approaches:
  1. Data Imbalance
  2. Data Duplication
  3. Inadequate Model Capabilities
  4. Learning Irrelevant Features

Training Process & Model Design

# Background

- Vulnerability detection is the task of analyzing a given code example to predict whether it is vulnerable (i.e., possesses vulnerabilities such as Denial of Service, Memory Corruption, etc.), or benign.

- Recent advances in machine and deep learning has prompted a surge in applying these techniques for automated vulnerability detection.

- However, Chakraborty et al. [1] reported four key-issues with these approaches:
  1. Data Imbalance
  2. Data Duplication
  3. Inadequate Model Capabilities
  4. **Learning Irrelevant Features**

Training Process & Model Design (issues 1–3)

We focus on identifying “*Class-Separation*” features

# Motivating Examples

**Figure 1.** CVE-2020-18899: Denial of Service (DoS) from an Uncontrolled Memory Allocation in Exiv2 0.27

```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file
6    if (subBox.length > io_->size() - io_->tell()) {
7      throw Error(kerCorruptedMetadata);
8    }
9    DataBuf data(subBox.length - sizeof(box));
10   io_->read(data.pData_,data.size_);
11 }
```

```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file || 0
6    if (subBox.length == 0 ||
7        subBox.length > io_->size() - io_->tell()) {
8      throw Error(kerCorruptedMetadata);
9    }
10   DataBuf data(subBox.length - sizeof(box));
11   io_->read(data.pData_,data.size_);
12 }
```

# Motivating Examples

**Figure 1.** CVE-2020-18899: Denial of Service (DoS) from an Uncontrolled Memory Allocation in Exiv2 0.27

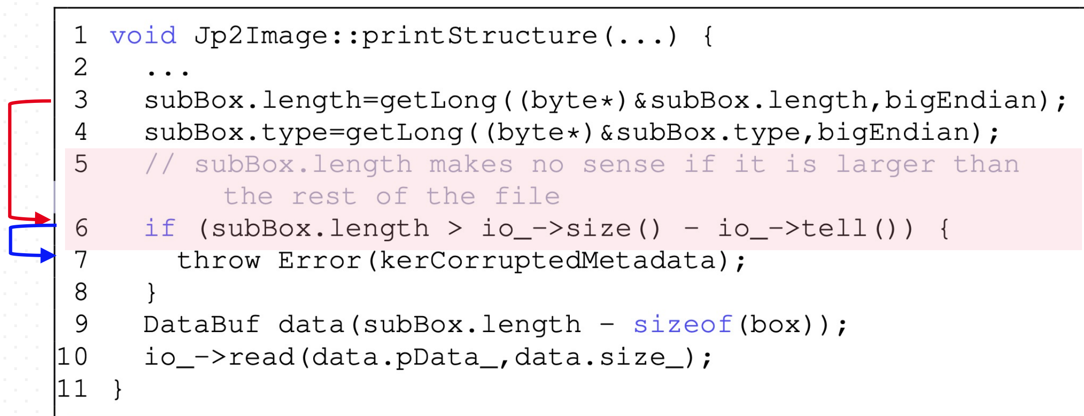
```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file
6    if (subBox.length > io_->size() - io_->tell()) {
7      throw Error(kerCorruptedMetadata);
8    }
9    DataBuf data(subBox.length - sizeof(box));
10   io_->read(data.pData_,data.size_);
11 }
```

```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file || 0
6    if (subBox.length == 0 ||
7        subBox.length > io_->size() - io_->tell()) {
8      throw Error(kerCorruptedMetadata);
9    }
10   DataBuf data(subBox.length - sizeof(box));
11   io_->read(data.pData_,data.size_);
12 }
```

Value of 0 for `subBox.length` results in Integer Overflow

# Motivating Examples

**Figure 1.** CVE-2020-18899: Denial of Service (DoS) from an Uncontrolled Memory Allocation in Exiv2 0.27



```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file
6    if (subBox.length > io_->size() - io_->tell()) {
7      throw Error(kerCorruptedMetadata);
8    }
9    DataBuf data(subBox.length - sizeof(box));
10   io_->read(data.pData_,data.size_);
11 }
```

```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file || 0
6    if (subBox.length == 0 ||
7        subBox.length > io_->size() - io_->tell()) {
8      throw Error(kerCorruptedMetadata);
9    }
10   DataBuf data(subBox.length - sizeof(box));
11   io_->read(data.pData_,data.size_);
12 }
```

We can observe a data dependency (**red**) from line 3 to line 6, and a control dependency (**blue**) from line 6 to line 7.

# Motivating Examples

**Figure 1.** CVE-2020-18899: Denial of Service (DoS) from an Uncontrolled Memory Allocation in Exiv2 0.27

```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file
6    if (subBox.length > io_->size() - io_->tell()) {
7      throw Error(kerCorruptedMetadata);
8    }
9    DataBuf data(subBox.length - sizeof(box));
10   io_->read(data.pData_,data.size_);
11 }
```

```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file || 0
6    if (subBox.length == 0 ||
7        subBox.length > io_->size() - io_->tell()) {
8      throw Error(kerCorruptedMetadata);
9    }
10   DataBuf data(subBox.length - sizeof(box));
11   io_->read(data.pData_,data.size_);
12 }
```

**Observation 1**

*A model could investigate the* data and control flows toward the exception/error-handling points to detect a potential vulnerability.

# Motivating Examples

**Figure 2.** CVE-2020-19155: Improper Access Control in `Jfinal`

```
1 public JSONObject rename() {
2   String oldFile = this.get.get("old");
3   String newFile = this.get.get("new");
4   oldFile = getFilePath(oldFile);
5   ...
6   String path = oldFile.substring(0, pos + 1);
7   File fileFrom = null;
8   File fileTo = null;
9   try {
10    fileFrom = new File(this.fileRoot + oldFile);
11    fileTo = new File(this.fileRoot + path + newFile);
12    if (fileTo.exists()) {
13     if (fileTo.isDirectory()) {
14      this.error(sprintf(lang("DIRECTORY_ALREADY_EXISTS");
15      error = true;
16     } else { // fileTo.isFile
17      this.error(sprintf(lang("FILE_ALREADY_EXISTS").));
18      error = true;
19     }
20    } else if (!fileFrom.renameTo(fileTo)) {
21     this.error(sprintf(lang("ERROR_RENAMING_DIRECTORY"));
22     error = true;
23    }
24  } catch (Exception e) {
25    if (fileFrom.isDirectory()) {
26     this.error(sprintf(lang("ERROR_RENAMING_DIRECTORY").;
27    } else {
28     this.error(sprintf(lang("ERROR_RENAMING_FILE"),..));
29    }
30    error = true;
31  }...
```

# Motivating Examples

**Figure 2.** CVE-2020-19155: Improper Access Control in `Jfinal`

```
1 public JSONObject rename() {
2   String oldFile = this.get.get("old");
3   String newFile = this.get.get("new");
4   oldFile = getFilePath(oldFile);
5   ...
6   String path = oldFile.substring(0, pos + 1);
7   File fileFrom = null;
8   File fileTo = null;
9   try {
10    fileFrom = new File(this.fileRoot + oldFile);
11    fileTo = new File(this.fileRoot + path + newFile);
12    if (fileTo.exists()) {
13     if (fileTo.isDirectory()) {
14      this.error(sprintf(lang("DIRECTORY_ALREADY_EXISTS");
15      error = true;
16     } else { // fileTo.isFile
17      this.error(sprintf(lang("FILE_ALREADY_EXISTS").));
18      error = true;
19     }
20    } else if (!fileFrom.renameTo(fileTo)) {
21     this.error(sprintf(lang("ERROR_RENAMING_DIRECTORY"));
22     error = true;
23    }
24  } catch (Exception e) {
25    if (fileFrom.isDirectory()) {
26     this.error(sprintf(lang("ERROR_RENAMING_DIRECTORY").;
27    } else {
28     this.error(sprintf(lang("ERROR_RENAMING_FILE"),..));
29    }
30    error = true;
31  }...
```

# Key Ideas

Focused on improving ***class-separability***, we consider the following:

- **Exception-Flow Graph (EFG)**, which helps distinguish the key characteristics in the proper/improper handling of exceptions and error cases, a key aspect of vulnerabilities.

# Key Ideas

Focused on improving ***class-separability***, we consider the following:

- **Exception-Flow Graph (EFG)**, which helps distinguish the key characteristics in the proper/improper handling of exceptions and error cases, a key aspect of vulnerabilities.

- **Post-Dominator Tree (PDT)** considers the regular flows, which EFG does not.

# Key Ideas

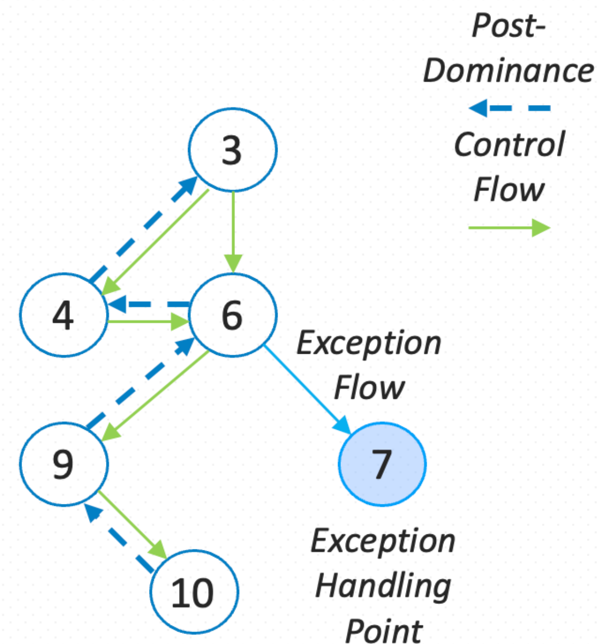
Focused on improving ***class-separability***, we consider the following:

- **Exception-Flow Graph (EFG)**, which helps distinguish the key characteristics in the proper/improper handling of exceptions and error cases, a key aspect of vulnerabilities.

- **Post-Dominator Tree (PDT)** considers the regular flows, which EFG does not.

**Figure 3.** Exception-Flow Graph (EFG) and Post-Dominator Tree (PDT) for vulnerable code example (left).

```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file
6    if (subBox.length > io_->size() - io_->tell()) {
7      throw Error(kerCorruptedMetadata);
8    }
9    DataBuf data(subBox.length - sizeof(box));
10   io_->read(data.pData_,data.size_);
11 }
```



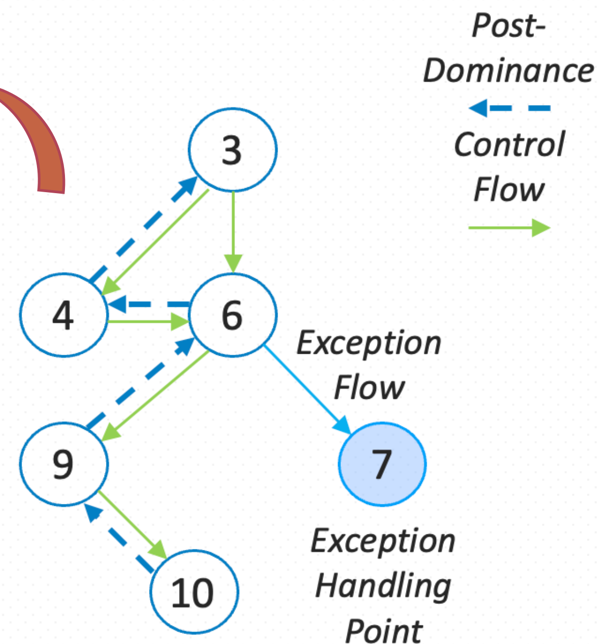
# Key Ideas

Focused on improving ***class-separability***, we consider the following:

- **Exception-Flow Graph (EFG)**, which helps distinguish the key characteristics in the proper/improper handling of exceptions and error cases, a key aspect of vulnerabilities.
- **Post-Dominator Tree (PDT)** considers the regular flows, which EFG does not.

**Figure 3.** Exception-Flow Graph (EFG) and Post-Dominator Tree (PDT) for vulnerable code example (left).

Statement ***d*** is considered as a *post-dominator* of another statement ***s*** if all the paths to the exit point of the method starting at ***s*** must go through ***d***.



# Key Ideas

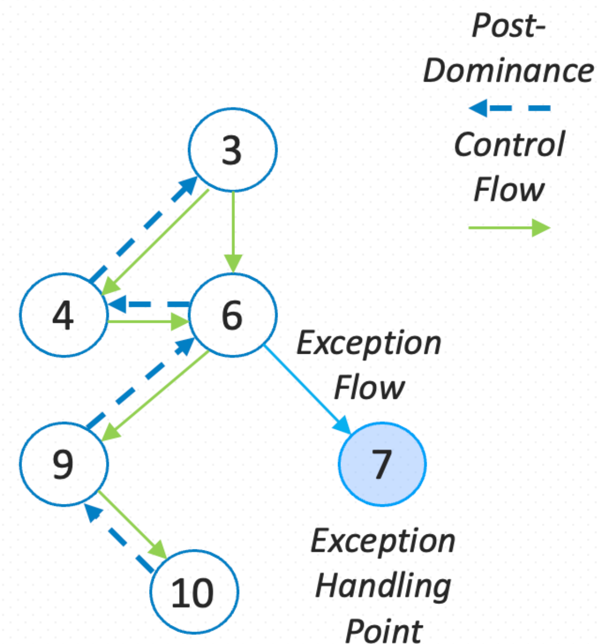
Focused on improving ***class-separability***, we consider the following:

- **Exception-Flow Graph (EFG)**, which helps distinguish the key characteristics in the proper/improper handling of exceptions and error cases, a key aspect of vulnerabilities.

- **Post-Dominator Tree (PDT)** considers the regular flows, which EFG does not.

**Figure 3.** Exception-Flow Graph (EFG) and Post-Dominator Tree (PDT) for vulnerable code example (left).

```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file
6    if (subBox.length > io_->size() - io_->tell()) {
7      throw Error(kerCorruptedMetadata);
8    }
9    DataBuf data(subBox.length - sizeof(box));
10   io_->read(data.pData_,data.size_);
11 }
```



# Key Ideas

Focused on improving ***class-separability***, we consider the following:

- **Exception-Flow Graph (EFG)**, which helps distinguish the key characteristics in the proper/improper handling of exceptions and error cases, a key aspect of vulnerabilities.

- **Post-Dominator Tree (PDT)** considers the regular flows, which EFG does not.

- With each node, we associate a **Statement Type** (i.e., the root of the sub-AST corresponding to the statement) which is analogous to the POS-tags in natural language.

# Key Ideas

Focused on improving ***class-separability***, we consider the following:

- **Exception-Flow Graph (EFG)**, which helps distinguish the key characteristics in the proper/improper handling of exceptions and error cases, a key aspect of vulnerabilities.
- **Post-Dominator Tree (PDT)** considers the regular flows, which EFG does not.
- With each node, we associate a **Statement Type** (i.e., the root of the sub-AST corresponding to the statement) which is analogous to the POS-tags in natural language.
- To capture the syntactic structure, we consider the **Long Path** between two leaf nodes.

# Key Ideas

Focused on improving ***class-separability***, we consider the following:

- **Exception-Flow Graph (EFG)**, which helps distinguish the key characteristics in the proper/improper handling of exceptions and error cases, a key aspect of vulnerabilities.

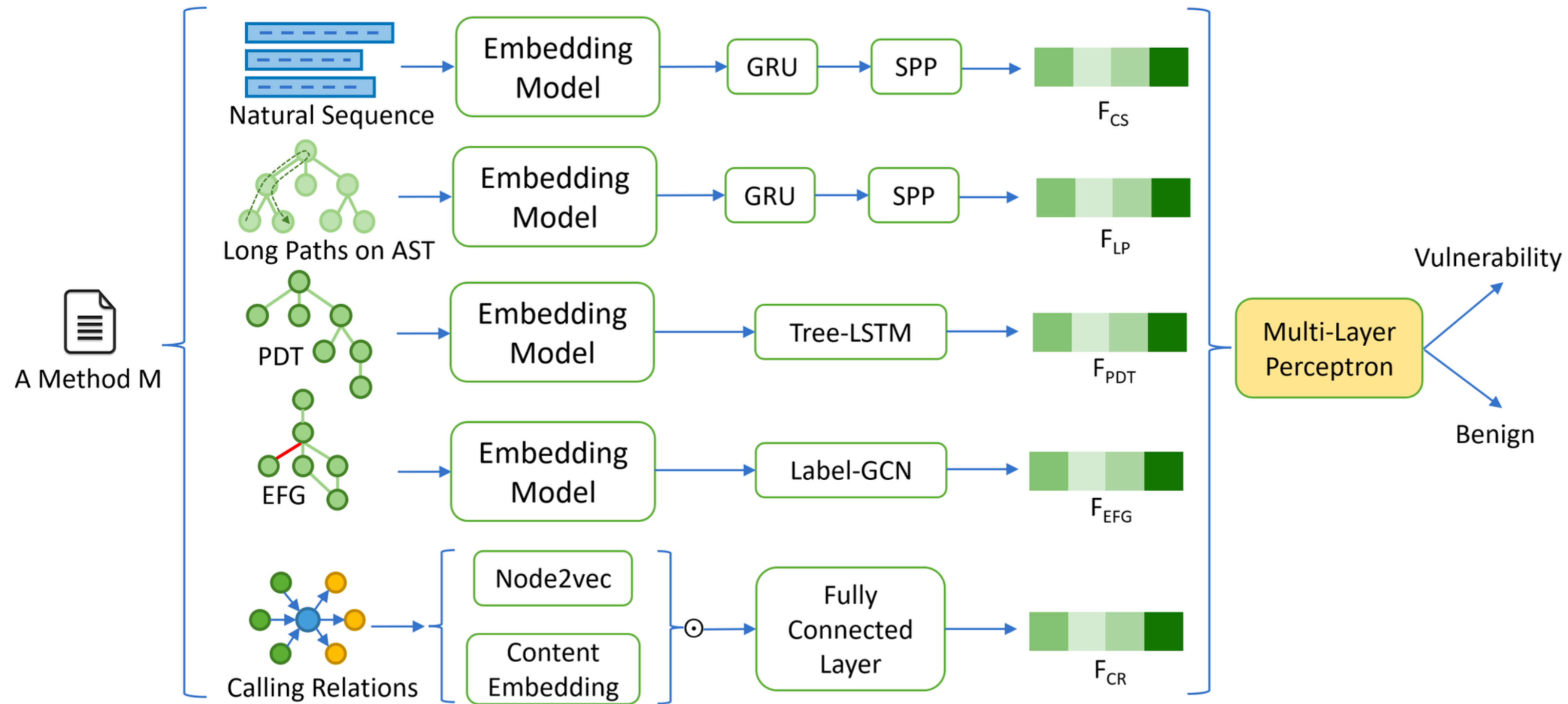
- **Post-Dominator Tree (PDT)** considers the regular flows, which EFG does not.

- With each node, we associate a **Statement Type** (i.e., the root of the sub-AST corresponding to the statement) which is analogous to the POS-tags in natural language.

- To capture the syntactic structure, we consider the **Long Path** between two leaf nodes.

- To capture the global context, we consider the caller/callee relations.

# Architecture Overview



# Empirical Evaluation

**Table 1:** Comparison with other DL-Based VD Approaches

| Approach | Precision | Recall | F-score |
|---|---|---|---|
| VulDeePecker | 0.55 | 0.77 | 0.64 |
| SySeVR | 0.54 | 0.74 | 0.63 |
| Russell *et al.* | 0.54 | 0.72 | 0.62 |
| Devign | 0.56 | 0.73 | 0.63 |
| Reveal | 0.62 | 0.69 | 0.65 |
| IVDetect | 0.54 | 0.77 | 0.67 |
| DeepVD | **0.70** | **0.89** | **0.78** |

# Empirical Evaluation

**Table 1:** Comparison with other DL-Based VD Approaches

| Approach | Precision | Recall | F-score |
|---|---|---|---|
| VulDeePecker | 0.55 | 0.77 | 0.64 |
| SySeVR | 0.54 | 0.74 | 0.63 |
| Russell *et al.* | 0.54 | 0.72 | 0.62 |
| Devign | 0.56 | 0.73 | 0.63 |
| Reveal | 0.62 | 0.69 | 0.65 |
| IVDetect | 0.54 | 0.77 | 0.67 |
| DEEPVD | **0.70** | **0.89** | **0.78** |

*Overall, DEEPVD* relatively improves over the baseline models from **13%– 29.6%** in Precision, from **15.6%–28.9%** in Recall, and from **16.4%–25.8%** in F-score.

**Figure 4.** CVE-2019-1563: A vulnerable code example in OpenSSL.

```
1  BIO *PKCS7_dataDecode(PKCS7 *p7, EVP_PKEY *pkey, BIO
       *in_bio, X509 *pcert) {
2    ...
3    if (evp_cipher != NULL) {
4      ...
5      if (pcert == NULL) {
6        for (i = 0; i < sk_PKCS7_RECIP_INFO_num(rsk); i++) {
7          ri = sk_PKCS7_RECIP_INFO_value(rsk, i);
8          if (pkcs7_decrypt_rinfo(&ek, &eklen, ri, pkey) < 0)
9            goto err;
10         ERR_clear_error();
11       }
12     } else {...}
13 }
```

# Empirical Evaluation

**Figure 4.** CVE-2019-1563: A vulnerable code example in OpenSSL.

```
1  BIO *PKCS7_dataDecode(PKCS7 *p7, EVP_PKEY *pkey, BIO
        *in_bio, X509 *pcert) {
2    ...
3    if (evp_cipher != NULL) {
4      ...
5      if (pcert == NULL) {
6        for (i = 0; i < sk_PKCS7_RECIP_INFO_num(rsk); i++) {
7          ri = sk_PKCS7_RECIP_INFO_value(rsk, i);
8          if (pkcs7_decrypt_rinfo(&ek, &eklen, ri, pkey) < 0)
9            goto err;
10         ERR_clear_error();
11       }
12     } else {...}
13 }
```

- Has 186 lines of code after removing comments and empty lines.
- PDG with 145 nodes and 477 edges, and the CPG with 622 nodes and 1,393 edges.
- In contrast, EFG + PDT has 145 nodes and 295 edges.

# Empirical Evaluation

**Table 2:** Comparison on different vulnerability types

| | Vulnerability Type | TN | FP | FN | TP | Total | **Precision** | **Recall** | **F-score** |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Denial Of Service | 424 | 490 | 64 | 658 | 1,636 | 0.57 | 0.91 | 0.70 |
| 2 | Overflow | 225 | 371 | 28 | 340 | 964 | 0.48 | 0.92 | 0.63 |
| 3 | Execute Code | 129 | 279 | 11 | 202 | 621 | 0.42 | 0.95 | 0.58 |
| 4 | Memory corruption | 102 | 190 | 9 | 162 | 463 | 0.46 | 0.95 | 0.62 |
| 5 | Obtain information | 63 | 45 | 7 | 76 | 191 | 0.63 | 0.92 | 0.75 |

# Empirical Evaluation

**Table 2:** Comparison on different vulnerability types

| | Vulnerability Type | TN | FP | FN | TP | Total | **Precision** | **Recall** | **F-score** |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Denial Of Service | 424 | 490 | 64 | 658 | 1,636 | 0.57 | 0.91 | 0.70 |
| 2 | Overflow | 225 | 371 | 28 | 340 | 964 | 0.48 | 0.92 | 0.63 |
| 3 | Execute Code | 129 | 279 | 11 | 202 | 621 | 0.42 | 0.95 | 0.58 |
| 4 | Memory corruption | 102 | 190 | 9 | 162 | 463 | 0.46 | 0.95 | 0.62 |
| 5 | Obtain information | 63 | 45 | 7 | 76 | 191 | 0.63 | 0.92 | 0.75 |

*Leveraging EFG+PDT particularly also helped with identifying the popular **DOS-based vulnerabilities**, that are majorly identified with improper exception/error-handling.*

# Conclusion

## Background

- Vulnerability detection is the task of analyzing a given code example to predict whether it is vulnerable (i.e., possesses vulnerabilities such as Denial of Service, Memory Corruption, etc.), or benign.
- Recent advances in machine and deep learning has prompted a surge in applying these techniques for automated vulnerability detection.
- However, Chakraborty et al. [1] reported four key-issues with these approaches:
  1. Data Imbalance
  2. Data Duplication
  3. Inadequate Model Capabilities
  4. **Learning Irrelevant Features**

Items 1–3: Training Process & Model Design

Item 4: We focus on identifying "*Class-Separation*" features

## nclusion

## Background

- Vulnerability detection is the task of analyzing a given code example to predict whether it is vulnerable (i.e., possesses vulnerabilities such as Denial of Service, Memory Corruption, etc.), or benign.
- Recent advances in machine and deep learning has prompted a surge in applying these techniques for automated vulnerability detection.
- However, Chakraborty et al. [1] reported four key-issues with these approaches:
  1. Data Imbalance
  2. Data Duplication
  3. Inadequate Model Capabilities
  4. **Learning Irrelevant Features**

(1–3) Training Process & Model Design

We focus on identifying "*Class-Separation*" features

## Motivating Examples

**Figure 1.** CVE-2020-18899: Denial of Service (DoS) from an Uncontrolled Memory Allocation in Exiv2 0.27

```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file
6    if (subBox.length > io_->size() - io_->tell()) {
7      throw Error(kerCorruptedMetadata);
8    }
9    DataBuf data(subBox.length - sizeof(box));
10   io_->read(data.pData_,data.size_);
11 }
```

```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file || 0
6    if (subBox.length == 0 ||
7        subBox.length > io_->size() - io_->tell()) {
8      throw Error(kerCorruptedMetadata);
9    }
10   DataBuf data(subBox.length - sizeof(box));
11   io_->read(data.pData_,data.size_);
12 }
```

**Observation 1**

*A model could investigate the* data and control flows toward the exception/error-handling points to detect a potential vulnerability.

## Background

- Vulnerability detection is the task of analyzing a given code example to predict whether it is vulnerable (i.e., possesses vulnerabilities such as Denial of Service, Memory Corruption, etc.), or benign.
- Recent advances in machine and deep learning has prompted a surge in applying these techniques for automated vulnerability detection.
- However, Chakraborty et al. [1] reported four key-issues with these approaches:
  1. Data Imbalance
  2. Data Duplication
  3. Inadequate Model Capabilities
  4. Learning Irrelevant Features

(Items 1–3: Training Process & Model Design)

We focus on identifying "*Class-Separation*" features

## Motivating Examples

**Figure 1.** CVE-2020-18899: Denial of Service (DoS) from an Uncontrolled Memory Allocation in Exiv2 0.27

```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file
6    if (subBox.length > io_->size() - io_->tell()) {
7      throw Error(kerCorruptedMetadata);
8    }
9    DataBuf data(subBox.length - sizeof(box));
10   io_->read(data.pData_,data.size_);
11 }
```

```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file || 0
6    if (subBox.length == 0 ||
7        subBox.length > io_->size() - io_->tell()) {
8      throw Error(kerCorruptedMetadata);
9    }
10   DataBuf data(subBox.length - sizeof(box));
11   io_->read(data.pData_,data.size_);
12 }
```

**Observation 1**

*A model could investigate the* data and control flows toward the exception/error-handling points to detect a potential vulnerability.

## Key Ideas

Focused on improving ***class-separability***, we consider the following:

- **Exception-Flow Graph (EFG)**, which helps distinguish the key characteristics in the proper/improper handling of exceptions and error cases, a key aspect of vulnerabilities.
- While EFG accommodates the exception-flows, **Post-Dominator Tree (PDT)** considers the regular flows.
- With each node, we associate a **Statement Type** (i.e., the root of the sub-AST corresponding to the statement) which is analogous to the POS-tags in natural language.
- To capture the syntactic structure, we consider the **Long Path** between two leaf nodes.
- To capture the global context, we consider the caller/callee relations.

## Background

❖ Vulnerability detection is the task of analyzing a given code example to predict whether it is vulnerable (i.e., possesses vulnerabilities such as Denial of Service, Memory Corruption, etc.), or benign.

❖ Recent advances in machine and deep learning has prompted a surge in applying these techniques for automated vulnerability detection.

❖ However, Chakraborty et al. [1] reported four key-issues with these approaches:

1. Data Imbalance
2. Data Duplication
3. Inadequate Model Capabilities
4. Learning Irrelevant Features

(1–3: Training Process & Model Design)

We focus on identifying "*Class-Separation*" features

## Key Ideas

Focused on improving ***class-separability***, we consider the following:

❖ **Exception-Flow Graph (EFG)**, which helps distinguish the key characteristics in the proper/improper handling of exceptions and error cases, a key aspect of vulnerabilities.

❖ While EFG accommodates the exception-flows, **Post-Dominator Tree (PDT)** considers the regular flows.

❖ With each node, we associate a **Statement Type** (i.e., the root of the sub-AST corresponding to the statement) which is analogous to the POS-tags in natural language.

❖ To capture the syntactic structure, we consider the **Long Path** between two leaf nodes.

❖ To capture the global context, we consider the caller/callee relations.

## Motivating Examples

**Figure 1.** CVE-2020-18899: Denial of Service (DoS) from an Uncontrolled Memory Allocation in Exiv2 0.27

```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file
6    if (subBox.length > io_->size() - io_->tell()) {
7      throw Error(kerCorruptedMetadata);
8    }
9    DataBuf data(subBox.length - sizeof(box));
10   io_->read(data.pData_,data.size_);
11 }
```

```
1  void Jp2Image::printStructure(...) {
2    ...
3    subBox.length=getLong((byte*)&subBox.length,bigEndian);
4    subBox.type=getLong((byte*)&subBox.type,bigEndian);
5    // subBox.length makes no sense if it is larger than
         the rest of the file || 0
6    if (subBox.length == 0 ||
7        subBox.length > io_->size() - io_->tell()) {
8      throw Error(kerCorruptedMetadata);
9    }
10   DataBuf data(subBox.length - sizeof(box));
11   io_->read(data.pData_,data.size_);
12 }
```

**Observation 1**

*A model could investigate the* data and control flows toward the exception/error-handling points to detect a potential vulnerability.

## Empirical Evaluation

**Table 1:** Comparison with other DL-Based VD Approaches

| Approach | Precision | Recall | F-sc [No Title] |
|---|---|---|---|
| VulDeePecker | 0.55 | 0.77 | 0.64 |
| SySeVR | 0.54 | 0.74 | 0.63 |
| Russell *et al.* | 0.54 | 0.72 | 0.62 |
| Devign | 0.56 | 0.73 | 0.63 |
| Reveal | 0.62 | 0.69 | 0.65 |
| IVDetect | 0.54 | 0.77 | 0.67 |
| DEEPVD | **0.70** | **0.89** | **0.78** |

*Overall, DEEPVD* relatively improves over the baseline models from **13%– 29.6%** in Precision, from **15.6%–28.9%** in Recall, and from **16.4%–25.8%** in F-score.

# EXTRA SLIDES