TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Fakultät Informatik

Deep learning assisted vulnerability detection and classification in C/C++ source code

Bachelorarbeit im Studiengang Wirtschaftsinformatik

vorgelegt von

Ralph Scheuerer

Matrikelnummer 3076352

Erstgutachter:	Prof. Dr. Tobias Bocklet
Zweitgutachter:	Prof. Dr. Korbinian Riedhammer

Kurzdarstellung

Neue Sicherheitsrisiken und mögliche Exploits in Software tauchen praktisch täglich auf und ziehen für Nutzer und Unternehmen regelmäßig verheerende Konsequenzen nach sich. Sie verursachen auf verschiedenste Weise Schäden, angefangen vom Ausfall unternehmenskritischer Systeme über Identitätsdiebstahl bis hin zu unauthorisierten Zugriffen Dritter auf Finanzinformationen und -konten. Abgesehen von inkorrekter Konfiguration und seltener hardwarebedingten Designfehlern werden derartige Szenarien oftmals von versehentlich eingeführtem angreifbarem Quellcode verursacht. Nachdem die beste Art und Weise dies zu unterbinden darin besteht, möglicherweise fehlerhaften oder unsicheren Code so früh wie möglich im Entwicklungszyklus zu erkennen, haben sogenannte statische Codeanalyse-Tools in den letzten Jahrzehnten stetig steigende Aufmerksamkeit erfahren.

Kürzlich angefertigte Studien beschäftigen sich nun damit, Methoden des Machine Learning und insbesondere Deep learning anzuwenden um Sicherheitslücken zu erkennen, oft mit vielversprechenden Ergebnissen. Ziel dieser Arbeit ist die Implementierung und Gegenüberstellung verschiedener Arten neuronaler Netzwerke für die Erkennung unsicherer Codeausschnitte und in einem zweiten Schritt die Einordnung in vorher festgelegte Kategorien von Sicherheitslücken. Training und Erkennung wird auf Funktionsebene unter Verwendung etablierter Natural Language Processing Methoden geschehen.

Abstract

New possible security issues and exploits in software programs appear every day and regularly have disastrous consequences for both users and software companies. These can cause a wide variety of damages, including downtime of critical business systems, identity theft and unauthorized access to financial accounts and information. Apart from incorrect configuration and rare hardware issues such scenarios are often caused by vulnerable code mistakenly inserted by software developers. Since the best way to prevent this is to catch possibly faulty code as soon as possible in the development cycle, static code analysis tools have seen rising popularity over the last decades.

With the recent hype in machine learning and particularly deep learning, work has also gone into applying these methods to automatically detect such vulnerabilities, often with promising results. This work aims to implement and compare multiple types of neural networks for this task and in a second step, distinguish and classify vulnerabilities of different kinds. Training and detection will happen on function level using well-known Natural Language Processing methods.

Contents

1. Introduction

Security issues in software, hardware and configuration are a major risk for both companies and individuals relying on information technology for their daily business and life. It has become apparent that there is practically no such thing as perfect security, instead it is often viewed as an ongoing process of maintaining and providing updates in order to close newly found safety issues. New possible security issues and exploits in software programs appear every day and regularly have disastrous consequences for both users and software companies. They cause a wide variety of damages, including downtime of critical business systems, identity theft and unauthorized access to financial accounts and information.

In the recent decade, black markets for exploits that are not yet detected and patched matured and grew in size [33]. An industry has formed around the issue of software security, with many companies now offering so called *bug bounties* [6, 25] to honest security researchers. Their goal is to provide an incentive to report exploits instead of selling them to bad actors. Major companies like Intel and Microsoft advertise significant payouts for reporting possibles exploit in their software and service offerings.

Apart from incorrect configuration and rare hardware issues, such devastating security flaws are often accidentally enabled through erroneous source code inserted by software developers. Since the best way to prevent this is to catch possibly faulty code as soon as possible within the development cycle, so called *static code analysis tools* [5] are being offered by various software firms. These tools work by analysing the code statically without executing the actual program. Analysis happens either on the plain source code or, alternatively, on a parsed form of it. This could be bytecode in the case of Java applications or machine code generated by a compiler. So called *dynamic code analysis tools* are different in that they work at runtime and therefore require compilation and execution of the code [5, p. 76]. Not needing to compile the program is one of the benefits of the static approach, because that way security testing can happen before the application is even able to run, similar to syntax validation in compilers and development environments. [5, p. 76]

One of the biggest drawbacks of traditional software security analysis are the rule sets that need to be prepared. Rules contain known patterns of vulnerabilities or simply library functions that are deprecated and known to be unsafe [5, p. 76f.]. The tool matches

these rules against parts of the source code to determine whether it is secure or not. Many applications do not work on the plain source code but rather use a more sophisticated, preprocessed representation of the program. Common approaches are lexical analysis or augmenting it with information from the program's *Abstract Syntax Tree (AST)*, a data structure that describes a program in parsed form. These are typically used internally by compilers and interpreters for code optimization and generation [17]. Still, the underlying issue remains: The performance of these kinds of tools heavily depends on the comprehensiveness and quality of their rule sets. A security bug that is not included will be nearly impossible to detect for them, even though new kinds of vulnerabilities often pose the biggest risks. [5, p. 76f.]

1.1. Related work

There has already gone a lot of work into using machine learning and particularly deep learning for the purpose of estimating software security. A couple of notable examples will be presented in the following paragraphs.

One early architecture named *VulDeePecker (Vulnerability Deep Pecker)* [23] aimed at solving this task was published in february 2018. The study primarily tackles the challenge of ǵenerating suitable representations of programs wihtout requiring manual effort. It introduces so called *code gadgets* and defines them as "a number of [...] lines of code that are semantically related to each other, and can be vectorized as input to deep learning" [23, p. 2]. It asks three fundamental questions in order to judge the success of VulDeePecker: Effectiveness regarding vulnerability detection compared to state-of-the-art approaches, if and how human experts could help further improve VulDeePecker's performance and the feasibility of simultaneously detecting a variety of vulnerability types through deep learning. The framework achieved lower false negative rates than traditional static code analysis and even detected 4 possible exploits in popular software products that were not yet reported. It was concluded that deep learning promises great potential in the field of software security analysis. Still, the authors acknowledged that further research is required in order to realize the full potential and that their study merely presents some fundamental principles for the task at hand [23, p. 13].

[21] investigates the "quantitative impact of different factors on the effectiveness of deep learning-based vulnerability detection, involving more semantic information, imbalanced data processing, and different neural networks" [21, p. 103196]. More precisely, it evaluates the aforementioned *VulDeePecker* on 126 types of security issues versus only two in the original work, introduces a similar system using a different source code parser *Joern* for generating features and explores the effect of considering control dependency

in programs, imbalanced data processing techniques as well as different network architectures [21, p. 103185]. The study concludes that the best results are achieved when programs are augmented with information about both data and control dependency, data sets are balanced through oversampling and classified using *Bidirectional Recurrent Neural Networks (BRNNS)* [21, p. 6f.]. Its limitations are primarily rooted in its focus on external library and API calls and on C/C++ programs. [21, p. 103196]

[13] compares build based and source based features for classification of code snippets. The build-based features consist of the *control flow graph* for each function which was generated using clang and other compiler related LLVM utilities. Furthermore, *opcode vectors* defining specific operations within the function as well as the *use-def matrix* which contains information about variable definitions and usages in the code snippet are included [13, p. 2]. These features were used to train a random forest classifier [13, p. 4]. The second approach was to extract features directly from the source code. Feature embedding happened through both the well-known bag-of-words vectorization and a word2vec representation [13, p. 3]. For the source-based approach, models based on *extremely random trees*, TextCNN architecture and a combination of the two were implemented and compared [13, p. 4]. Results show that the features created from plain source code consistenly lead to superior prediction performance [13, p. 6]. It should be noted that this work was not yet published but still is in preprint status.

SySeVR [22] is short for *Syntax-based, Semantics-based, and Vector Representations*. Its goal is to serve as a framework for representations that contain information about both syntactic and semantic information about programs. This is supposed to help create a common understanding and toolkit for using deep learning to detect vulnerabilities [22, p. 1]. The researchers applied a variety of neural network models and compared their performance to those of traditional software analysis tools. Achieved Results were significantly better than those of conventional analyzers, with the framework even detecting 15 vulnerabilities in the source programs that were not yet known and reported to the *National Vulnerability Database (NVD)* [22, p. 10ff.]. Just like [13], this study also only exists as a preprint so far.

1.2. Goals

This thesis aims to implement and compare multiple types of neural networks for differentiating safe from dangerous code snippets. Training and detection will happen on function level using various established Natural Language Processing methods. *Natural language processing (NLP)* [37] describes a field of study that includes all tasks related to analysis, interpretation and representation of language as humans intuitively use it. These can be based on either text data or spoken language from audio recordings [37,

p. 55]. There will be no features derived from build-related information. Rather, the plain source code will be used as is and investigated whether deep learning is able to learn relevant patterns in the text without the use of compilers or sophisticated program code parsers. Two different feature embeddings and three model architectures will be implemented and their results compared using appropriate measures.

Furthermore it will look whether deep learning could be used to classify known vulnerabilities into groups that share similar characteristics based on a standardized hierarchy of weakness types. A multi-class classifier neural network will be trained and evaluated on testing data to see how it performs in recognition of learned types of security flaws.

2. Data

Deep learning techniques generally benefit from sufficiently sized and high-quality training data. Therefore, when applying such models, a majority of the effort involved is usually related to collecting and processing the source data. This chapter will depict the process of data collection and preparation specifically developed for this thesis.

After the data sets to use are decided, the source code samples will be isolated from another and their representation harmonized and generalized. Finally, information about which functions contain which kind of vulnerability (if any) are gathered and mapped to the respective samples in section 2.3.

2.1. Compiling the data sources

To achieve the best possible real world detection performance, training examples should preferably consist of natural source code that is used in existing real-world applications. However, data sets like that are hard to find. Known security vulnerabilities in code bases of big open source projects generally get fixed quickly and even if one would use an older version that still contained it, the exact part of the source that caused it would be laborious to identify. Crafting a sufficiently large data set through such a manual process was not feasible within the scope of this thesis.

There is an unofficial project by Google called Vulncode-DB [12] that aims to tackle this problem by combining information from the *Common Vulnerabilities and Exposures database*[1] with the relevant commits in git repositories for patching them. Additionally, users can help by providing more precise information about the affected parts of the source code and adding comments about the patch [12]. However, at the time of writing Vulncode-DB was still in alpha stage, with only a few vulnerabilities connected to offending parts of program source code and no interface for extracting the data in an efficient manner [12].

One approach would be to simply run conventional static analyzer tools on collected programs and use that as a benchmark for new, deep learning based systems. However, solely relying on static analysis to generate ground truth might fall short when it comes

[1] https://cve.mitre.org/index.html

to including sophisticated bugs that are more rare. That's why a high quality collection of code containing large amounts of various security issues is needed, along with information on where they are located. The seemingly best way to obtain such data with reasonable effort are so called *Software Assurance Reference Datasets (SARD)* [27] which are managed and provided by the *National Institute of Standards and Technology (NIST)*. These are specifically meant for the purpose of evaluating and developing tools around software security as well as for research on such topics.

Finally the decision was made on three data sets:

- **LibreOffice 6.4.2.2 code base**[2]
  The LibreOffice project is a popular open source office suite capable of creating text documents, spreadsheets, presentations, databases among others. At the time of writing, LibreOffice 6.4.2.2 is the latest version. This code base will mainly serve as a collection of largely safe code snippets, which is a reasonable assumption because of it being an active open source project with many contributors reviewing and patching possible security bugs. That said, this source code will be analyzed using a basic static code analysis tool just as everything else. This will be described in section 2.3.2.

- **IARPA STONESOUP Phase 3 Test Cases**
  The SARD test suite 102 [30] called *IARPA STONESOUP Phase 3 Test Cases* was chosen because of its size as well as how the test cases were generated. It is comprised of 7770 total test cases (each containing multiple functions for different variants) which were created by purposely injecting security vulnerabilities into source code files of 16 different popular open source programs. This method allows for an adequate compromise between completely synthetic test programs and real code bases that lack information about contained vulnerabilities.

  Together with LibreOffice 6.4.2.2, this compilation of test cases will be used for both training and evaluation of the classifiers that will be developed. That combined data set will be abbreviated as `LOSARD102` in the rest of this thesis.

- **Juliet Test Suite v1.3 for C/C++**
  The *Juliet Test Suite (JTT)* [29][28, p. 1] represents SARD-108 and is a collection of test cases aimed at performance testing of static code analysis software published by the NSA Center for Assured Software. It amounts to over 60 thousand individual source code files containing over 100 different error classes.

  The files were created by putting affected source code into a test case template which often also includes a clean, non affected version. This was done automatically using CAS's *Test Case Template Engine* or, in some cases, manually [28, p. 3].

[2] https://downloadarchive.documentfoundation.org/libreoffice/old/6.4.2.2/src/

In the scope of this thesis, a representative excerpt of this test suite will serve exclusively as a testing set for evaluating vulnerability detection on completely separate and unseen data.

2.2. Data preparation

2.2.1. Extracting functions

Isolating function bodies out of a magnitude of unknown source code files of different origin posed somewhat of a challenge.

One of the first approaches was to query the code for function heads using relevant text patterns. *Regular Expressions (RegEx)* [18] provide a syntax for specifying string patterns to find in a larger set of characters. The python code seen in listing 2.1 will apply a regular expression on the string `source_code` which contains the functions from listing 2.2. The RegEx pattern will match the function heads, enabling a script to read the function body in the following lines of code. The RegEx matches in listing 2.2 are highlighted to visualize this.

```
1 import re
2 re.search('(\S+) (\S+) (\w+)::(\w+)\((.*)\)', source_code)
```

Listing 2.1: Regular expression called in Python to capture function heads

However, this method turned out to be too unreliable, missing a lot of functions because of unexpected C/C++ syntax structures while at the same time producing false positives and extracting many incomplete code snippets. With their vast variety of possible syntax, the C and C++ language don't lend themselves to being reliably queried with regular expressions.

After a few experiments with different source code parsing tools, the widely renowned clang[3] compiler for C, C++ and Objective-C based on the LLVM Compiler Infrastructure project[4] proved to be up to the task since it offers rich features beyond just a compiler frontend. The C++ library `libclang` [24] provides a C programming interface for accessing clang features. Apart from the native libraries it also features bindings for use in Python[5] which were used in this work. Using libclang it was possible to generate and traverse clang's Abstract Syntax Tree for the source files. These allowed reliable identification of function and method declarations. Both the actual code snippets and

[3] http://clang.llvm.org/
[4] https://llvm.org/
[5] https://github.com/llvm-mirror/clang/tree/master/bindings/python

```
1  // XAnimationNode
2  sal_Int16 SAL_CALL AnimationNode::getFill()
3  {
4      Guard< Mutex > aGuard( maMutex );
5      return mnFill;
6  }
7
8
9  // XAnimationNode
10 void SAL_CALL AnimationNode::setFill( sal_Int16 _fill )
11 {
12     Guard< Mutex > aGuard( maMutex );
13     if( _fill != mnFill )
14     {
15         mnFill = _fill;
16         fireChangeListener();
17     }
18 }
```

Listing 2.2: Content of source_file

Dataset	Samples
LibreOffice	292724
SARD-102	65189
LOSARD102 (combined)	227535
JTT	24999

Table 2.1.: Number of extracted C/C++ functions

meta information such as their respective filepath and relevant line numbers within that file were stored. That information will be used to match the extracted functions and found vulnerabilities in the labeling step (section 2.3). Table 2.1 shows the number of functions and therefore samples retrieved from the chosen datasets.

2.2.2. Identifying library and API calls

In section 2.2.3 the code samples will be manipulated in order to reduce the overall vocabulary. Before that, calls to library functions in the examples need to be identified, isolated and measures implemented to preserve them during that process. This is necessary since possible vulnerabilities can be caused by either using deprecated library functions that are not deemed safe to use anymore, or by erroneously calling them in an unsafe fashion. Externally defined in this case means that the definition is located in system libraries like `stdio.h` or in a third party dependency that the program uses.

The names of these methods will were isolated and saved for each code snippet so that they can be spared in the lexing step. For this task, the Abstract Syntax Tree provided by LLVM and clang was utilized once again. In order to find all external functions for a given example, the containing file's AST was traversed, ignoring all tree nodes that don't lie between the `line_start` and `line_stop` for that sample. LLVM provides a `kind` property for each node in the tree that was used to identify all nodes of `CursorKind.` ↪ `CALL_EXPR` which refers to method and function calls [24]. When examining such an expression, `get_definition()` returns a pointer to the node containing the definition for that function. If that definition is located outside, `get_definition()` will return `null` and the function name was added to the list of external function calls in that sample.

2.2.3. Lexing of code snippets

2.2.3.1. Get output of clang's lexing step

As mentioned above, a lexing step was applied to the source code samples instead of feeding it into the classifier as-is. The goal here was to reduce the overall vocabulary size and eliminate program specific names as much as possible. Thankfully, once again clang provides a raw output of the compiler's lexing step via the command line. The extracted function body can be forwarded into clang as demonstrated below.

```
1 echo code_sample | clang -cc1 -x c++ -dump-tokens
```

An excerpt of the output is shown in listing 2.3. The complete command output for the two functions in listing 2.2 can be found in appendix A.2.

`clang` assigns categories to all C/C++ tokens such as `identifier` for developer-defined variable and function names, `l_paren` and `r_paren` for opening and closing parenthesis, `return` for return statements and more. Code comments are automatically discarded in

```
1  identifier 'Guard'        [StartOfLine] [LeadingSpace] Loc=<<stdin
       ↪ >:5:5>
2  less '<'                  Loc=<<stdin>:5:10>
3  identifier 'Mutex'         [LeadingSpace] Loc=<<stdin>:5:12>
4  greater '>'         [LeadingSpace] Loc=<<stdin>:5:18>
5  identifier 'aGuard'        [LeadingSpace] Loc=<<stdin>:5:20>
6  l_paren '('               Loc=<<stdin>:5:26>
7  identifier 'maMutex'       [LeadingSpace] Loc=<<stdin>:5:28>
8  r_paren ')'         [LeadingSpace] Loc=<<stdin>:5:36>
9  semi ';'                  Loc=<<stdin>:5:37>
10 return 'return'   [StartOfLine] [LeadingSpace]    Loc=<<stdin>:6:5>
11 identifier 'mnFill'        [LeadingSpace] Loc=<<stdin>:6:12>
12 semi ';'                  Loc=<<stdin>:6:18>
```

Listing 2.3: Excerpt of clang -dump-tokens output

this step since they don't show up in the command's output. This is desirable since they don't influence program functionality and therefore should not be included into feature embedding.

2.2.3.2. Generalizing and pseudonymizing source code tokens

Before the identified code samples can be labeled, they have to be converted into an appropriate representation, utilizing the output of previous steps.

Strings and Chars. Statically defined or *hard-coded* strings and chars in source programs are assumed to not affect their security and will therefore be discarded. The relevant token categories are

- `string_literal`
- `wide_string_literal`
- `utf16_string_literal`
- `char_constant`

These tokens are replaced by

- `<string_literal>`
- `<wide_string_literal>`
- `<utf16_string_literal>`
- `<char_constant>`

Numbers. Numbers are represented by the clang token category `numeric_constant`. All numeric constants are replaced by the string `<numeric_constant>`.

Identifiers. Tokens of type `identifier` are handled in a more sophisticated manner. First, each token is checked on whether it is the name of an externally defined function. This is achieved by looking up the list of external function names that were stored for each sample in step 2.2.2. If a name is contained in that list, it will be preserved as is. Otherwise, the token at hand is some other kind of developer-defined name, e. g., a variable name. These should be discarded as they aren't relevant for code vulnerability, but if the same variable is used and manipulated multiple times within a code sample, that information could be valuable for recognition of repeating patterns. To address this, all identifiers are replaced by the string `<identifierX>`, where `X` is an incrementing integer that will be the same for occurences of the same identifier within one code sample.

Other. If a given token does not belong to any of the aforementioned categories, it will remain unaltered. This way, symbols and language components such as semicolons, `return`, `void` but also data types like `int` and `char` will be preserved for feature embedding. Furthermore, all whitespace is discarded.

Listing 2.4 shows an example of how the previously shown two functions will look like after this step is completed. Note that whitespace was reinserted here for better readability.

```
1   // Code sample 1
2   identifier0< identifier1 > identifier2( identifier3 );
3   return identifier4;
4
5   // Code sample 2
6   identifier0< identifier1 > identifier2( identifier3 );
7   if( identifier4 != identifier5 )
8   {
9       identifier5 = identifier4;
10      fireChangeListener();
11  }
```

Listing 2.4: Content of source_file

2.3. Labeling

After completing the steps described above, all data sets are now compiled into a common representation with a drastically reduced vocabulary. This section will describe the labeling process that maps information about contained vulnerabilities to samples in the data set. It is necessary since the classifiers need to be able to learn from training data with known labels. This kind of learning or *training* process is called *Supervised Learning* and will be depicted in chapter 3 (section 3.3.1).

The definition of what encompasses a security vulnerability and furthermore how such issues can be differentiated into types poses a challenge. Fortunately the MITRE Corporation, a north american government-funded research organization, maintains the *Common Weakness Enumeration (CWE^{TM})* [7]. Its goal is to provide an enumeration of common types of security flaws in both software and hardware. At the base level, these types are catalogued as individual *Weaknesses*. MITRE defines Weaknesses as "flaws, faults, bugs, vulnerabilities, or other errors in software or hardware implementation, code, design, or architecture that if left unaddressed could result in systems, networks, or hardware being vulnerable to attack" [7].

Since this work seeks to train classifiers for both *safe vs. vulnerable* detection as well as multi-class classification of weaknesses into different clusters of related vulnerabilities, two different sets of labels needed to be generated. One is be the simple binary 0/1 label, while the other is a vector that shows occurrences of weakness types. Each vector component represents a single weakness type as one binary value (one bit) that indicates that specific type.

2.3.1. From test suite metadata

In the case of the IARPA STONESOUP and Juliet Test Suite, labeling is relatively easy. Since the test suite is comprised of source code files of free software projects purposely injected with various vulnerabilities, high-quality ground truth about where these are located is made available by the authors. Each collection of test cases in the Software Assurance Reference Dataset is accompanied by a `manifest.xml` containing that information. An entry consists of the CWE type, relative file path and line number where a vulnerability can be found. Listing 2.5 shows an example of an entry that points out a vulnerability of type *CWE-476* in the file `utf.c`.

Vulnerabilities in test cases can span over multiple lines but are always contained within one function or method. This information is used to match the XML entries with the extracted code samples via line numbers and file path. Samples with no matches to an entry in the manifest file are deemed to be safe.

```
1  <testcase id="149249" type="Source Code" status="Candidate"
     ↪ submissionDate="2015-10-06" language="C" author="IARPA STONESOUP
     ↪ Test and Evaluation team" numberOfFiles="5" testsuiteid="102"
     ↪ applicationid="7">
2    <description><![CDATA[This test case reads a space-delimited string
     ↪ from the taint source. The first element in the string is the number
     ↪  of elements following it. The test cases reads in the following
     ↪ elements and outputs them. If there are fewer elements than expected
     ↪ , a segmentation fault occurs. <br />Metadata<br /> - Base program:
     ↪ Subversion<br /> - Source Taint: ENVIRONMENT_VARIABLE<br /> - Data
     ↪ Type: STRUCT<br /> - Data Flow: INDEX_ALIAS_50<br /> - Control Flow:
     ↪  MACROS<br />]]></description>
3    <file path="shared/102/scripts/runFifos.py" size="2736" checksum="9507
     ↪ bed983a10b7da25b9e7cf7aa5ccbae3c305e"/>
4    <file path="shared/102/scripts/service_mon.sh" size="100" checksum="36
     ↪ ffb4791fcba87916b9295ffa9d6d73b5b9a696"/>
5    <file path="000/149/249/C-C476C-SUBV-03-ST01-DT05-DF10-CF22-01.yaml"
     ↪ size="3087" checksum="d9b8598b3674befb9b73e08bd44a47f21bdd2906"/>
6    <file path="000/149/249/src/subversion/libsvn_subr/utf.c" language="C"
     ↪ size="45216" checksum="f67348d3ac04807b63db800b633cd0976383fd8c">
7      <flaw line="1130" name="CWE-476: NULL Pointer Dereference"/>
8      <flaw line="1131" name="CWE-476: NULL Pointer Dereference"/>
9      <flaw line="1133" name="CWE-476: NULL Pointer Dereference"/>
10     <flaw line="1134" name="CWE-476: NULL Pointer Dereference"/>
11     <flaw line="1128" name="CWE-476: NULL Pointer Dereference"/>
12     <flaw line="1129" name="CWE-476: NULL Pointer Dereference"/>
13     <flaw line="1132" name="CWE-476: NULL Pointer Dereference"/>
14   </file>
15   <file path="000/149/249/C-C476C-SUBV-03-ST01-DT05-DF10-CF22-01.xml"
     ↪ size="62169" checksum="110bb51c39934b347f2979d60d52ffabdd1ca5d8"/>
16 </testcase>
```

Listing 2.5: Example flaw entry in manifest.xml [30]

2.3.2. From traditional static analysis

For the rest of the data set, ground truth must be determined without the help of auxiliary manifest files. Given that LibreOffice is an established open-source project, one valid option would be to declare all contained code samples as safe. However, in order to create a larger and more diverse set of vulnerable samples, a basic static analysis tool named *Flawfinder* [35] is run on all three datasets and merged with labels generated from test suite auxiliary data.

Flawfinder was created by David A. Wheeler and promises to detect the most common mistakes in regard to C and C++ program security. It comes with a built-in database of C/C++ functions and patterns known to induce problems such as buffer overflows, race conditions, insecure random number generation or others [35]. Instead of trying to examine the actual semantics or control flow in the target program, Flawfinder simply searches for the patterns contained in its database. However, it does look into parameters in function calls to estimate the risk level for a hit [35]. This method makes the tool easy to run while still finding a few more vulnerabilities in both the test suites as well as the LibreOffice codebase. `flawfinder --listrules` prints a list of all included rules that can be seen in appendix A.3.

In this case, Flawfinder version 2.0.11 was the current version and was executed with the `--falsepositive` flag. Furthermore, the minimum risk level to 3 out of 5. Both of these options are supposed to reduce the number of false positives when running the tool [36]. The complete command is shown below:

```
1 flawfinder --falsepositive --minlevel=3 --dataonly --quiet --csv
```

Similarly to the labels deduced from test suite manifest files, the analyzer's output contained the relative filepath, CWE type and the line number of the offending statement. Using this static analyzer, another 42,689 possibly unsafe functions in LOSARD102 and 19 in Juliet's Test Suite were identified.

2.3.3. Binary labels

The binary label was defined by using 0 for safe and 1 for vulnerable source code samples. A code snippet would get defined as vulnerable if any one CWE was detected within the function and safe otherwise. Note that from now on samples labeled `1` will be sometimes refered to as *bad* and treated as *positives* while all others will be called *good* and defined as *negatives*.

As can be seen in table 2.2, the vast majority of labeled functions are *good*, meaning no vulnerabilities were found in the labeling step. This means that the *bad* samples, while being the critical type of sample, are largely underrepresented. Section 3.2 will explore strategies on how to mitigate this effect.

Dataset	Good	Bad	Total
LibreOffice (LO)	64951	238	65189
IARPA Test Suite (SARD102)	204770	22765	227535
Combined (LOSARD102)	269721	23003	292724
Juliet Test Suite (JTT)	2417	828	24999

Table 2.2.: Overview of C/C++ functions and corresponding labels

2.3.4. Multi-class labels

When it comes to classifying types of security issues, instead of the binary label derived in section 2.3.3, the values generated in sections 2.3.1 and 2.3.2 are used directly. They are each made up of 79 bit values representing the different CWEs that are included in the data set. Since a 79-class classifier would be hard to train on such relatively few samples and in order to achieve a better degree of generalization, these 79 CWE types were condensed into groups. The MITRE Corporation provides multiple hierarchical views meant to sort the multitude of CWE types into more general clusters, with each view catering to different topics and uses. The one chosen for this task is called *By Research Concepts* [8] and summarizes all 875 current CWE weakness types into ten groups that are defined as follows:

CWE ID	Description
284	Improper Access Control
435	Improper Interaction Between Multiple Correctly-Behaving Entities
664	Improper Control of a Resource Through its Lifetime
682	Incorrect Calculation
691	Insufficient Control Flow Management
693	Protection Mechanism Failure
697	Incorrect Comparison
703	Improper Check or Handling of Exceptional Conditions
707	Improper Neutralization
710	Improper Adherence to Coding Standards

Table 2.3.: CWE-1000: By Research Concepts

CWE	284	435	664	682	691	693	697	703	707	710
Good	23003	23003	9549	22332	22239	11693	22893	23003	9681	21672
Bad	NaN	NaN	13454	671	764	11310	110	NaN	13322	1331

Table 2.4.: CWE weakness cluster occurrences in vulnerable LOSARD102 samples

Each of these high-level clusters contains a collection of weaknesses which are described by individual CWE IDs. The ten clusters don't overlap, meaning each kind of vulnerability can be distinctly mapped to exactly one *Research Concept*. If a given code example was found to contain one or more weaknesses of a given group, it will be labeled as a positive for that whole group. Similarly, if it were to contain CWE from different Research Concepts, it would be positive for all of them.

Table 2.4 shows the distribution of labels within all vulnerable functions in the combined LOSARD102 data set. There are no positives for the Research Concepts *CWE-284*, *CWE-435* and *CWE-703*. Some CWE clusters such as *CWE-691* are much more common than others, leading to imbalanced ground truth labels which will be discussed in chapter 3 (section 3.2).

2.4. Summary

High quality data are crucial for training and evaluating the classifiers that will be implemented in this work. In this chapter, three major data sets were defined that contain source code from large open source software as well as synthetically generated programs. Since classification in this case shall happen on function level, all C/C++ functions were identified using Abstract Syntax Trees. Information not relevant for program security such as given names was then substituted by generic placeholders, consequently reducing the overall vocabulary. Critical information like usages of system and third-party libraries were kept during this process.

Finally, two different methods were applied to map information about contained vulnerabilities to the samples collected before. Two of the three data sources came with metadata about the location of these security issues which were mapped to the corresponding training and testing data. In addition, a conventional static analysis tool was used to detect and label some more vulnerabilities. Data from these two methods were merged with all training and evaluation samples to build the final sets of labels.

Apart from the labels showing whether a given function is safe or not, a second set of labels was produced denoting the general kind of vulnerability which will be used for type classification in section 3.4.

3. Method

This section will describe the central methods that were used int this thesis. First, the techniques used to create features for deep learning from the tokenized samples compiled in chapter 2 are shown. In section 3.2, three alternative techniques of compensating for the effects of biased data sets will be described. Finally, the fundamentals of modern neural networks and deep learning will be introduced. Sections 3.3.1 to 3.3.3 discuss the network architectures benchmarked in the next chapter.

3.1. Feature embedding

Before actual models can be trained on the prepared data, so called *features* need to be generated from that tokenized code representation. Machine learning and deep learning models demand numeric vectors or matrices in order to calculate output vectors based on that input. The task of capturing word and sentence meaning in numeric vector space constitutes a central challenge for NLP related research, which is why there are a multiple established techniques of how to do so. Two of those were chosen for comparison in this work and will be discussed in the following sections.

3.1.1. Word2Vec

The first feature embedding that will be implemented is the *Word2Vec* framework published in [26]. Its goal was to use the advanced techniques made possible by recent developments in machine learning and NLP to derive word embedding from large collections of text. Simpler embedding techniques such as *N-gram models* may be widely tested and robust but would lack the ability to learn vocabularies from huge data sets which might lead to significant performance increases [26, p. 1].

In this case, the variant using a *Continuous Bag of Words* architecture was employed. The name stems from the fact that similarly to the traditional *Bag of Words (BOW)* embedding it does not take the order of words in a sentence into account. However, in contrast to BOW it uses a *continuous distributed representation* and therefore does consider context. This is achieved by including nearby words from both before and after a given word when determining embeddings [26, p. 4]. Figure 3.1 shows how the CBOW

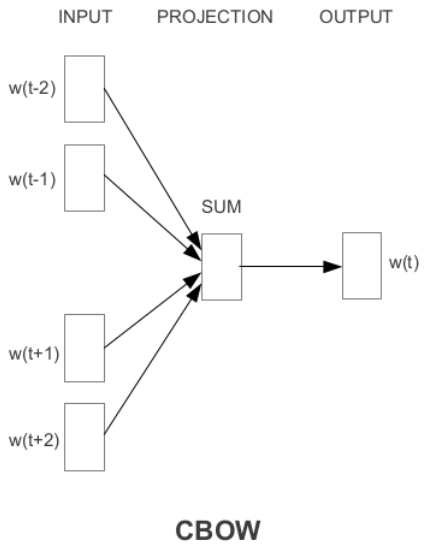


Figure 3.1.: Schema of the CBOW architecture (taken from [26, p. 5])

architecture uses context from `t-i` to `t+i` where `t` is the current word's location and `i` is the number of words to include in either direction.

Generally, the size of the base vocabulary and higher dimensionality of the embedding vector space is supposed to further improve the quality and performance of these word vectors [26, p. 10]. In this case, the output size and therefore the feature vector dimensionality was set to 100 which is also the default value in some deep learning libraries.

Figure 3.2 shows the output of the word2vec model trained on the LibreOffice codebase split up into tokens as described in earlier sections. t-SNE reduction [34] was used to reduce the hundred dimensional vectors into flat two dimensional space. t-SNE is a technique for visualizing high-dimensional data that regularly produces superior results compared to previous algorithms. It performs better especially at preserving relations and distance between but also within clusters [34, p. 1]. Note that for visualization purposes, distinct identifiers were not preserved but rather represented by a universal `<identifer>` token. It is apparent that similar tokens that often appear in the same context tend to be grouped together, e. g., logic operators, control flow structures like `if...else`, `do...while` or types such as `double`, `long`, `char`, `int` as well as `bool`, `true` and `false`. A vector representation incorporating this kind of context might provide a good basis for identifying patterns in source code extracts.

After the model was trained on the whole vocabulary in the data sets, it was used to predict a vector for each word (each token) in the pre-processed code samples. For this thesis, two sets of word2vec feature vectors will be prepared: The first one, called `w2v_avg`, represents every sample as a single 100 dimensional vector and simply is the

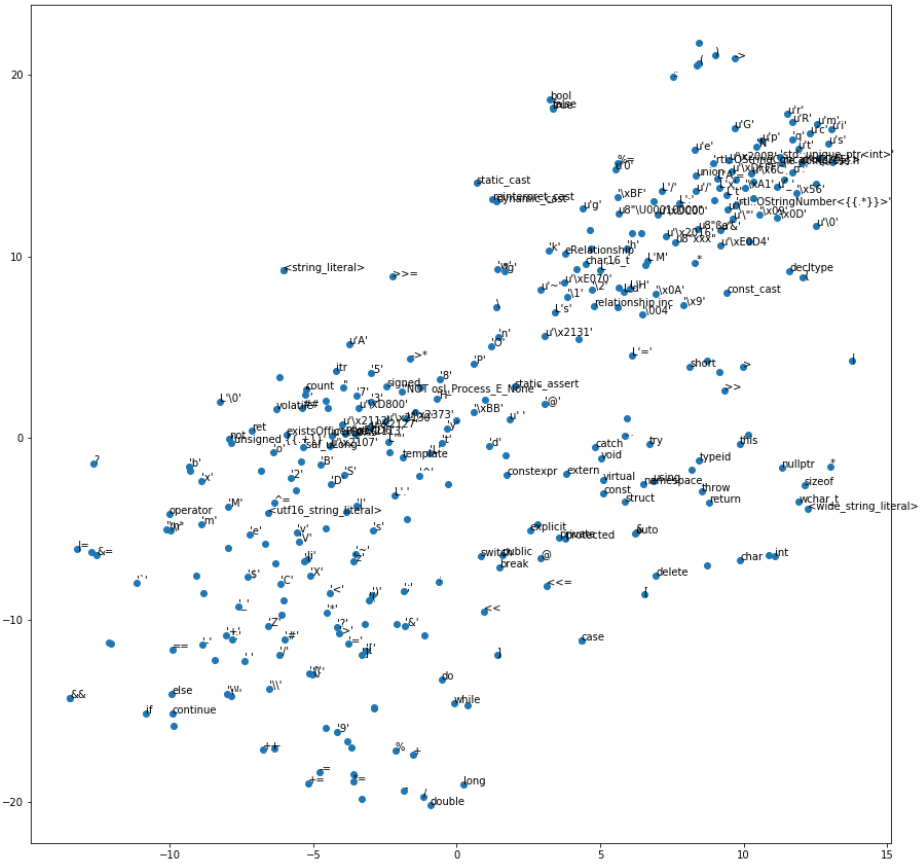


Figure 3.2.: word2vec plot of the LibreOffice code base with all identifiers omitted

average over all word vectors in a given sample. Apart from that, a second set called `w2v_seq` was created by saving the individual word vectors as is. Because of resource limitations, only samples with a maximum length of 100 tokens were included. Shorter samples were then padded with zeros in order to obtain the same number of vectors for all samples. This sequence-oriented data set might enable the appropriate classifier model architectures to learn context within functions, possibly further improving their understanding of the source code at hand.

3.1.2. BERT

Another recently introduced technique of word embedding is the so called *BERT* framework [9] published by Google's research team. It is the second approach to building feature vectors from plain text that will be implemented. Just like with Word2Vec, the source texts will be the tokenized function bodies prepared in chapter 2.

BERT is an abbreviation for *Bidirectional Encoder Representations from Transformers* [9, p. 4171]. This framework follows a principle called *pre-training* where it "is trained on unlabeled data over different pre-training tasks" [9, p. 4173]. Google's research experts state that previous methods based on pre-trained networks were limited by their purely unidirectional language models [9, p. 4171]. Therefore, one of the primary goals was to use bidirectional context as opposed to left-to-right language pre-training. BERT was able to score significantly better in popular NLP tasks such as *GLUE* or *MultiNLI* [9, p. 4171f.].

The BERT model initiated with pre-trained parameters will be used to predict word embeddings and thus generate feature vectors, but not for the actual supervised learning task. In this case, the `bert_en_uncased_L-12_H-768_A-12` variant, also called *BERT-Base* [11], was selected. This model was pretrained on lowercase english language and features 12 layers with a hidden size of 768. Many programming languages do expect specific casing of keywords and usually treat developer-given names for entities like variables and methods as case-sensitive. However, when learning patterns in source code, the actual casing of keywords and other language components should play a smaller role since the tokenized source code is already lowercase and library calls will still be unambiguously recognizable. Additionally, converting everything to lowercase could help further standardize the code representation across the three different data sources. Google recently also published a collection of smaller BERT models for use in environments with fewer computing resources [11].

This technique was solely used to generate sequence-preserving features for the models that are able to take time sequence and context into account. Because of the considerable amount of computing effort required to predict embeddings using BERT and the large

amount of high-dimensional vectors involved, the data sets were culled to only include samples with a sequence length of 256 or fewer. As with the `w2v_seq` dataset, the sentences were zero-padded to the common length of 256 afterwards.

3.2. Mitigation of imbalanced classes

As briefly mentioned earlier, these kinds of datasets entail an inherent bias towards one or more classes. The majority of source code will be safe by today's technical standards, even more so when talking about big open-source projects. In those cases, many eyes will review a piece of code before it's submitted to the next version. Furthermore, some larger projects even undergo professional security audits and publish the results, such as the audits around core components of the *Kubernetes* project [2]. In the case of classifying the ten vulnerability types defined in section 2.3.4 there is a similar situation. It is improbable for these ten classes to be distributed evenly over all training data.

If the underlying dataset is imbalanced, simple evaluation by measuring predictive accuracy is insufficient. That is because e. g., if 80 percent of a binary labeled dataset belong to class 0, a binary classifier that simply predicts 0 for every input will achieve 80 percent accuracy. This inflated prediction accuracy doesn't tell the whole truth for the purpose of evaluation [4, p.322]. Fortunately, this is a common challenge when developing classifiers and there are ways to mitigate this effect, some of which will be discussed in this section and their performance compared in subsequent chapters.

3.2.1. Undersampling

One straight-forward way of ensuring balanced representation of all possible label values is to simply use a subsample of the majority class's data points. This subsample will be the same size as the total number of examples for the underrepresented class. So if e. g., there are 100 examples for class 0 and only 50 for class 1, 50 samples of class 0 (the majority class) will be randomly chosen and combined with all 50 of class 1. This method has the drawback of completely leaving out some examples, thus removing possibly valuable information from the training set. It also shrinks the total size of training data the classifier can use to learn from.

3.2.2. SMOTE Oversampling

Other methods of dealing with bias in ground truth rely on oversampling the minority classes until they encompass the same amount of samples as the majority class. A

popular variant of this mitigation measure is the so called *Synthetic Minority Oversampling Technique (SMOTE)* [4]. It aims to augment the existing data points of underrepresented classes with synthetically generated feature vectors that exhibit similar properties in their features (and therefore are "neighbours" in the feature vector space). These additional training samples are calculated by determining the difference between a given sample and its most similar neighbors in the feature space. This difference vector is then randomly weighed between 0 and 1 and subsequently added to the original feature vector. The number of neighbors to be considered depends on how many vectors need to be generated. This process is repeated for each example of the minority class [4, p. 328]. That means the synthetic samples always lie in between two or more samples that are actually contained in the data set, creating a more dense cluster of feature vectors, thus not deviating from the general characteristic properties of the class. It was shown that models trained on training data augmented by SMOTE exhibited better prediction performance on the underrepresented classes than those using other oversampling techniques [4, p. 329].

The calculation of synthetic samples through SMOTE introduces some amount of computing time, especially for larger data sets. On the other hand, it augments the existing training data with new synthetic data, therefore providing at least as much training samples as the original set.

3.2.3. Class weights

The last mitigation method this thesis will explore does not involve changing the actual training data, but rather the learning process. To prevent the model from developing bias towards the majority, weights are introduced that serve as multipliers for calculation of the network's prediction error. This effectively means that misclassifying a sample of the less represented class will lead to a much higher error cost than one of the majority class. These *class weights* are inversely correlated to the proportion between number of samples of the classes. The calculation formula was defined as

$$w_i = \frac{1}{s_i} \cdot \frac{s}{n}$$

where w_i is the error weight for class i, s_i is the number of training samples in class i, s is the total number of training samples and n is the number of classes in the data set.

3.3. Classification of safe and vulnerable functions

As was stated in the introduction, the primary goal of this thesis is to compare different approaches to creating classifier for a *safe vs. vulnerable* prediction of source code snippets on function granularity level. In the following sections, the actual models used to build and train this binary classifier will be discussed.

3.3.1. Multi-layer perceptron (MLP)

Neural networks [10] as used in this thesis are enablers of many modern machine learning applications. The term *network* originates from the basic structure of these models. To build a neural network, a series of functions are chained together, creating multiple *layers*. The first layer receives the input vectors and the last or final layer is also called the *output layer*; In between those two there can be an arbitrary amount of *hidden layers* where the number of layers creates *depth* in the network model [10, p. 164f.]. The depth of these models enables them to learn more complex non-linear relationships between inputs and outputs. For *supervised learning*, the model gets fed both *feature vectors* `x` and corresponding labels or more generally *target vectors* `y` that define the desirable outcome for each example [10, p. 164f.]. The term *neural network* was coined by the original desire to imitate human brain functionality through software. However, nowadays the field of deep learning is primarily driven by the disciplines of advanced mathematics and engineering. Its goal is not actually coupled to modeling animals' brains and the term is mostly kept for historical purposes.

Neural Networks [10] learn by comparing their predicted output for each input vector with the correct output vector, also called the *target vector*. That difference is then used to calculate the *error* by means of an *error function* that was chosen beforehand. Training a neural network means gradually adjusting the weights for each connection between individual network nodes in such a way that prediction errors are minimized, leading to better overall accuracy. *Back propagation* and *gradient descent* [32] are utilized to determine necessary adjustments for each weight by computing derivatives of the error along all connections within the neural network. Weight adjustment can either occur after every single input-output relation or only after all training samples were processed. In the latter case, the derivatives are still computed for every sample but accumulated before the aggregated gradient descent step. In its basic form, gradient descent uses this accumulated derivative as follows [32, p. 535]:

$$\Delta \mathbf{w} = -\epsilon \frac{\partial E}{\partial \mathbf{w}}$$

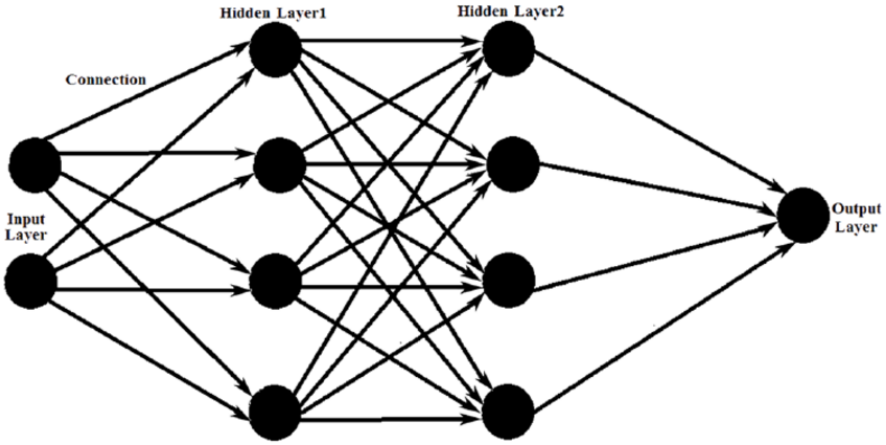


Figure 3.3.: MLP with two hidden layers (taken from [3])

where E is the total error over all input-output pairs, $\mathbf{w}$ is the weight in question, Δw is the signed amount of change to apply to $\mathbf{w}$ and finally, ϵ is a factor that controls how much this step in gradient descent influences the weights and is therefore also referred to as the *learning rate*. $\frac{\partial E}{\partial \mathbf{w}}$ is calculated through use of the chain rule [32, p. 534.]:

$$\frac{\partial E}{\partial \mathbf{w}_{ji}} = \frac{\partial E}{\partial \mathbf{x}_j} \cdot \frac{\partial \mathbf{x}_j}{\partial \mathbf{w}_{ji}} = \frac{\partial E}{\partial \mathbf{x}_j} \cdot \mathbf{y}_i$$

where j is an index for output units, i is an index over input units, $\mathbf{w}_{ji}$ is the weight for the connection from unit i to j, $\mathbf{x}_j$ is the input to unit j and y is the actual value in that output unit for the current input. Thanks to the chain rule, this can be extended to deeper networks of more than two layers.

Iterative processes like this generally face two challenges in real world applications [31, p. 8]: First, iteratively minimizing the overall loss could lead to the algorithm getting stuck in a local minimum and never reaching the best possible performance. Secondly, even if the learning process would always eventually reach a global minimum, the question remains how much computing resources and therefore time this would take. However, the mechanism of backpropagation and gradient descent proves to generally be robust regarding these concerns [31, p. 8].

The first network for the binary classification of C/C++ functions will be a simple multi-layer perceptron model. MLP networks represent basic topologies for building artificial neural networks. They are also called *feed-forward networks* because information solely flows from front to back, as in from the input layer through hidden layers to the output layer. In the case of *fully connected* networks, all neurons in each given layer are connected to every neuron in the following layer [10, p. 164f.]. The basic structure of a Multilayer Perceptron model with two hidden layers is illustrated in figure 3.3.

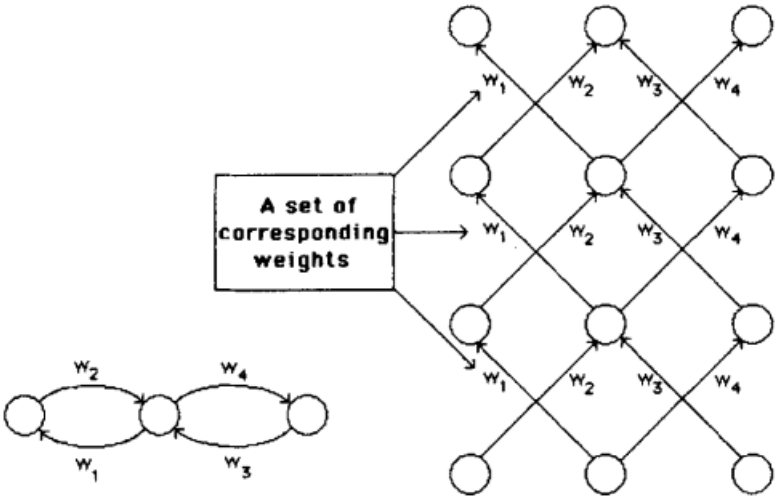


Figure 3.4.: Unfolding a RNN into a conventional neural network (taken from [32, p. 535])

3.3.2. Long short-term memory (LSTM)

The MLP networks described previously only take one single vector as input. In order to actually take sentences of source code and thus context between the "words" into account, the input would have to be a matrix built from a sequence of word vectors. The following paragraphs will now look at another type of neural networks that is better capable of processing these kinds of inputs.

The general go-to architecture for these kinds of tasks are *Recurrent Neural Networks (RNN)* [16]. RNNs use *feedback connections* from back to front and within layers to store and "remember" state information about recent inputs. It enables them to simulate *short-term memory* and directly use recent inputs to influence prediction for current inputs. This ability is crucial for improving applications related to time-dependent sequential data such as speech or music [16, p. 1]. Since text and thus source code also represents language or "speech", these kinds of architectures are also appropriate to employ in the context of this work. Theoretically, every RNN could also be replicated as a feed-forward network. However, this brings a much higher complexity with it Figure 3.4 demonstrates how a simple recurrent network can be unfolded into an equivalent feed-forward neural network (with some necessary adjustments) [32, p. 535].

Earlier methods of using recurrent neural networks to learn from time-dependent data often suffered from errors either exploding and consequently sabotaging the learning process, or vanishing [16, p. 4]. The latter phenomenon is caused by dependencies along the network's internal connections and leads to exponentially decreasing weights, thus drastically slowing down or completely prohibiting the learning process [15, p. 64]. The

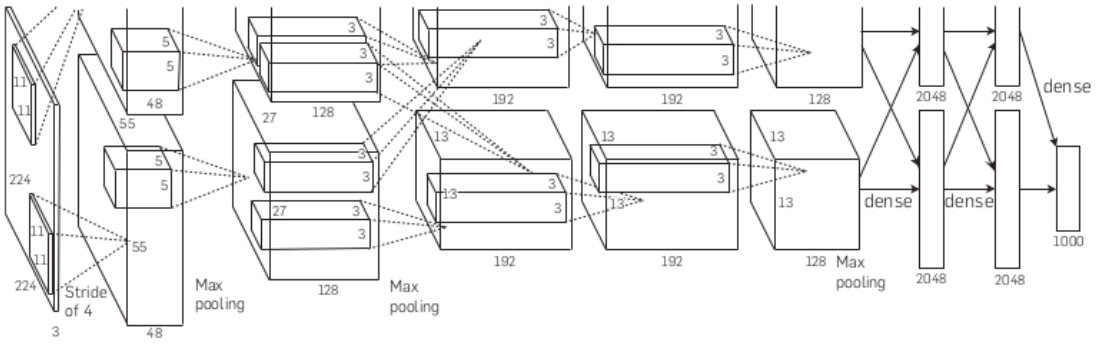


Figure 3.5.: Exemplary topology of deep convolutional neural networks (taken from [19])

new architecture for recurrent neural networks introduced by [16] is called *Long short-term memory (LSTM)* and aims to avoid this. To make it more resilient to the challenges mentioned above, *memory cells* are responsible for storing error information within the network. These are discrete components built around a self-connected linear unit accompanied by *input* and *output gates*. Those *gates* can learn to prevent irrelevant inputs from entering the memory cell unaltered, but also to apply appropriate scaling to its output. This way, the training process can tune gate parameters such that they stop possibly inappropriate short-term error information from interfering with long-term memory built from weights learned over many iterations [16, p. 1][16, p. 6f.]. This improved RNN architecture proved to learn faster and more successful than traditional architectures also based on recurrent neural networks, specifically for remembering over longer time spans [16, p. 1f.]. Another advantage of LSTMs is their general robustness regarding hyperparameters. Learning rates and other parameters do not require a lot of tuning for the network to perform well [16, p. 23].

3.3.3. ResNet

The third type of classifier model that will be benchmarked in the task of vulnerability detection will be so called *Convolutional Neural Networks* [38][20]. In the last decade, CNNs became increasingly crucial for complex image recognition tasks and nowadays play a central role in that field [38, p. 818f.]. These types of networks use convolutional layers where the input gets split up into regions in order to identify local features of an image. This allows the network to learn locally contained patterns that can appear anywhere within a picture. *Feature maps* in hidden layers each take one of the partial images to try and learn from them [20, p. 542f.]. Figure 3.5 visualizes the topology of a deep convolutional neural network that was used for prediction of 1000 classes in the *ImageNet* data set [19].

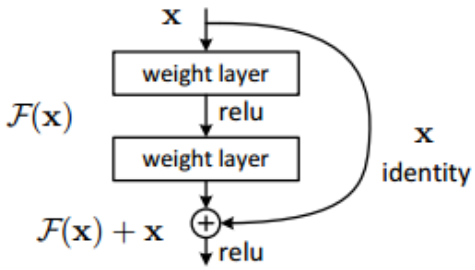


Figure 3.6.: Single building block for residual learning (taken from [14, p. 771])

More recent works found higher depth to be significant for such networks to perform well [14, p. 770]. However, deeper models are more resource intensive to train and exhibit a phenomenon described as the *degradation problem* [14, p. 770f.]: During the training process, accuracy and error curves of the network get slowly saturated (as expected) before suddenly improving drastically again. To solve this, [14] introduced the concept of *Deep Residual Learning* which - in the case of feed-forward networks - adds *shortcut connections* to cause stacks of hidden layers to learn residual mappings in relation to the *identity mapping*. An example for such a connection can be seen in figure 3.6. Networks with these components can still be trained regularly through backpropagation and gradient descent [14, p. 771].

Evaluations on popular challenging data sets like *ImageNet* indicated significantly better prediction accuracy for extremely deep networks of hundreds of layers using the technique of residual learning. In this thesis, the *ResNet* model introduced by [14] will be applied to the collected data sets. The series of word vectors was used as one dimension while the depth of word vectors created through Word2Vec or BERT constituted the second dimension. This way samples were converted into two dimensional input planes that ResNet expects.

3.4. Classification of vulnerabilities into CWE clusters

So far the goal was to create a binary classifier that predicts a source code snippets's probability to contain some kind of security vulnerability. The second question that was asked in the beginning was whether it would be possible to reliably classify the general types of security issues found in source code. This section will discuss the latter question.

The only samples included in these experiments are *bad* samples since in this case, the goal is merely to differentiate types of security flaws instead of detecting them. Thus, only the 23,003 positives in the `LibreOffice` and `SARD-102` datasets were used to create the base data for these experiments, albeit with one caveat: About half of those vulnerable

examples were marked as positive for more than one of the seven major CWE clusters via the labels generated in section 2.3.4. This circumstance introduces a significant change to the classification problem since now samples need to get possibly sorted in multiple different classes through the network's output. So called *Multi-label classification* where samples can be given multiple labels constitute much more challenging problems in deep learning applications. Therefore, a second variant was added only including *distinct* samples that only belong to a single CWE cluster label. This left 9,859 data points and enabled the model to make definite predictions for each sample. Training, validation and test data were generated by dividing these into a 60/20/20 split with 5915 samples for training, 1972 for validation and 1972 for testing. In both cases, the experiment uses a similar setup to the first binary classifier experiment, using `w2v_avg` averaged feature vector and a multi-layer perceptron network.

3.5. Summary

The first part of this chapter introduced methods of transferring words and contextual information into appropriate vector representations, so called *features*. The two established frameworks of *Word2Vec* and *BERT* were presented since they will be compared in this thesis. Four feature sets were created: `w2v_avg` are Word2Vec vectors that were averaged over all word vectors of each sample. The others try to preserve information about word sequences and context, containing a matrix (series of word vectors) for each code sample. `w2v_seq` is the Word2Vec variant of this. Because of technical limitations, only functions of up to 100 source code tokens were included; shorter samples were zero-padded to match in length. For BERT, two sets of feature embeddings were created: `bert128` and `bert64` contain samples with BERT sequences of lengths up to 128 and 64 respectively.

For mitigating the imbalance in the source data, three solutions were discussed: Randomly undersampling over-represented classes, oversampling others by generating additional samples through *Synthetic Minority Oversampling Technique (SMOTE)* as well as applying error weights inversely proportional to class sizes during the training process. Finally, section 3.3 described the three underlying models of basic feed-forward networks, long short-term memory models and the ResNet topology that were implemented in the scope of this thesis.

4. Results

In the last two chapters, the data sources, data processing as well as the deep learning techniques that will be used to build classifiers for the source code snippets in question were described and discussed. This chapter will compile and document all relevant results of the different experiments conducted within the scope of this thesis.

First, outcomes for the various binary classification techniques will be listed in section 4.1. Secondly, results for experiments that try to recognize clusters of common weakness types are demonstrated in section 4.2. In total, 64 different experiment variants were fine-tuned and documented within the scope of this thesis. While this chapter only includes those results that will be discussed later on, figures for all experiments can be found in appendices A.4 and A.5. The results presented here will be discussed in chapter 5 in order to deduct insights about which features or network architectures might generally better fit the task at hand.

4.1. Results of binary classification experiments

The following pages will show the results that were achieved with regards to vulnerability detection in the collected programs. In order to easily distinguish the individual models and model variants, a naming scheme was defined. For each combination of model, feature set and data used for training and evaluation, the experiments were named using specific keywords.

`MLP`, `LSTM` and `ResNet` refer to the underlying network architectures that were introduced in chapter 3. Usually the SMOTE variant of experiments was used for trying out different network parameters like number and size of hidden layers. The most promising topology was then used for the related experiments with weighted classes, undersampling or no imbalance mitigation at all. Other training parameters were in some cases slightly changed for other variants if they provided superior results in those cases. In some cases, more than one variant of the specific network structure is of interest. These are denoted by `Var1` and `Var2`.

`SMOTE`, `undersample` and `weights` refer to the corresponding techniques of imbalanced label mitigation as presented in 3.2. Additionally, some variants were added without

any attempt at compensating those imbalances, which is why they don't contain any of these keywords. As already mentioned before, classifier models were trained on two different feature embeddings described in 3.1.1 and 3.1.2. The two sets of Word2Vec vectors are called `w2v_avg` for vectors averaged per sample and `w2v_seq` for sequences of word vectors. For BERT sequences there are the two variants identified by `bert128` and `bert64`.

Finally, `testJTT` denotes experiments where training and validation were done on the combined `LOSARD102` data (80/20 split) and Juliet's Test Suite v1.3 served as a completely separate set of unseen data. These are supposed to give an idea of how the algorithm would perform in real-world applications. In all other cases, a 20% randomized share of `LOSARD102` was used as test examples.

The primary metrics used for comparing the prediction success of different binary classifiers were *Sensitivity* and *Specifity* [1]. Functions containing one or more security issues were treated as *positives* and the rest as *negatives*. That is because similarly to their use in medical diagnosis, these two terms "refer to the presence or absence of the condition of interest" [1], in this case vulnerability towards software exploits. While sensitivity defines which percentage of positive samples were detected as such in relation to all positives, specificity represents the share of correctly identified negatives in relation to all negatives in the base data. In tasks like this where detecting positives is of higher priority than avoiding *false positives* (safe samples falsely identified as positives), specifity is often compromised and more importance placed on higher sensitivity.

Further, in order to have a single measure for comparing the experiments, *Balanced Accuracy* was calculated as $\frac{Sensitivity}{2} + \frac{Specifity}{2}$. Because it doesn't take class imbalances into account, it is not susceptible to the issue of inflated accuracy numbers as described in section 3.2. This measure is indicated as `Accuracy` in the tables containing experiment results.

The results presented below will be interpreted and discussed in chapter 5 (section 5.1).

Model	Features	Train./Val.	Test	Imb. mit.	Accuracy	Sensitivity	Specifity
MLP (Var1)	w2v_avg	LOSARD102	LOSARD102	SMOTE	95.57	96.50	96.64
MLP (Var2)	w2v_avg	LOSARD102	LOSARD102	SMOTE	95.60	95.44	95.76
MLP	w2v_avg	LOSARD102	LOSARD102	None	91.15	82.87	99.43
MLP	w2v_avg	LOSARD102	LOSARD102	undersample	96.34	96.46	96.22
MLP	w2v_avg	LOSARD102	LOSARD102	Weights	96.92	96.94	96.89
LSTM	w2v_seq	LOSARD102	LOSARD102	None	99.44	98.92	99.96
LSTM	w2v_seq	LOSARD102	LOSARD102	undersample	99.27	99.15	99.39
LSTM	w2v_seq	LOSARD102	LOSARD102	SMOTE	99.44	99.00	99.88
LSTM	w2v_seq	LOSARD102	LOSARD102	Weights	98.88	97.77	99.99
LSTM	bert128	LOSARD102	LOSARD102	None	98.33	96.74	99.91
LSTM	bert128	LOSARD102	LOSARD102	undersample	98.33	96.74	99.91
LSTM (Var 1)	bert128	LOSARD102	LOSARD102	SMOTE	98.83	97.83	99.83
LSTM (Var 2)	bert128	LOSARD102	LOSARD102	SMOTE	98.83	99.84	97.83
LSTM	bert128	LOSARD102	LOSARD102	Weights	98.56	97.83	99.29
LSTM	bert64	LOSARD102	LOSARD102	None	100.00	100.00	100.00
LSTM	bert64	LOSARD102	LOSARD102	undersample	99.87	100.00	99.74
LSTM (Var 1)	bert64	LOSARD102	LOSARD102	SMOTE	99.96	100.00	99.92
LSTM (Var 2)	bert64	LOSARD102	LOSARD102	SMOTE	99.95	100.00	99.90
LSTM	bert64	LOSARD102	LOSARD102	Weights	99.92	100.00	99.85
ResNet	w2v_seq	LOSARD102	LOSARD102	None	99.26	99.08	99.44
ResNet	w2v_seq	LOSARD102	LOSARD102	undersample	99.50	99.23	99.77
ResNet	w2v_seq	LOSARD102	LOSARD102	SMOTE	52.05	4.93	99.18
ResNet	w2v_seq	LOSARD102	LOSARD102	Weights	99.07	99.54	98.60
ResNet	bert128	LOSARD102	LOSARD102	None	98.33	96.74	99.92
ResNet	bert128	LOSARD102	LOSARD102	undersample	50.00	0.00	100.00
ResNet	bert128	LOSARD102	LOSARD102	SMOTE	98.84	97.83	99.85
ResNet	bert128	LOSARD102	LOSARD102	Weights	98.17	97.83	98.52
ResNet	bert64	LOSARD102	LOSARD102	None	99.96	100.00	99.92
ResNet	bert64	LOSARD102	LOSARD102	undersample	99.96	100.00	99.92
ResNet	bert64	LOSARD102	LOSARD102	SMOTE	99.96	100.00	99.92
ResNet	bert64	LOSARD102	LOSARD102	Weights	99.46	100.00	98.93

Table 4.1.: Results of binary classifier evaluation on LOSARD102

Model	Features	Train./Val.	Test	Imb. mit.	Accuracy	Sensitivity	Specifity
MLP	w2v_avg	LOSARD102	JTT	SMOTE	62.57	91.55	33.60
MLP	w2v_avg	LOSARD102	JTT	undersample	59.11	83.57	34.65
LSTM	w2v_seq	LOSARD102	JTT	None	50.00	0.00	100.00
LSTM	w2v_seq	LOSARD102	JTT	undersample	50.93	6.08	95.77
LSTM	w2v_seq	LOSARD102	JTT	SMOTE	51.95	5.43	98.47
LSTM	w2v_seq	LOSARD102	JTT	Weights	49.56	0.16	98.95
LSTM	bert128	LOSARD102	JTT	undersample	52.65	6.45	98.85
ResNet	w2v_seq	LOSARD102	JTT	SMOTE	39.69	22.37	57.00
ResNet	bert128	LOSARD102	JTT	Weights	53.69	8.60	98.78

Table 4.2.: Notable results of binary classifier evaluation on JTT

Model	Features	Train./Val.	Test	Imb. mit.	Accuracy	Sensitivity	Specifity
MLP	w2v_avg	LOSARD102	LOSARD102	None	91.15	82.87	99.43
LSTM	w2v_seq	LOSARD102	LOSARD102	None	99.44	98.92	99.96
LSTM	bert128	LOSARD102	LOSARD102	None	98.33	96.74	99.91
LSTM	bert64	LOSARD102	LOSARD102	None	100.00	100.00	100.00
ResNet	w2v_seq	LOSARD102	LOSARD102	None	99.26	99.08	99.44
ResNet	bert128	LOSARD102	LOSARD102	None	98.33	96.74	99.92
ResNet	bert64	LOSARD102	LOSARD102	None	99.96	100.00	99.92
MLP	w2v_avg	LOSARD102	JTT	None	50.00	0.00	100.00
LSTM	w2v_seq	LOSARD102	JTT	None	50.00	0.00	100.00
LSTM	bert128	LOSARD102	JTT	None	50.00	0.00	100.00
LSTM	bert64	LOSARD102	JTT	None	50.00	0.00	100.00
ResNet	w2v_seq	LOSARD102	JTT	None	50.00	0.00	100.00
ResNet	bert128	LOSARD102	JTT	None	50.00	0.00	100.00
ResNet	bert64	LOSARD102	JTT	None	50.00	0.00	100.00

Table 4.3.: Results of experiments without imbalanced data processing

Model	Features	Train./Val.	Test	Imb. mit.	Accuracy	Sensitivity	Specifity
LSTM	bert64	LOSARD102	LOSARD102	None	100.00	100.00	100.00
LSTM	bert64	LOSARD102	LOSARD102	undersample	99.87	100.00	99.74
LSTM (Var 1)	bert64	LOSARD102	LOSARD102	SMOTE	99.96	100.00	99.92
LSTM (Var 2)	bert64	LOSARD102	LOSARD102	SMOTE	99.95	100.00	99.90
LSTM	bert64	LOSARD102	LOSARD102	Weights	99.92	100.00	99.85
ResNet	bert64	LOSARD102	LOSARD102	None	99.96	100.00	99.92
ResNet	bert64	LOSARD102	LOSARD102	undersample	99.96	100.00	99.92
ResNet	bert64	LOSARD102	LOSARD102	SMOTE	99.96	100.00	99.92
ResNet	bert64	LOSARD102	LOSARD102	Weights	99.46	100.00	98.93
LSTM	bert64	LOSARD102	JTT	None	50.00	0.00	100.00
LSTM	bert64	LOSARD102	JTT	undersample	50.00	0.00	100.00
LSTM	bert64	LOSARD102	JTT	SMOTE	50.00	0.00	100.00
LSTM	bert64	LOSARD102	JTT	Weights	50.00	0.00	100.00
ResNet	bert64	LOSARD102	JTT	None	50.00	0.00	100.00
ResNet	bert64	LOSARD102	JTT	undersample	50.00	0.00	100.00
ResNet	bert64	LOSARD102	JTT	SMOTE	50.00	0.00	100.00
ResNet	bert64	LOSARD102	JTT	Weights	50.00	0.00	100.00

Table 4.4.: Results of experiments using the bert64 feature set

4.2. Results of vulnerability classification experiments

The second goal was to create deep learning models that are able to identify which kinds of exploits a bad code snippet might contain. A subset of the achieved results are documented below. The multi-class prediction results will be summarized and discussed in 5.2

For all CWE classifier experiments, class weights were used to deal with different class sizes within the data. Since they work on the `w2v_avg` single vectors, the input layer always contains 100 neurons. Additionally, a Dropout layer was added to make the models less prone to overfitting. The error on both the training and validation set was recorded over the course of the training.

Model	Train./Val.	Test	Precision	Recall
MLP Var1	LOSARD102	LOSARD102	80.89	56.28
MLP Var2	LOSARD102	LOSARD102	85.39	57.93
MLP Var1	LOSARD102 (distinct)	LOSARD102 (distinct)	75.45	89.00
MLP Var2	LOSARD102 (distinct)	LOSARD102 (distinct)	78.07	90.22

Table 4.5.: Metrics comparison of CWE classifier models on total and distinct dataset

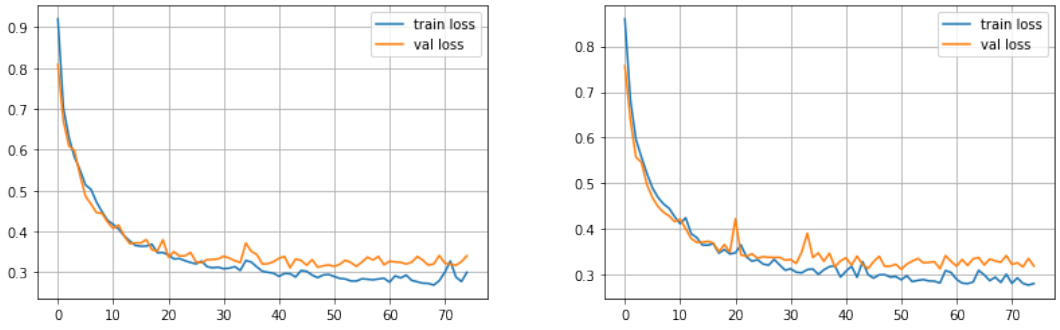
4.2.1. Multi-label classification of positive code examples

The first variant features a two hidden layers of size 100, the same size as the input vector. It used a batch size of 256 and learned for 75 epochs. A slightly adjusted topology *Variant 2* has three hidden layers of 100, 64 and 32 nodes respectively. Apart from that, it was trained on a smaller batch size of 128 instead of 256.

The essential prediction metrics are listed below. Figure 4.1 visualizes the networks' learning process through training and validation error curves while figure 4.2 compares the confusion matrices for these two experiments.

Metrics for Variant 1

	precision	recall	f1-score	support
CWE-664	0.9989	0.7221	0.8383	2634
CWE-682	0.5132	0.3362	0.4062	116
CWE-691	0.9405	0.4566	0.6148	173
CWE-693	0.9845	0.8900	0.9349	2291
CWE-697	0.3333	0.2812	0.3051	32
CWE-707	0.9883	0.8369	0.9063	2716



(a) Variant 1 (b) Variant 2

Figure 4.1.: CWE MLP - Training and validation error over epochs

CWE-710	0.9032	0.4164	0.5700	269
micro avg	0.9798	0.7840	0.8710	8231
macro avg	0.8089	0.5628	0.6536	8231
weighted avg	0.9776	0.7840	0.8660	8231
samples avg	0.7655	0.7547	0.7566	8231

Metrics for Variant 2

	precision	recall	f1-score	support
CWE-664	0.9949	0.6602	0.7937	2634
CWE-682	0.5676	0.3621	0.4421	116
CWE-691	0.9615	0.4335	0.5976	173
CWE-693	0.9827	0.9158	0.9480	2291
CWE-697	0.7500	0.5625	0.6429	32
CWE-707	0.9813	0.8682	0.9213	2716
CWE-710	0.7391	0.2528	0.3767	269
micro avg	0.9762	0.7773	0.8655	8231
macro avg	0.8539	0.5793	0.6746	8231
weighted avg	0.9709	0.7773	0.8555	8231
samples avg	0.7438	0.7367	0.7372	8231

4.2.2. Classification of distinct CWE clusters

The model structures used for the distinct data set are largely identical to those described in the previous section. *Variant 1* again contains two hidden layers with a hidden size of 100 neurons while the second alternative uses more hidden layers that get smaller with increasing model depth and a smaller batch size. This is supposed to help isolate the performance differences between overlapping and discrete clusters in the data set.

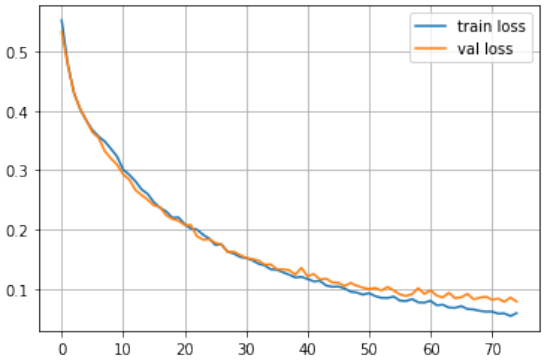
Apart from the metrics plots during training, figure 4.2 also contains the confusion matrices for the two variants. The rows and columns are named after the CWE-IDs of the respective *Research Concept*. While the rows represent the actual labels of samples, the columns refer to the network's prediction and the values within the matrix show the absolute number of predicted samples for each combination. For a perfect classifier (meaning no prediction error), all fields not on the main diagonal would contain zeros.

Metrics for Variant 1

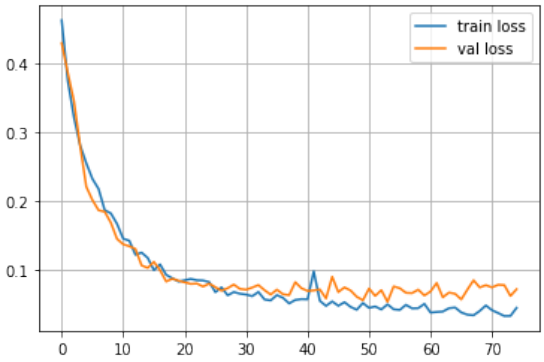
	precision	recall	f1-score	support
CWE-664	0.9922	0.8783	0.9318	1454
CWE-682	0.6350	0.9775	0.7699	89
CWE-691	0.8158	0.9208	0.8651	101
CWE-693	1.0000	0.9091	0.9524	11
CWE-697	0.1606	0.9565	0.2750	23
CWE-707	1.0000	0.6790	0.8088	162
CWE-710	0.6780	0.9091	0.7767	132
micro avg	0.8717	0.8717	0.8717	1972
macro avg	0.7545	0.8900	0.7685	1972
weighted avg	0.9370	0.8717	0.8930	1972
samples avg	0.8717	0.8717	0.8717	1972

Metrics for Variant 2

	precision	recall	f1-score	support
CWE-664	0.9924	0.9017	0.9449	1454
CWE-682	0.4265	0.9775	0.5939	89
CWE-691	0.8230	0.9208	0.8692	101
CWE-693	0.9091	0.9091	0.9091	11
CWE-697	0.5128	0.8696	0.6452	23
CWE-707	1.0000	0.7901	0.8828	162
CWE-710	0.8013	0.9470	0.8681	132
micro avg	0.8996	0.8996	0.8996	1972
macro avg	0.7807	0.9022	0.8161	1972
weighted avg	0.9400	0.8996	0.9112	1972
samples avg	0.8996	0.8996	0.8996	1972



(a) Variant 1 - Loss



(b) Variant 2 - Loss

	664	682	691	693	697	707	710
664	1277	35	19	0	95	0	28
682	0	87	0	0	1	0	1
691	1	1	93	0	4	0	2
693	1	0	0	10	0	0	0
697	0	0	1	0	22	0	0
707	7	11	0	0	8	110	26
710	1	3	1	0	7	0	120

(c) Variant 1 - Confusion Matrix

	664	682	691	693	697	707	710
664	1311	94	20	0	10	0	19
682	0	87	0	0	1	0	1
691	1	3	93	0	4	0	0
693	1	0	0	10	0	0	0
697	0	3	0	0	20	0	0
707	8	12	0	1	2	128	11
710	0	5	0	0	2	0	125

(d) Variant 2 - Confusion Matrix

Figure 4.2.: CWE distinct MLP - Loss, Accuracy plots and confusion matrices

5. Discussion

Chapter 4 listed the results for a subset of experiments implemented for this thesis. The goal of this chapter now is to use those recorded metrics to try and draw assumptions regarding strengths and limitations of the models in question, but also which factors might contribute to better classification capability.

5.1. Detection of vulnerable functions in program code

The experiments to be discussed differ in three aspects: Underlying model architecture, method of feature embedding and different ways of dealing with imbalanced labels in training data. First, the influence of architecture and features on detection performance will be discussed. After that, patterns and differences in learning success between the techniques of class imbalance mitigation will be discussed.

The first set of experiments evaluated the models' ability to identify vulnerable functions in a small random share of the `LOSARD102` functions. In this scenario, all architectures managed to detect over 90% of bad samples in the test data in some or all tested variants (see table 4.1). The only exceptions were `ResNet_w2v_seq_SMOTE` and `ResNet_bert128_undersample` which identified almost no vulnerabilities. These outlier results might have been possible to resolve if a bit more time could have been dedicated to model and parameter tuning. Transferring the patterns learned within the LOSARD102 data onto completely new code bases such as the Juliet Test Suite proved to be a lot more challenging as will be discussed in the following paragraphs.

Surprisingly, one of the more basic models - the feed-forward MLP trained on averaged word2vec vectors - achieved the highest sensitivity on the Juliet Test Suite by a large margin (see table 4.2), scoring as high as 91.55% on SMOTE-augmented training data. The MLP model trained on the smaller training set created by undersampling safe samples achieves the second best result, detecting 83.57% of unsafe functions in the JTT data set. Both of these models compromise in order to get to these results, only reaching an average specifity of 34.13%.

Experiments using LSTM and ResNet models do not come close to the results of the feed-forward network. In the case of `testJTT_LSTM_w2v_seq` experiments (table 4.2), two out of four experiments fail to show any detection capability. The undersampling and SMOTE variants achieve detection rates on JTT in the single digits (5.43% - 6.08%). ResNet trained on `w2v_seq` is an exception, scoring a bit higher at 22.37% sensitivity. It seems as if either the additional information of features based on word sequences (`w2v_seq`, `bert128` and `bert64`) or more sophisticated network architectures impede their ability to compromise on specificity in favor of higher sensitivity.

Despite their excellent performance on split training and test data (100% sensitivity and very high accuracy in all cases), experiments using `bert64` (table 4.4) fail to predict any vulnerable samples in JTT at all. The networks perform well on inputs that are somewhat similar to their training data, but are not able to transfer that knowledge onto foreign test data from other code bases. This poor generalization performance might have been caused by the smaller number of training samples and, more importantly, the shorter sequences simply not containing enough information compared to `bert128`. Unfortunately, within the scope of this thesis it was not feasible to try out longer BERT word sequences with lengths of e. g., 256 or more tokens because of time constraints, but also due to technical limitations such as system memory and storage space. The potential of including BERT sequences of longer source code snippets might present an interesting question for following works to look into.

When predicting samples from the combined LOSARD102 data set, all methods of addressing class imbalance perform similarly well. Overall, applying weights to the error calculation proved the most consistent, achieving an average sensitivity score of 98,56% (see table 4.1). It avoids the computational effort associated with SMOTE-based oversampling and preserves all training examples while often outperforming the approach of not addressing imbalances at all. In cases where training and test programs share similar properties, this approach is therefore recommended.

In the *real world* tests using the Juliet Test Suite, none of the three techniques for balancing out training data consistently scored better than their alternatives. However, models trained on SMOTE-enhanced data sets performed best in two cases and competitively in one other case. SMOTE won on `testJTT_MLP_w2v_avg` as well as `testJTT_ResNet_w2v_seq` (table 4.2), scoring highest in sensitivity (91.55% and 22.37%) while maintaining high, but reasonable specifity (33.60% and 57.00% respectively). Furthermore, it enabled `testJTT_LSTM_w2v_seq` to detect 5.43% of vulnerable samples, within reach of the undersampling variant which performed best in that group (6.08%) (see table 4.2). The SMOTE variants seemed to be better at finding a balance between sensitivity and false positive rates that fits a task like this where false negatives are highly undesirable. Models trained using other techniques were much more susceptible to

putting everything into either one of the two classes. Finally, the weighted errors and undersampling approach each won in exactly one experiment (`testJTT_ResNet_bert128` and `testJTT_LSTM_bert128` respectively). Interestingly, in both of those cases they were the only variant that achieved any meaningful learning effect at all.

In addition to the three different alternatives for balancing out biased ground truth in training data, the binary classifiers were also evaluated without any of these techniques (table 4.3). When testing prediction metrics on a subset of LOSARD102, this variant noticeably worse for the MLP models but competitively for all other experiments. However, it completely failed when the classifiers were tested on the separate Juliet Test Suite: Regardless of the model architecture, it immediately went to classifying everything as *good* when no countermeasures for biased class representation were applied. This was true for all combinations of parameters and modifications to the network design that were implemented. One possible explanation could lie in the different ratios of bad samples between LOSARD102 and JTT: While approximately 7.86% of LOSARD102 are affected by known security weaknesses, this is only true for 3.31% of programs in JTT.

5.2. Recognition of vulnerability types

Despite the complex nature of multi-class multi-label classification, the experiments on all positive samples still consistently managed to learn patterns for different weakness types. The MLP classifier model that was implemented and trained on all vulnerable samples in `LOSARD` achieved an average recall across the seven clusters of up to 57.93% (see table 4.5). *Recall* refers to the percentage of samples per cluster that were correctly identified. Additionally, this discussion will use the balanced average of these seven recall figures for comparison.

For the second group of experiments, only samples that belong to one single Research Concept were included, which led to significantly better classification success. They take longer to learn than their multi-label counterpart (see figures 4.2 and 4.1) but converge on a lower total prediction error, also known as the *loss*. This is true even more so for underrepresented clusters such as *CWE-697* or *CWE-693*. When looking into the two model varations that were configured, the prediction metrics indicate significantly better performance for *Variant 2* with smaller batches and smaller hidden layers. This topology achieves the best results on discrete clusters with an average recall of 90.22% (see table 4.5). It can be deduced that the classification of vulnerability types through use of deep learning scored considerably higher on samples that belong to single CWE groups.

6. Conclusion

This thesis introduced a basic feed-forward network trained on Word2Vec-based features that was able to detect over 95% of faulty C/C++ functions in a representative extract of *Juliet Test Suite v1.3* which is part of the Software Assurance Reference Dataset provided by NIST.

Mitigation of major imbalances in the training data turned out to be crucial for successful detection of learned patterns in other programs. Oversampling training samples via SMOTE provides adequate results for a wide variety of network architectures and feature embeddings.

When confronted with unseen programs from the same code base as the training data, almost all topologies and techniques of feature generation provided excellent results. This however does not accurately represent real world applications of static code analysis tools. Classification specifity on new code bases (with possibly different kinds of vulnerability types) is mediocre so far. The success in vulnerability detection by the MLP feed-forward model comes at the expense of a significantly lower accuracy. About two thirds of safe program functions were falsely labeled as flawed. This means that a hypothetical static code analysis tool based on this classifier would produce lots of false warnings and thus cause more work for developers and testers in verifying them.

One of the major challenges when working on this topic turned out to be the sourcing of suitable training data. High-quality source code collections along with detailed information about included security flaws are few and far between. The big difference in classification success between programs from the same code base versus others might be caused by different kinds of security weaknesses in the two data sets. A more varied training set that is compiled from many different projects and contains many different types of vulnerabilities might help improve prediction accuracy on new programs. However, creating such a rich collection of C/C++ programs requires a lot of manual work from software security experts.

Another central aspect that further works could focus on is generating and training on longer BERT vector series, which wasn't feasible within the time and resource constraints of this thesis. Models trained on BERT features of up to 64 words failed at detecting vulnerabilities in unseen data sets, while those including up to 128 words exhibited at least some detection capabilities. This indicates that even longer sample sizes might lead to better recognition sensitivity and accuracy.

The second question this thesis asked was whether deep learning could be used to classify commonly known types of security flaws. A feed-forward model trained on all unsafe programs in the combined LibreOffice and SARD-102 code base achieved a balanced overall recall of up to 57.93%. Modified versions of these experiments showed that this classification performs a lot better on weaknesses that can be linked to a single cluster. Those variants correctly classified up to 90.22% of samples into their correct cluster. This leads to the conclusion that in order to achieve better vulnerability type classification, it is advisable to use training examples that represent single, distinct weakness types.

As stated in some related works that were briefly presented in chapter 1, detection of software security weaknesses using deep learning techniques shows promising results in some cases. However, it still has a lot of research to go into it can be considered for productive use.

A. Supplemental Information

A.1. Source code of implemented programs

All programs developed for this thesis are available at https://github.com/ralphscheuerer/ba-nn-vdisc. The saved models are downloadable separately at https://github.com/ralphscheuerer/ba-nn-vdisc/releases.

A.2. Exemplary output of the DumpTokens step

```
identifier 'sal_Int16'    [StartOfLine]  Loc=<<stdin>:3:1>
identifier 'SAL_CALL'     [LeadingSpace] Loc=<<stdin>:3:11>
identifier 'AnimationNode'        [LeadingSpace] Loc=<<stdin>:3:20>
coloncolon '::'           Loc=<<stdin>:3:33>
identifier 'getFill'              Loc=<<stdin>:3:35>
l_paren '('               Loc=<<stdin>:3:42>
r_paren ')'               Loc=<<stdin>:3:43>
l_brace '{'       [StartOfLine]  Loc=<<stdin>:4:1>
identifier 'Guard'        [StartOfLine] [LeadingSpace] Loc=<<stdin>:5:5>
less '<'                  Loc=<<stdin>:5:10>
identifier 'Mutex'        [LeadingSpace] Loc=<<stdin>:5:12>
greater '>'       [LeadingSpace] Loc=<<stdin>:5:18>
identifier 'aGuard'       [LeadingSpace] Loc=<<stdin>:5:20>
l_paren '('               Loc=<<stdin>:5:26>
identifier 'maMutex'      [LeadingSpace] Loc=<<stdin>:5:28>
r_paren ')'       [LeadingSpace] Loc=<<stdin>:5:36>
semi ';'                  Loc=<<stdin>:5:37>
return 'return'  [StartOfLine] [LeadingSpace]    Loc=<<stdin>:6:5>
identifier 'mnFill'       [LeadingSpace] Loc=<<stdin>:6:12>
semi ';'                  Loc=<<stdin>:6:18>
r_brace '}'       [StartOfLine]  Loc=<<stdin>:7:1>
void 'void'       [StartOfLine]  Loc=<<stdin>:11:1>
identifier 'SAL_CALL'     [LeadingSpace] Loc=<<stdin>:11:6>
identifier 'AnimationNode'        [LeadingSpace] Loc=<<stdin>:11:15>
coloncolon '::'           Loc=<<stdin>:11:28>
identifier 'setFill'              Loc=<<stdin>:11:30>
l_paren '('               Loc=<<stdin>:11:37>
identifier 'sal_Int16'    [LeadingSpace] Loc=<<stdin>:11:39>
identifier '_fill'        [LeadingSpace] Loc=<<stdin>:11:49>
r_paren ')'       [LeadingSpace] Loc=<<stdin>:11:55>
l_brace '{'       [StartOfLine]  Loc=<<stdin>:12:1>
identifier 'Guard'        [StartOfLine] [LeadingSpace] Loc=<<stdin>:13:5>
less '<'                  Loc=<<stdin>:13:10>
identifier 'Mutex'        [LeadingSpace] Loc=<<stdin>:13:12>
```

```
greater '>'       [LeadingSpace] Loc=<<stdin>:13:18>
identifier 'aGuard'       [LeadingSpace] Loc=<<stdin>:13:20>
l_paren '('               Loc=<<stdin>:13:26>
identifier 'maMutex'      [LeadingSpace] Loc=<<stdin>:13:28>
r_paren ')'       [LeadingSpace] Loc=<<stdin>:13:36>
semi ';'                  Loc=<<stdin>:13:37>
if 'if'  [StartOfLine] [LeadingSpace]   Loc=<<stdin>:14:5>
l_paren '('               Loc=<<stdin>:14:7>
identifier '_fill'        [LeadingSpace] Loc=<<stdin>:14:9>
exclaimequal '!='         [LeadingSpace] Loc=<<stdin>:14:15>
identifier 'mnFill'       [LeadingSpace] Loc=<<stdin>:14:18>
r_paren ')'       [LeadingSpace] Loc=<<stdin>:14:25>
l_brace '{'       [StartOfLine] [LeadingSpace]    Loc=<<stdin>:15:5>
identifier 'mnFill'  [StartOfLine] [LeadingSpace] Loc=<<stdin>:16:9>
equal '='         [LeadingSpace] Loc=<<stdin>:16:16>
identifier '_fill'        [LeadingSpace] Loc=<<stdin>:16:18>
semi ';'                  Loc=<<stdin>:16:23>
identifier 'fireChangeListener' [StartOfLine] [LeadingSpace] Loc=<<stdin>:17:9>
l_paren '('               Loc=<<stdin>:17:27>
r_paren ')'               Loc=<<stdin>:17:28>
semi ';'                  Loc=<<stdin>:17:29>
r_brace '}'       [StartOfLine] [LeadingSpace]    Loc=<<stdin>:18:5>
r_brace '}'       [StartOfLine]  Loc=<<stdin>:19:1>
eof ''          Loc=<<stdin>:19:2>
```

Listing A.1: Complete output of clang -dump-tokens

A.3. Flawfinder 2.0.11 rules

```
Flawfinder version 2.0.11, (C) 2001-2019 David A. Wheeler.
Number of rules (primarily dangerous function names) in C/C++ ruleset: 223
AddAccessAllowedAce  3  This doesn't set the inheritance bits in the access
    ↪ control entry (ACE) header (CWE-732)
CoImpersonateClient  4  If this call fails, the program could fail to drop
    ↪ heightened privileges (CWE-250)
CopyMemory  2  Does not check for buffer overflows when copying to destination (
    ↪ CWE-120)
CreateProcess  3  This causes a new process to execute and is difficult to use
    ↪ safely (CWE-78)
CreateProcessAsUser  3  This causes a new process to execute and is difficult to
    ↪ use safely (CWE-78)
CreateProcessWithLogon  3  This causes a new process to execute and is difficult
    ↪ to use safely (CWE-78)
EVP_des_cbc 4  DES only supports a 56-bit keysize, which is too small given today
    ↪ 's computers (CWE-327)
EVP_des_cfb 4  DES only supports a 56-bit keysize, which is too small given today
    ↪ 's computers (CWE-327)
EVP_des_ecb 4  DES only supports a 56-bit keysize, which is too small given today
    ↪ 's computers (CWE-327)
EVP_des_ofb 4  DES only supports a 56-bit keysize, which is too small given today
    ↪ 's computers (CWE-327)
EVP_desx_cbc  4  DES only supports a 56-bit keysize, which is too small given
    ↪ today's computers (CWE-327)
EVP_rc2_40_cbc 4  These keysizes are too small given today's computers (CWE-327)
```

```
EVP_rc2_64_cbc 4 These keysizes are too small given today's computers (CWE-327)
EVP_rc4_40 4 These keysizes are too small given today's computers (CWE-327)
EnterCriticalSection 3 On some versions of Windows, exceptions can be thrown in
    ↪ low-memory situations
GetTempFileName 3 Temporary file race condition in certain cases (e.g., if run
    ↪ as SYSTEM in many versions of Windows) (CWE-377)
ImpersonateDdeClientWindow 4 If this call fails, the program could fail to drop
    ↪ heightened privileges (CWE-250)
ImpersonateLoggedOnUser 4 If this call fails, the program could fail to drop
    ↪ heightened privileges (CWE-250)
ImpersonateNamedPipeClient 4 If this call fails, the program could fail to drop
    ↪ heightened privileges (CWE-250)
ImpersonateSecurityContext 4 If this call fails, the program could fail to drop
    ↪ heightened privileges (CWE-250)
InitializeCriticalSection 3 Exceptions can be thrown in low-memory situations
LoadLibrary 3 Ensure that the full path to the library is specified, or current
    ↪ directory may be used (CWE-829, CWE-20)
LoadLibraryEx 3 Ensure that the full path to the library is specified, or
    ↪ current directory may be used (CWE-829, CWE-20)
MultiByteToWideChar 2 Requires maximum length in CHARACTERS, not bytes (CWE
    ↪ -120)
RpcImpersonateClient 4 If this call fails, the program could fail to drop
    ↪ heightened privileges (CWE-250)
SetSecurityDescriptorDacl 5 Never create NULL ACLs; an attacker can set it to
    ↪ Everyone (Deny All Access), which would even forbid administrator access (
    ↪ CWE-732)
SetThreadToken 4 If this call fails, the program could fail to drop heightened
    ↪ privileges (CWE-250)
ShellExecute 4 This causes a new program to execute and is difficult to use
    ↪ safely (CWE-78)
StrCat 4 Does not check for buffer overflows when concatenating to destination
    ↪ [MS-banned] (CWE-120)
StrCatA 4 Does not check for buffer overflows when concatenating to destination
    ↪ [MS-banned] (CWE-120)
StrCatBuffA 4 Does not check for buffer overflows when concatenating to
    ↪ destination [MS-banned] (CWE-120)
StrCatBuffW 4 Does not check for buffer overflows when concatenating to
    ↪ destination [MS-banned] (CWE-120)
StrCatChainW 4 Does not check for buffer overflows when concatenating to
    ↪ destination [MS-banned] (CWE-120)
StrCatN 4 Does not check for buffer overflows when concatenating to destination
    ↪ [MS-banned] (CWE-120)
StrCatNA 4 Does not check for buffer overflows when concatenating to destination
    ↪ [MS-banned] (CWE-120)
StrCatNW 4 Does not check for buffer overflows when concatenating to destination
    ↪ [MS-banned] (CWE-120)
StrCpy 4 Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
StrCpyA 4 Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
StrCpyN 4 Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
StrCpyNA 4 Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
StrCpyNW 4 Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
```

```
StrNCat  4  Does not check for buffer overflows when concatenating to destination
    ↪  [MS-banned] (CWE-120)
StrNCatA 4  Does not check for buffer overflows when concatenating to destination
    ↪  [MS-banned] (CWE-120)
StrNCatW 4  Does not check for buffer overflows when concatenating to destination
    ↪  [MS-banned] (CWE-120)
StrNCpy  4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
StrNCpyA 4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
StrNCpyW 4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
StrcatW  4  Does not check for buffer overflows when concatenating to destination
    ↪  [MS-banned] (CWE-120)
TCHAR 2  Statically-sized arrays can be improperly restricted, leading to
    ↪ potential overflows or other issues (CWE-119!/CWE-120)
WinExec  4  This causes a new program to execute and is difficult to use safely (
    ↪ CWE-78)
_ftcscat 4  Does not check for buffer overflows when concatenating to destination
    ↪  [MS-banned] (CWE-120)
_ftcscpy 4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
_ftprintf  4  If format strings can be influenced by an attacker, they can be
    ↪ exploited (CWE-134)
_ftscanf 4  The scanf() family's %s operation, without a limit specification,
    ↪ permits buffer overflows (CWE-120, CWE-20)
_gettc  1  Check buffer boundaries if used in a loop including recursive loops (
    ↪ CWE-120, CWE-20)
_getts  5  Does not check for buffer overflows (CWE-120, CWE-20)
_mbccat  4  Does not check for buffer overflows when concatenating to destination
    ↪  [MS-banned] (CWE-120)
_mbccpy  4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
_mbscat  4  Does not check for buffer overflows when concatenating to destination
    ↪  [MS-banned] (CWE-120)
_mbscpy  4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
_mbslen  1  Does not handle strings that are not \0-terminated; if given one it
    ↪ may perform an over-read (it could cause a crash if unprotected) (CWE-126)
_mbsnbcat  1  Easily used incorrectly (e.g., incorrectly computing the correct
    ↪ maximum size to add) [MS-banned] (CWE-120)
_mbsnbcpy  1  Easily used incorrectly; doesn't always \0-terminate or check for
    ↪ invalid pointers [MS-banned] (CWE-120)
_mbsncpy 4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
_snprintf  4  If format strings can be influenced by an attacker, they can be
    ↪ exploited, and note that sprintf variations do not always \0-terminate (CWE
    ↪ -134)
_sntprintf  4  If format strings can be influenced by an attacker, they can be
    ↪ exploited, and note that sprintf variations do not always \0-terminate (CWE
    ↪ -134)
_stprintf  4  Does not check for buffer overflows (CWE-120)
_tccat  4  Does not check for buffer overflows when concatenating to destination
    ↪  [MS-banned] (CWE-120)
_tccpy  4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
```

```
_tcscat  4  Does not check for buffer overflows when concatenating to destination
    ↪  [MS-banned] (CWE-120)
_tcscpy  4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
_tcslen  1  Does not handle strings that are not \0-terminated; if given one it
    ↪ may perform an over-read (it could cause a crash if unprotected) (CWE-126)
_tcsncat 1  Easily used incorrectly (e.g., incorrectly computing the correct
    ↪ maximum size to add) [MS-banned] (CWE-120)
_tcsncpy 1  Easily used incorrectly; doesn't always \0-terminate or check for
    ↪ invalid pointers [MS-banned] (CWE-120)
_tscanf  4  The scanf() family's %s operation, without a limit specification,
    ↪ permits buffer overflows (CWE-120, CWE-20)
_vftprintf  4  If format strings can be influenced by an attacker, they can be
    ↪ exploited (CWE-134)
_vsntprintf 4  If format strings can be influenced by an attacker, they can be
    ↪ exploited, and note that sprintf variations do not always \0-terminate (CWE
    ↪ -134)
_vstprintf  4  Does not check for buffer overflows (CWE-120)
_vtprintf   4  If format strings can be influenced by an attacker, they can be
    ↪ exploited (CWE-134)
_wtoi 2  Unless checked, the resulting number can exceed the expected range (CWE
    ↪ -190)
_wtoi64  2  Unless checked, the resulting number can exceed the expected range (
    ↪ CWE-190)
access   4  This usually indicates a security flaw.  If an attacker can change
    ↪ anything along the path between the call to access() and the file's actual
    ↪ use (e.g., by moving files), the attacker can exploit the race condition (
    ↪ CWE-362/CWE-367!)
atoi  2  Unless checked, the resulting number can exceed the expected range (CWE
    ↪ -190)
atol  2  Unless checked, the resulting number can exceed the expected range (CWE
    ↪ -190)
bcopy 2  Does not check for buffer overflows when copying to destination (CWE
    ↪ -120)
char  2  Statically-sized arrays can be improperly restricted, leading to
    ↪ potential overflows or other issues (CWE-119!/CWE-120)
chgrp 5  This accepts filename arguments; if an attacker can move those files, a
    ↪ race condition results. (CWE-362)
chmod 5  This accepts filename arguments; if an attacker can move those files, a
    ↪ race condition results. (CWE-362)
chown 5  This accepts filename arguments; if an attacker can move those files, a
    ↪ race condition results. (CWE-362)
chroot   3  chroot can be very helpful, but is hard to use correctly (CWE-250,
    ↪ CWE-22)
crypt 4  The crypt functions use a poor one-way hashing algorithm; since they
    ↪ only accept passwords of 8 characters or fewer and only a two-byte salt,
    ↪ they are excessively vulnerable to dictionary attacks given today's faster
    ↪ computing equipment (CWE-327)
crypt_r  4  The crypt functions use a poor one-way hashing algorithm; since they
    ↪ only accept passwords of 8 characters or fewer and only a two-byte salt,
    ↪ they are excessively vulnerable to dictionary attacks given today's faster
    ↪ computing equipment (CWE-327)
curl_getenv 3  Environment variables are untrustable input if they can be set by
    ↪ an attacker.  They can have any content and length, and the same variable
    ↪ can be set more than once (CWE-807, CWE-20)
cuserid  4  Exactly what cuserid() does is poorly defined (e.g., some systems use
    ↪  the effective uid, like Linux, while others like System V use the real uid
```

```
    ↪ ). Thus, you can't trust what it does. It's certainly not portable (The
    ↪ cuserid function was included in the 1988 version of POSIX, but removed
    ↪ from the 1990 version). Also, if passed a non-null parameter, there's a
    ↪ risk of a buffer overflow if the passed-in buffer is not at least L_cuserid
    ↪  characters long (CWE-120)
drand48 3 This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
equal 1 Function does not check the second iterator for over-read conditions (
    ↪ CWE-126)
erand48 3 This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
execl 4 This causes a new program to execute and is difficult to use safely (CWE
    ↪ -78)
execle 4 This causes a new program to execute and is difficult to use safely (
    ↪ CWE-78)
execlp 4 This causes a new program to execute and is difficult to use safely (
    ↪ CWE-78)
execv 4 This causes a new program to execute and is difficult to use safely (CWE
    ↪ -78)
execvp 4 This causes a new program to execute and is difficult to use safely (
    ↪ CWE-78)
fgetc 1 Check buffer boundaries if used in a loop including recursive loops (CWE
    ↪ -120, CWE-20)
fopen 2 Check when opening files - can an attacker redirect it (via symlinks),
    ↪ force the opening of special file type (e.g., device files), move things
    ↪ around to create a race condition, control its ancestors, or change its
    ↪ contents? (CWE-362)
fprintf 4 If format strings can be influenced by an attacker, they can be
    ↪ exploited (CWE-134)
fread 0 Function accepts input from outside program (CWE-20)
fscanf 4 The scanf() family's %s operation, without a limit specification,
    ↪ permits buffer overflows (CWE-120, CWE-20)
fvwprintf 4 If format strings can be influenced by an attacker, they can be
    ↪ exploited (CWE-134)
fwprintf 4 If format strings can be influenced by an attacker, they can be
    ↪ exploited (CWE-134)
fwscanf 4 The scanf() family's %s operation, without a limit specification,
    ↪ permits buffer overflows (CWE-120, CWE-20)
g_get_home_dir 3 This function is synonymous with 'getenv("HOME");it returns
    ↪ untrustable input if the environment can beset by an attacker. It can have
    ↪  any content and length, and the same variable can be set more than once (
    ↪ CWE-807, CWE-20)
g_get_tmp_dir 3 This function is synonymous with 'getenv("TMP");it returns
    ↪ untrustable input if the environment can beset by an attacker. It can have
    ↪  any content and length, and the same variable can be set more than once (
    ↪ CWE-807, CWE-20)
g_rand_boolean 3 This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
g_rand_double 3 This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
g_rand_double_range 3 This function is not sufficiently random for security-
    ↪ related functions such as key and nonce creation (CWE-327)
g_rand_int 3 This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
g_rand_int_range 3 This function is not sufficiently random for security-
    ↪ related functions such as key and nonce creation (CWE-327)
```

```
g_random_boolean  3  This function is not sufficiently random for security-
    ↪ related functions such as key and nonce creation (CWE-327)
g_random_double  3  This function is not sufficiently random for security-
    ↪ related functions such as key and nonce creation (CWE-327)
g_random_double_range  3  This function is not sufficiently random for security-
    ↪ related functions such as key and nonce creation (CWE-327)
g_random_int  3  This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
g_random_int_range  3  This function is not sufficiently random for security-
    ↪ related functions such as key and nonce creation (CWE-327)
getc  1  Check buffer boundaries if used in a loop including recursive loops (CWE
    ↪ -120, CWE-20)
getchar  1  Check buffer boundaries if used in a loop including recursive loops (
    ↪ CWE-120, CWE-20)
getenv  3  Environment variables are untrustable input if they can be set by an
    ↪ attacker. They can have any content and length, and the same variable can
    ↪ be set more than once (CWE-807, CWE-20)
getlogin 4  It's often easy to fool getlogin. Sometimes it does not work at all,
    ↪  because some program messed up the utmp file. Often, it gives only the
    ↪ first 8 characters of the login name. The user currently logged in on the
    ↪ controlling tty of our program need not be the user who started it. Avoid
    ↪ getlogin() for security-related purposes (CWE-807)
getopt  3  Some older implementations do not protect against internal buffer
    ↪ overflows (CWE-120, CWE-20)
getopt_long 3  Some older implementations do not protect against internal buffer
    ↪ overflows (CWE-120, CWE-20)
getpass  4  This function is obsolete and not portable. It was in SUSv2 but
    ↪ removed by POSIX.2. What it does exactly varies considerably between
    ↪ systems, particularly in where its prompt is displayed and where it gets
    ↪ its data (e.g., /dev/tty, stdin, stderr, etc.). In addition, some
    ↪ implementations overflow buffers. (CWE-676, CWE-120, CWE-20)
getpw 4  This function is dangerous; it may overflow the provided buffer. It
    ↪ extracts data from a 'protected' area, but most systems have many commands
    ↪ to let users modify the protected area, and it's not always clear what
    ↪ their limits are. Best to avoid using this function altogether (CWE-676,
    ↪ CWE-120)
gets  5  Does not check for buffer overflows (CWE-120, CWE-20)
getwd 3  This does not protect against buffer overflows by itself, so use with
    ↪ caution (CWE-120, CWE-20)
gsignal  2  These functions are considered obsolete on most systems, and very non
    ↪ -portable (Linux-based systems handle them radically different, basically
    ↪ if gsignal/ssignal were the same as raise/signal respectively, while System
    ↪  V considers them a separate set and obsolete) (CWE-676)
is_permutation 1  Function does not check the second iterator for over-read
    ↪ conditions (CWE-126)
jrand48  3  This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
lcong48  3  This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
lrand48  3  This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
lstrcat  4  Does not check for buffer overflows when concatenating to destination
    ↪  [MS-banned] (CWE-120)
lstrcatA 4  Does not check for buffer overflows when concatenating to destination
    ↪  [MS-banned] (CWE-120)
lstrcatW 4  Does not check for buffer overflows when concatenating to destination
    ↪  [MS-banned] (CWE-120)
```

```
lstrcatn 1  Easily used incorrectly (e.g., incorrectly computing the correct
    ↪ maximum size to add) [MS-banned] (CWE-120)
lstrcatnA   4  Does not check for buffer overflows when concatenating to
    ↪ destination [MS-banned] (CWE-120)
lstrcatnW   4  Does not check for buffer overflows when concatenating to
    ↪ destination [MS-banned] (CWE-120)
lstrcpy  4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
lstrcpyA 4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
lstrcpyW 4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
lstrcpyn 1  Easily used incorrectly; doesn't always \0-terminate or check for
    ↪ invalid pointers [MS-banned] (CWE-120)
lstrcpynA   4  Does not check for buffer overflows when copying to destination [
    ↪ MS-banned] (CWE-120)
lstrcpynW   4  Does not check for buffer overflows when copying to destination [
    ↪ MS-banned] (CWE-120)
lstrncat 4  Does not check for buffer overflows when concatenating to destination
    ↪  [MS-banned] (CWE-120)
memalign 1  On some systems (though not Linux-based systems) an attempt to free()
    ↪  results from memalign() may fail. This may, on a few systems, be
    ↪ exploitable.  Also note that memalign() may not check that the boundary
    ↪ parameter is correct (CWE-676)
memcpy   2  Does not check for buffer overflows when copying to destination (CWE
    ↪ -120)
mismatch 1  Function does not check the second iterator for over-read conditions
    ↪ (CWE-126)
mkstemp  2  Potential for temporary file vulnerability in some circumstances.
    ↪ Some older Unix-like systems create temp files with permission to write by
    ↪ all by default, so be sure to set the umask to override this. Also, some
    ↪ older Unix systems might fail to use O_EXCL when opening the file, so make
    ↪ sure that O_EXCL is used by the library (CWE-377)
mktemp   4  Temporary file race condition (CWE-377)
mrand48  3  This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
nrand48  3  This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
open  2  Check when opening files - can an attacker redirect it (via symlinks),
    ↪ force the opening of special file type (e.g., device files), move things
    ↪ around to create a race condition, control its ancestors, or change its
    ↪ contents? (CWE-362)
popen 4  This causes a new program to execute and is difficult to use safely (CWE
    ↪ -78)
printf   4  If format strings can be influenced by an attacker, they can be
    ↪ exploited (CWE-134)
random   3  This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
read  1  Check buffer boundaries if used in a loop including recursive loops (CWE
    ↪ -120, CWE-20)
readlink 5  This accepts filename arguments; if an attacker can move those files
    ↪ or change the link content, a race condition results.  Also, it does not
    ↪ terminate with ASCII NUL. (CWE-362, CWE-20)
readv 0  Function accepts input from outside program (CWE-20)
realpath 3  This function does not protect against buffer overflows, and some
    ↪ implementations can overflow internally (CWE-120/CWE-785!)
recv  0  Function accepts input from outside program (CWE-20)
```

```
recvfrom 0  Function accepts input from outside program (CWE-20)
recvmsg  0  Function accepts input from outside program (CWE-20)
scanf 4  The scanf() family's %s operation, without a limit specification,
    ↪ permits buffer overflows (CWE-120, CWE-20)
seed48   3  This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
setstate 3  This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
snprintf 4  If format strings can be influenced by an attacker, they can be
    ↪ exploited, and note that sprintf variations do not always \0-terminate (CWE
    ↪ -134)
sprintf  4  Does not check for buffer overflows (CWE-120)
srand 3  This function is not sufficiently random for security-related functions
    ↪ such as key and nonce creation (CWE-327)
srandom  3  This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
sscanf   4  The scanf() family's %s operation, without a limit specification,
    ↪ permits buffer overflows (CWE-120, CWE-20)
ssignal  2  These functions are considered obsolete on most systems, and very non
    ↪ -portable (Linux-based systems handle them radically different, basically
    ↪ if gsignal/ssignal were the same as raise/signal respectively, while System
    ↪  V considers them a separate set and obsolete) (CWE-676)
strCatBuff  4  Does not check for buffer overflows when concatenating to
    ↪ destination [MS-banned] (CWE-120)
strcadd  1  Subject to buffer overflow if buffer is not as big as claimed (CWE
    ↪ -120)
strcat   4  Does not check for buffer overflows when concatenating to destination
    ↪  [MS-banned] (CWE-120)
strccpy  1  Subject to buffer overflow if buffer is not as big as claimed (CWE
    ↪ -120)
strcpy   4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
strcpyA  4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
strcpyW  4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
strcpynA 4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
streadd  4  This function does not protect against buffer overflows (CWE-120)
strecpy  4  This function does not protect against buffer overflows (CWE-120)
strfry   3  This function is not sufficiently random for security-related
    ↪ functions such as key and nonce creation (CWE-327)
strlen   1  Does not handle strings that are not \0-terminated; if given one it
    ↪ may perform an over-read (it could cause a crash if unprotected) (CWE-126)
strncat  1  Easily used incorrectly (e.g., incorrectly computing the correct
    ↪ maximum size to add) [MS-banned] (CWE-120)
strncpy  1  Easily used incorrectly; doesn't always \0-terminate or check for
    ↪ invalid pointers [MS-banned] (CWE-120)
strtrns  3  This function does not protect against buffer overflows (CWE-120)
swprintf 4  Does not check for buffer overflows (CWE-120)
syslog   4  If syslog's format strings can be influenced by an attacker, they can
    ↪  be exploited (CWE-134)
system   4  This causes a new program to execute and is difficult to use safely (
    ↪ CWE-78)
tempnam  3  Temporary file race condition (CWE-377)
tmpfile  2  Function tmpfile() has a security flaw on some systems (e.g., older
    ↪ System V systems) (CWE-377)
```

```
tmpnam   3  Temporary file race condition (CWE-377)
ulimit   1  This C routine is considered obsolete (as opposed to the shell
    ↪ command by the same name, which is NOT obsolete) (CWE-676)
umask 1  Ensure that umask is given most restrictive possible setting (e.g., 066
    ↪ or 077) (CWE-732)
usleep   1  This C routine is considered obsolete (as opposed to the shell
    ↪ command by the same name).  The interaction of this function with SIGALRM
    ↪ and other timer functions such as sleep(), alarm(), setitimer(), and
    ↪ nanosleep() is unspecified (CWE-676)
vfork 2  On some old systems, vfork() permits race conditions, and it's very
    ↪ difficult to use correctly (CWE-362)
vfprintf 4  If format strings can be influenced by an attacker, they can be
    ↪ exploited (CWE-134)
vfscanf  4  The scanf() family's %s operation, without a limit specification,
    ↪ permits buffer overflows (CWE-120, CWE-20)
vfwprintf   4  If format strings can be influenced by an attacker, they can be
    ↪ exploited (CWE-134)
vfwscanf 4  The scanf() family's %s operation, without a limit specification,
    ↪ permits buffer overflows (CWE-120, CWE-20)
vprintf  4  If format strings can be influenced by an attacker, they can be
    ↪ exploited (CWE-134)
vscanf   4  The scanf() family's %s operation, without a limit specification,
    ↪ permits buffer overflows (CWE-120, CWE-20)
vsnprintf   4  If format strings can be influenced by an attacker, they can be
    ↪ exploited, and note that sprintf variations do not always \0-terminate (CWE
    ↪ -134)
vsprintf 4  Does not check for buffer overflows (CWE-120)
vsscanf  4  The scanf() family's %s operation, without a limit specification,
    ↪ permits buffer overflows (CWE-120, CWE-20)
vswprintf   4  Does not check for buffer overflows (CWE-120)
vswscanf 4  The scanf() family's %s operation, without a limit specification,
    ↪ permits buffer overflows (CWE-120, CWE-20)
vwprintf 4  If format strings can be influenced by an attacker, they can be
    ↪ exploited (CWE-134)
vwscanf  4  The scanf() family's %s operation, without a limit specification,
    ↪ permits buffer overflows (CWE-120, CWE-20)
wchar_t  2  Statically-sized arrays can be improperly restricted, leading to
    ↪ potential overflows or other issues (CWE-119!/CWE-120)
wcscat   4  Does not check for buffer overflows when concatenating to destination
    ↪  [MS-banned] (CWE-120)
wcscpy   4  Does not check for buffer overflows when copying to destination [MS-
    ↪ banned] (CWE-120)
wcslen   1  Does not handle strings that are not \0-terminated; if given one it
    ↪ may perform an over-read (it could cause a crash if unprotected) (CWE-126)
wcsncat  1  Easily used incorrectly (e.g., incorrectly computing the correct
    ↪ maximum size to add) [MS-banned] (CWE-120)
wcsncpy  1  Easily used incorrectly; doesn't always \0-terminate or check for
    ↪ invalid pointers [MS-banned] (CWE-120)
wprintf  4  If format strings can be influenced by an attacker, they can be
    ↪ exploited (CWE-134)
wscanf   4  The scanf() family's %s operation, without a limit specification,
    ↪ permits buffer overflows (CWE-120, CWE-20)
```

Listing A.2: List of all rules in Flawfinder 2.0.11

A.4. All results of binary classification

Model	Features	Train./Val.	Test	Imb.mit.	Accuracy (%)	Sensitivity (%)	Specifity (%)
MLP (Var1)	w2v_avg	LOSARD102	LOSARD102	SMOTE	95.57	96.50	96.64
MLP (Var2)	w2v_avg	LOSARD102	LOSARD102	SMOTE	95.60	95.44	95.76
MLP	w2v_avg	LOSARD102	LOSARD102	None	91.15	82.87	99.43
MLP	w2v_avg	LOSARD102	LOSARD102	undersample	96.34	96.46	96.22
MLP	w2v_avg	LOSARD102	LOSARD102	Weights	96.92	96.94	96.89
MLP	w2v_avg	LOSARD102	JTT	SMOTE	62.57	91.55	33.60
MLP	w2v_avg	LOSARD102	JTT	undersample	59.11	83.57	34.65
MLP	w2v_avg	LOSARD102	JTT	None	50.00	0.00	100.00
MLP	w2v_avg	LOSARD102	JTT	Weights	50.51	98.79	2.22
LSTM	w2v_seq	LOSARD102	LOSARD102	None	99.44	98.92	99.96
LSTM	w2v_seq	LOSARD102	LOSARD102	undersample	99.27	99.15	99.39
LSTM	w2v_seq	LOSARD102	LOSARD102	SMOTE	99.44	99.00	99.88
LSTM	w2v_seq	LOSARD102	LOSARD102	Weights	98.88	97.77	99.99
LSTM	w2v_seq	LOSARD102	JTT	None	50.00	0.00	100.00
LSTM	w2v_seq	LOSARD102	JTT	undersample	50.93	6.08	95.77
LSTM	w2v_seq	LOSARD102	JTT	SMOTE	51.95	5.43	98.47
LSTM	w2v_seq	LOSARD102	JTT	Weights	49.56	0.16	98.95
LSTM	bert128	LOSARD102	LOSARD102	None	98.33	96.74	99.91
LSTM	bert128	LOSARD102	LOSARD102	undersample	98.33	96.74	99.91
LSTM (Var 1)	bert128	LOSARD102	LOSARD102	SMOTE	98.83	97.83	99.83
LSTM (Var 2)	bert128	LOSARD102	LOSARD102	SMOTE	98.83	99.84	97.83

LSTM	bert128	LOSARD102	LOSARD102	Weights	98.56	97.83	99.29
LSTM	bert128	LOSARD102	JTT	None	50.00	0.00	100.00
LSTM	bert128	LOSARD102	JTT	undersample	52.65	6.45	98.85
LSTM (Var 1)	bert128	LOSARD102	JTT	SMOTE	49.96	0.00	99.92
LSTM (Var 2)	bert128	LOSARD102	JTT	SMOTE	50.00	0.00	100.00
LSTM	bert128	LOSARD102	JTT	Weights	50.50	1.08	99.92
LSTM	bert64	LOSARD102	LOSARD102	None	100.00	100.00	100.00
LSTM	bert64	LOSARD102	LOSARD102	undersample	99.87	100.00	99.74
LSTM (Var 1)	bert64	LOSARD102	LOSARD102	SMOTE	99.96	100.00	99.92
LSTM (Var 2)	bert64	LOSARD102	LOSARD102	SMOTE	99.95	100.00	99.90
LSTM	bert64	LOSARD102	LOSARD102	Weights	99.92	100.00	99.85
LSTM	bert64	LOSARD102	JTT	None	50.00	0.00	100.00
LSTM	bert64	LOSARD102	JTT	undersample	50.00	0.00	100.00
LSTM	bert64	LOSARD102	JTT	SMOTE	50.00	0.00	100.00
LSTM	bert64	LOSARD102	JTT	Weights	50.00	0.00	100.00
ResNet	w2v_seq	LOSARD102	LOSARD102	None	99.26	99.08	99.44
ResNet	w2v_seq	LOSARD102	LOSARD102	undersample	99.50	99.23	99.77
ResNet	w2v_seq	LOSARD102	LOSARD102	SMOTE	52.05	4.93	99.18
ResNet	w2v_seq	LOSARD102	LOSARD102	Weights	99.07	99.54	98.60
ResNet	w2v_seq	LOSARD102	JTT	None	50.00	0.00	100.00
ResNet	w2v_seq	LOSARD102	JTT	undersample	49.97	0.16	99.78
ResNet	w2v_seq	LOSARD102	JTT	SMOTE	39.69	22.37	57.00
ResNet	w2v_seq	LOSARD102	JTT	Weights	50.00	0.00	100.00
ResNet	bert128	LOSARD102	LOSARD102	None	98.33	96.74	99.92

ResNet	bert128	LOSARD102	LOSARD102	undersample	50.00	0.00	100.00
ResNet	bert128	LOSARD102	LOSARD102	SMOTE	98.84	97.83	99.85
ResNet	bert128	LOSARD102	LOSARD102	Weights	98.17	97.83	98.52
ResNet	bert128	LOSARD102	JTT	None	50.00	0.00	100.00
ResNet	bert128	LOSARD102	JTT	undersample	50.00	0.00	100.00
ResNet	bert128	LOSARD102	JTT	SMOTE	50.00	0.00	100.00
ResNet	bert128	LOSARD102	JTT	Weights	53.69	8.60	98.78
ResNet	bert64	LOSARD102	LOSARD102	None	99.96	100.00	99.92
ResNet	bert64	LOSARD102	LOSARD102	undersample	99.96	100.00	99.92
ResNet	bert64	LOSARD102	LOSARD102	SMOTE	99.96	100.00	99.92
ResNet	bert64	LOSARD102	LOSARD102	Weights	99.46	100.00	98.93
ResNet	bert64	LOSARD102	JTT	None	50.00	0.00	100.00
ResNet	bert64	LOSARD102	JTT	undersample	50.00	0.00	100.00
ResNet	bert64	LOSARD102	JTT	SMOTE	50.00	0.00	100.00
ResNet	bert64	LOSARD102	JTT	Weights	50.00	0.00	100.00

Table A.1.: Comparison of binary classification results

MLP word2vec

```
              precision    recall  f1-score   support
         0.0       0.99      0.99      0.99     53944
         1.0       0.93      0.83      0.87      4601
    accuracy                           0.98     58545
   macro avg       0.96      0.91      0.93     58545
weighted avg       0.98      0.98      0.98     58545

Confusion matrix:
[[53639   305]
 [  788  3813]]
Confusion matrix (Percentages):
[[91.62   0.521]
 [ 1.346  6.513]]
Metrics:
Sensitivity (TPR): 0.828733
Specifity (TNR):   0.994346
FPR: 0.005654
FNR: 0.171267
Balanced accuracy: 0.911539
```

MLP word2vec undersample

```
              precision    recall  f1-score   support
         0.0       1.00      0.96      0.98     53944
         1.0       0.69      0.96      0.80      4601
    accuracy                           0.96     58545
   macro avg       0.84      0.96      0.89     58545
weighted avg       0.97      0.96      0.97     58545

Confusion matrix:
[[51907  2037]
 [  163  4438]]
Confusion matrix (Percentages):
[[88.662  3.479]
 [ 0.278  7.58 ]]
Metrics:
Sensitivity (TPR): 0.964573
Specifity (TNR):   0.962239
FPR: 0.037761
FNR: 0.035427
Balanced accuracy: 0.963406
```

MLP word2vec weighted

```
              precision    recall  f1-score   support
         0.0       1.00      0.97      0.98     53944
         1.0       0.73      0.97      0.83      4601
    accuracy                           0.97     58545
   macro avg       0.86      0.97      0.91     58545
weighted avg       0.98      0.97      0.97     58545

Confusion matrix:
[[52269  1675]
 [  141  4460]]
Confusion matrix (Percentages):
[[89.28   2.861]
 [ 0.241  7.618]]
Metrics:
Sensitivity (TPR): 0.969354
Specifity (TNR):   0.968949
FPR: 0.031051
FNR: 0.030646
Balanced accuracy: 0.969152
```

MLP word2vec SMOTE Var1

```
              precision    recall  f1-score   support
         0.0       1.00      0.97      0.98     53944
         1.0       0.71      0.97      0.82      4601
    accuracy                           0.97     58545
   macro avg       0.85      0.97      0.90     58545
weighted avg       0.97      0.97      0.97     58545

Confusion matrix:
[[52130  1814]
 [  161  4440]]
Confusion matrix (Percentages):
[[89.043  3.098]
 [ 0.275  7.584]]
Metrics:
Sensitivity (TPR): 0.965008
Specifity (TNR):   0.966373
FPR: 0.033627
FNR: 0.034992
Balanced accuracy: 0.965690
```

MLP word2vec SMOTE Var2

```
              precision    recall  f1-score   support
         0.0       1.00      0.96      0.98     53944
         1.0       0.66      0.95      0.78      4601
    accuracy                           0.96     58545
   macro avg       0.83      0.96      0.88     58545
weighted avg       0.97      0.96      0.96     58545

Confusion matrix:
[[51656  2288]
 [  210  4391]]
Confusion matrix (Percentages):
[[88.233  3.908]
 [ 0.359  7.5  ]]
Metrics:
Sensitivity (TPR): 0.954358
Specify (TNR):   0.957586
FPR: 0.042414
FNR: 0.045642
Balanced accuracy: 0.955972
```

testJTT MLP word2vec

```
              precision    recall  f1-score   support
         0.0       0.97      1.00      0.98     24171
         1.0       0.00      0.00      0.00       828
    accuracy                           0.97     24999
   macro avg       0.48      0.50      0.49     24999
weighted avg       0.93      0.97      0.95     24999

Confusion matrix:
[[24171     0]
 [  828     0]]
Confusion matrix (Percentages):
[[96.688  0.   ]
 [ 3.312  0.   ]]
Metrics:
Sensitivity (TPR): 0.000000
Specify (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

testJTT MLP word2vec undersample

```
              precision    recall  f1-score   support

         0.0       0.98      0.35      0.51     24171
         1.0       0.04      0.84      0.08       828
    accuracy                           0.36     24999
   macro avg       0.51      0.59      0.30     24999
weighted avg       0.95      0.36      0.50     24999

Confusion matrix:
[[ 8376 15795]
 [  136   692]]
Confusion matrix (Percentages):
[[33.505 63.183]
 [ 0.544  2.768]]
Metrics:
Sensitivity (TPR): 0.835749
Specifity (TNR):   0.346531
FPR: 0.653469
FNR: 0.164251
Balanced accuracy: 0.591140
```

testJTT MLP word2vec weighted

```
              precision    recall  f1-score   support
         0.0       0.98      0.02      0.04     24171
         1.0       0.03      0.99      0.06       828
    accuracy                           0.05     24999
   macro avg       0.51      0.51      0.05     24999
weighted avg       0.95      0.05      0.04     24999

Confusion matrix:
[[  537 23634]
 [   10   818]]
Confusion matrix (Percentages):
[[2.148e+00 9.454e+01]
 [4.000e-02 3.272e+00]]
Metrics:
Sensitivity (TPR): 0.987923
Specifity (TNR):   0.022217
FPR: 0.977783
FNR: 0.012077
Balanced accuracy: 0.505070
```

testJTT MLP word2vec SMOTE

```
              precision    recall  f1-score   support
         0.0       0.99      0.34      0.50     24171
         1.0       0.05      0.92      0.09       828
    accuracy                           0.36     24999
   macro avg       0.52      0.63      0.29     24999
weighted avg       0.96      0.36      0.49     24999

Confusion matrix:
[[ 8122 16049]
 [   70   758]]
Confusion matrix (Percentages):
[[32.489 64.199]
 [ 0.28   3.032]]
Metrics:
Sensitivity (TPR): 0.915459
Specifity (TNR):   0.336023
FPR: 0.663977
FNR: 0.084541
Balanced accuracy: 0.625741
```

LSTM word2vec

```
              precision    recall  f1-score   support
         0.0       1.00      1.00      1.00     31916
         1.0       0.99      0.99      0.99      1299
    accuracy                           1.00     33215
   macro avg       0.99      0.99      0.99     33215
weighted avg       1.00      1.00      1.00     33215

Confusion matrix:
[[31903    13]
 [   14  1285]]
Confusion matrix (Percentages):
[[9.605e+01 3.900e-02]
 [4.200e-02 3.869e+00]]
Metrics:
Sensitivity (TPR): 0.989222
Specifity (TNR):   0.999593
FPR: 0.000407
FNR: 0.010778
Balanced accuracy: 0.994408
```

LSTM word2vec undersample

```
              precision    recall  f1-score   support
         0.0       1.00      0.99      1.00     31916
         1.0       0.87      0.99      0.93      1299
    accuracy                           0.99     33215
   macro avg       0.93      0.99      0.96     33215
weighted avg       0.99      0.99      0.99     33215

Confusion matrix:
[[31722   194]
 [   11  1288]]
Confusion matrix (Percentages):
[[9.5505e+01 5.8400e-01]
 [3.3000e-02 3.8780e+00]]
Metrics:
Sensitivity (TPR): 0.991532
Specifity (TNR):   0.993922
FPR: 0.006078
FNR: 0.008468
Balanced accuracy: 0.992727
```

LSTM word2vec weighted

```
              precision    recall  f1-score   support
         0.0       1.00      1.00      1.00     31916
         1.0       1.00      0.98      0.99      1299
    accuracy                           1.00     33215
   macro avg       1.00      0.99      0.99     33215
weighted avg       1.00      1.00      1.00     33215

Confusion matrix:
[[31913     3]
 [   29  1270]]
Confusion matrix (Percentages):
[[9.608e+01 9.000e-03]
 [8.700e-02 3.824e+00]]
Metrics:
Sensitivity (TPR): 0.977675
Specifity (TNR):   0.999906
FPR: 0.000094
FNR: 0.022325
Balanced accuracy: 0.988791
```

LSTM word2vec SMOTE

```
              precision    recall  f1-score   support
         0.0       1.00      1.00      1.00     31916
         1.0       0.97      0.99      0.98      1299
    accuracy                           1.00     33215
   macro avg       0.99      0.99      0.99     33215
weighted avg       1.00      1.00      1.00     33215

Confusion matrix:
[[31879    37]
 [   13  1286]]
Confusion matrix (Percentages):
[[9.5978e+01 1.1100e-01]
 [3.9000e-02 3.8720e+00]]
Metrics:
Sensitivity (TPR): 0.989992
Specifity (TNR):   0.998841
FPR: 0.001159
FNR: 0.010008
Balanced accuracy: 0.994417
```

testJTT LSTM word2vec

```
              precision    recall  f1-score   support
         0.0       0.97      1.00      0.99     21243
         1.0       0.00      0.00      0.00       608
    accuracy                           0.97     21851
   macro avg       0.49      0.50      0.49     21851
weighted avg       0.95      0.97      0.96     21851

Confusion matrix:
[[21243     0]
 [  608     0]]
Confusion matrix (Percentages):
[[97.218  0.   ]
 [ 2.782  0.   ]]
Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

testJTT LSTM word2vec undersample

```
              precision    recall  f1-score   support
         0.0       0.97      0.96      0.97     21243
         1.0       0.04      0.06      0.05       608
    accuracy                           0.93     21851
   macro avg       0.51      0.51      0.51     21851
weighted avg       0.95      0.93      0.94     21851

Confusion matrix:
[[20346   897]
 [  571    37]]
Confusion matrix (Percentages):
[[93.112  4.105]
 [ 2.613  0.169]]
Metrics:
Sensitivity (TPR): 0.060855
Specifity (TNR):   0.957774
FPR: 0.042226
FNR: 0.939145
Balanced accuracy: 0.509315
```

testJTT LSTM word2vec weighted

```
              precision    recall  f1-score   support
         0.0       0.97      0.99      0.98     21243
         1.0       0.00      0.00      0.00       608
    accuracy                           0.96     21851
   macro avg       0.49      0.50      0.49     21851
weighted avg       0.95      0.96      0.95     21851

Confusion matrix:
[[21021   222]
 [  607     1]]
Confusion matrix (Percentages):
[[9.6202e+01 1.0160e+00]
 [2.7780e+00 5.0000e-03]]
Metrics:
Sensitivity (TPR): 0.001645
Specifity (TNR):   0.989549
FPR: 0.010451
FNR: 0.998355
Balanced accuracy: 0.495597
```

testJTT LSTM word2vec SMOTE

```
              precision    recall  f1-score   support
         0.0       0.97      0.98      0.98     21243
         1.0       0.09      0.05      0.07       608
    accuracy                           0.96     21851
   macro avg       0.53      0.52      0.52     21851
weighted avg       0.95      0.96      0.95     21851

Confusion matrix:
[[20918   325]
 [  575    33]]
Confusion matrix (Percentages):
[[95.73   1.487]
 [ 2.631  0.151]]
Metrics:
Sensitivity (TPR): 0.054276
Specifity (TNR):   0.984701
FPR: 0.015299
FNR: 0.945724
Balanced accuracy: 0.519489
```

LSTM bert128

```
              precision    recall  f1-score   support
           0       1.00      1.00      1.00     10146
           1       0.91      0.97      0.94        92
    accuracy                           1.00     10238
   macro avg       0.95      0.98      0.97     10238
weighted avg       1.00      1.00      1.00     10238

Confusion matrix:
[[10137     9]
 [    3    89]]
Confusion matrix (Percentages):
[[9.9013e+01 8.8000e-02]
 [2.9000e-02 8.6900e-01]]
Metrics:
Sensitivity (TPR): 0.967391
Specifity (TNR):   0.999113
FPR: 0.000887
FNR: 0.032609
Balanced accuracy: 0.983252
```

LSTM bert128 undersample

```
              precision    recall  f1-score   support
           0       1.00      1.00      1.00     10146
           1       0.91      0.97      0.94        92
    accuracy                           1.00     10238
   macro avg       0.95      0.98      0.97     10238
weighted avg       1.00      1.00      1.00     10238

Confusion matrix:
[[10137     9]
 [    3    89]]
Confusion matrix (Percentages):
[[9.9013e+01 8.8000e-02]
 [2.9000e-02 8.6900e-01]]
Metrics:
Sensitivity (TPR): 0.967391
Specifity (TNR):   0.999113
FPR: 0.000887
FNR: 0.032609
Balanced accuracy: 0.983252
```

LSTM bert128 weighted

```
              precision    recall  f1-score   support
           0       1.00      0.99      1.00     10146
           1       0.56      0.98      0.71        92
    accuracy                           0.99     10238
   macro avg       0.78      0.99      0.85     10238
weighted avg       1.00      0.99      0.99     10238

Confusion matrix:
[[10074    72]
 [    2    90]]
Confusion matrix (Percentages):
[[9.8398e+01 7.0300e-01]
 [2.0000e-02 8.7900e-01]]
Metrics:
Sensitivity (TPR): 0.978261
Specifity (TNR):   0.992904
FPR: 0.007096
FNR: 0.021739
Balanced accuracy: 0.985582
```

LSTM bert128 SMOTE Var1

```
              precision    recall  f1-score   support
           0       1.00      1.00      1.00     10146
           1       0.84      0.98      0.90        92
    accuracy                           1.00     10238
   macro avg       0.92      0.99      0.95     10238
weighted avg       1.00      1.00      1.00     10238

Confusion matrix:
[[10129    17]
 [    2    90]]
Confusion matrix (Percentages):
[[9.8935e+01 1.6600e-01]
 [2.0000e-02 8.7900e-01]]
Metrics:
Sensitivity (TPR): 0.978261
Specifity (TNR):   0.998324
FPR: 0.001676
FNR: 0.021739
Balanced accuracy: 0.988293
```

LSTM bert128 SMOTE Var2

```
              precision    recall  f1-score   support
           0       1.00      1.00      1.00     10146
           1       0.85      0.98      0.91        92
    accuracy                           1.00     10238
   macro avg       0.92      0.99      0.95     10238
weighted avg       1.00      1.00      1.00     10238

Confusion matrix:
[[10130    16]
 [    2    90]]
Confusion matrix (Percentages):
[[9.8945e+01 1.5600e-01]
 [2.0000e-02 8.7900e-01]]
Metrics:
Sensitivity (TPR): 0.978261
Specifity (TNR):   0.998423
FPR: 0.001577
FNR: 0.021739
Balanced accuracy: 0.988342
```

testJTT LSTM bert128

```
              precision    recall  f1-score   support
           0       0.99      1.00      0.99      7879
           1       0.00      0.00      0.00        93
    accuracy                           0.99      7972
   macro avg       0.49      0.50      0.50      7972
weighted avg       0.98      0.99      0.98      7972

Confusion matrix:
[[7879    0]
 [  93    0]]
Confusion matrix (Percentages):
[[98.833  0.   ]
 [ 1.167  0.   ]]
Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

testJTT LSTM bert128 SMOTE Var1

```
              precision    recall  f1-score   support
           0       0.99      1.00      0.99      7879
           1       0.00      0.00      0.00        93
    accuracy                           0.99      7972
   macro avg       0.49      0.50      0.50      7972
weighted avg       0.98      0.99      0.98      7972

Confusion matrix:
[[7873    6]
 [  93    0]]
Confusion matrix (Percentages):
[[9.8758e+01 7.5000e-02]
 [1.1670e+00 0.0000e+00]]
Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   0.999238
FPR: 0.000762
FNR: 1.000000
Balanced accuracy: 0.499619
```

testJTT LSTM bert128 SMOTE Var2

```
              precision    recall  f1-score   support
           0       0.99      1.00      0.99      7879
           1       0.00      0.00      0.00        93
    accuracy                           0.99      7972
   macro avg       0.49      0.50      0.50      7972
weighted avg       0.98      0.99      0.98      7972

Confusion matrix:
[[7879    0]
 [  93    0]]
Confusion matrix (Percentages):
[[98.833  0.   ]
 [ 1.167  0.   ]]
Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

testJTT LSTM bert128 undersample

```
              precision    recall  f1-score   support
           0       0.99      0.99      0.99      7879
           1       0.06      0.06      0.06        93
    accuracy                           0.98      7972
   macro avg       0.53      0.53      0.53      7972
weighted avg       0.98      0.98      0.98      7972

Confusion matrix:
[[7788   91]
 [  87    6]]
Confusion matrix (Percentages):
[[9.7692e+01 1.1410e+00]
 [1.0910e+00 7.5000e-02]]
Metrics:
Sensitivity (TPR): 0.064516
Specifity (TNR):   0.988450
FPR: 0.011550
FNR: 0.935484
Balanced accuracy: 0.526483
```

testJTT LSTM bert128 weighted

```
              precision    recall  f1-score   support
           0       0.99      1.00      0.99      7879
           1       0.14      0.01      0.02        93
    accuracy                           0.99      7972
   macro avg       0.57      0.50      0.51      7972
weighted avg       0.98      0.99      0.98      7972

Confusion matrix:
[[7873    6]
 [  92    1]]
Confusion matrix (Percentages):
[[9.8758e+01 7.5000e-02]
 [1.1540e+00 1.3000e-02]]
Metrics:
Sensitivity (TPR): 0.010753
Specifity (TNR):   0.999238
FPR: 0.000762
FNR: 0.989247
Balanced accuracy: 0.504996
```

LSTM bert64

```
              precision    recall  f1-score   support
           0       1.00      1.00      1.00      3914
           1       1.00      1.00      1.00        77
    accuracy                           1.00      3991
   macro avg       1.00      1.00      1.00      3991
weighted avg       1.00      1.00      1.00      3991

Confusion matrix:
[[3914    0]
 [   0   77]]
Confusion matrix (Percentages):
[[98.071  0.   ]
 [ 0.     1.929]]
Metrics:
Sensitivity (TPR): 1.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 0.000000
Balanced accuracy: 1.000000
```

LSTM bert64 SMOTE Var1

```
              precision    recall  f1-score   support
           0       1.00      1.00      1.00      3914
           1       0.96      1.00      0.98        77
    accuracy                           1.00      3991
   macro avg       0.98      1.00      0.99      3991
weighted avg       1.00      1.00      1.00      3991

Confusion matrix:
[[3911    3]
 [   0   77]]
Confusion matrix (Percentages):
[[9.7995e+01 7.5000e-02]
 [0.0000e+00 1.9290e+00]]

Metrics:
Sensitivity (TPR): 1.000000
Specifity (TNR):   0.999234
FPR: 0.000766
FNR: 0.000000
Balanced accuracy: 0.999617
```

LSTM bert64 SMOTE Var2

```
              precision    recall  f1-score   support
           0       1.00      1.00      1.00      3914
           1       0.95      1.00      0.97        77
    accuracy                           1.00      3991
   macro avg       0.98      1.00      0.99      3991
weighted avg       1.00      1.00      1.00      3991

Confusion matrix:
[[3910    4]
 [   0   77]]
Confusion matrix (Percentages):
[[97.97   0.1  ]
 [ 0.     1.929]]
Metrics:
Sensitivity (TPR): 1.000000
Specifity (TNR):   0.998978
FPR: 0.001022
FNR: 0.000000
Balanced accuracy: 0.999489
```

LSTM bert64 undersample

```
              precision    recall  f1-score   support
           0       1.00      1.00      1.00      3914
           1       0.89      1.00      0.94        77
    accuracy                           1.00      3991
   macro avg       0.94      1.00      0.97      3991
weighted avg       1.00      1.00      1.00      3991

Confusion matrix:
[[3904   10]
 [   0   77]]
Confusion matrix (Percentages):
[[97.82   0.251]
 [ 0.     1.929]]
Metrics:
Sensitivity (TPR): 1.000000
Specifity (TNR):   0.997445
FPR: 0.002555
FNR: 0.000000
Balanced accuracy: 0.998723
```

LSTM bert64 weighted

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3914
           1       0.93      1.00      0.96        77

    accuracy                           1.00      3991
   macro avg       0.96      1.00      0.98      3991
weighted avg       1.00      1.00      1.00      3991

Confusion matrix:
[[3908    6]
 [   0   77]]
Confusion matrix (Percentages):
[[97.92   0.15 ]
 [ 0.     1.929]]
Metrics:
Sensitivity (TPR): 1.000000
Specifity (TNR):   0.998467
FPR: 0.001533
FNR: 0.000000
Balanced accuracy: 0.999234
```

testJTT LSTM bert64

```
              precision    recall  f1-score   support
           0       0.99      1.00      1.00      5578
           1       0.00      0.00      0.00        29
    accuracy                           0.99      5607
   macro avg       0.50      0.50      0.50      5607
weighted avg       0.99      0.99      0.99      5607

Confusion matrix:
[[5578    0]
 [  29    0]]
Confusion matrix (Percentages):
[[99.483  0.   ]
 [ 0.517  0.   ]]
Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

testJTT LSTM bert64 SMOTE

```
              precision    recall  f1-score   support
           0       0.99      1.00      1.00      5578
           1       0.00      0.00      0.00        29
    accuracy                           0.99      5607
   macro avg       0.50      0.50      0.50      5607
weighted avg       0.99      0.99      0.99      5607

Confusion matrix:
[[5578    0]
 [  29    0]]
Confusion matrix (Percentages):
[[99.483  0.   ]
 [ 0.517  0.   ]]
Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

testJTT LSTM bert64 undersample

```
              precision    recall  f1-score   support
           0       0.99      1.00      1.00      5578
           1       0.00      0.00      0.00        29
    accuracy                           0.99      5607
   macro avg       0.50      0.50      0.50      5607
weighted avg       0.99      0.99      0.99      5607

Confusion matrix:
[[5578    0]
 [  29    0]]
Confusion matrix (Percentages):
[[99.483  0.   ]
 [ 0.517  0.   ]]
Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

testJTT LSTM bert64 weighted

```
              precision    recall  f1-score   support
           0       0.99      1.00      1.00      5578
           1       0.00      0.00      0.00        29
    accuracy                           0.99      5607
   macro avg       0.50      0.50      0.50      5607
weighted avg       0.99      0.99      0.99      5607

Confusion matrix:
[[5578    0]
 [  29    0]]
Confusion matrix (Percentages):
[[99.483  0.   ]
 [ 0.517  0.   ]]
Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

ResNet word2vec

```
              precision    recall  f1-score   support
         0.0       1.00      0.99      1.00     31916
         1.0       0.88      0.99      0.93      1299
    accuracy                           0.99     33215
   macro avg       0.94      0.99      0.96     33215
weighted avg       0.99      0.99      0.99     33215

Confusion matrix:
[[31738   178]
 [   12  1287]]
Confusion matrix (Percentages):
[[9.5553e+01 5.3600e-01]
 [3.6000e-02 3.8750e+00]]
Metrics:
Sensitivity (TPR): 0.990762
Specifity (TNR):   0.994423
FPR: 0.005577
FNR: 0.009238
Balanced accuracy: 0.992592
```

ResNet word2vec SMOTE

```
              precision    recall  f1-score   support
         0.0       0.96      0.99      0.98     31916
         1.0       0.20      0.05      0.08      1299
    accuracy                           0.95     33215
   macro avg       0.58      0.52      0.53     33215
weighted avg       0.93      0.95      0.94     33215

Confusion matrix:
[[31654   262]
 [ 1235    64]]
Confusion matrix (Percentages):
[[95.3    0.789]
 [ 3.718  0.193]]
Metrics:
Sensitivity (TPR): 0.049269
Specifity (TNR):   0.991791
FPR: 0.008209
FNR: 0.950731
Balanced accuracy: 0.520530
```

ResNet word2vec undersample

```
              precision    recall  f1-score   support
         0.0       1.00      1.00      1.00     31916
         1.0       0.95      0.99      0.97      1299
    accuracy                           1.00     33215
   macro avg       0.97      0.99      0.98     33215
weighted avg       1.00      1.00      1.00     33215

Confusion matrix:
[[31841    75]
 [   10  1289]]
Confusion matrix (Percentages):
[[9.5863e+01 2.2600e-01]
 [3.0000e-02 3.8810e+00]]
Metrics:
Sensitivity (TPR): 0.992302
Specifity (TNR):   0.997650
FPR: 0.002350
FNR: 0.007698
Balanced accuracy: 0.994976
```

ResNet word2vec weighted

```
              precision    recall  f1-score   support
         0.0       1.00      0.99      0.99     31916
         1.0       0.74      1.00      0.85      1299
    accuracy                           0.99     33215
   macro avg       0.87      0.99      0.92     33215
weighted avg       0.99      0.99      0.99     33215

Confusion matrix:
[[31468   448]
 [    6  1293]]
Confusion matrix (Percentages):
[[9.474e+01 1.349e+00]
 [1.800e-02 3.893e+00]]
Metrics:
Sensitivity (TPR): 0.995381
Specifity (TNR):   0.985963
FPR: 0.014037
FNR: 0.004619
Balanced accuracy: 0.990672
```

testJTT ResNet word2vec

```
              precision    recall  f1-score   support
         0.0       0.97      1.00      0.99     21243
         1.0       0.00      0.00      0.00       608
    accuracy                           0.97     21851
   macro avg       0.49      0.50      0.49     21851
weighted avg       0.95      0.97      0.96     21851

Confusion matrix:
[[21243     0]
 [  608     0]]
Confusion matrix (Percentages):
[[97.218  0.   ]
 [ 2.782  0.   ]]
Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

testJTT ResNet word2vec undersample

```
              precision    recall  f1-score   support
         0.0       0.97      1.00      0.98     21243
         1.0       0.02      0.00      0.00       608
    accuracy                           0.97     21851
   macro avg       0.50      0.50      0.49     21851
weighted avg       0.95      0.97      0.96     21851

Confusion matrix:
[[21197    46]
 [  607     1]]
Confusion matrix (Percentages):
[[9.7007e+01 2.1100e-01]
 [2.7780e+00 5.0000e-03]]
Metrics:
Sensitivity (TPR): 0.001645
Specifity (TNR):   0.997835
FPR: 0.002165
FNR: 0.998355
Balanced accuracy: 0.499740
```

testJTT ResNet word2vec weighted

```
              precision    recall  f1-score   support
         0.0       0.97      1.00      0.99     21243
         1.0       0.00      0.00      0.00       608
    accuracy                           0.97     21851
   macro avg       0.49      0.50      0.49     21851
weighted avg       0.95      0.97      0.96     21851

Confusion matrix:
[[21243     0]
 [  608     0]]
Confusion matrix (Percentages):
[[97.218  0.   ]
 [ 2.782  0.   ]]
Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

testJTT ResNet word2vec SMOTE

```
              precision    recall  f1-score   support
         0.0       0.96      0.57      0.72     21243
         1.0       0.01      0.22      0.03       608
    accuracy                           0.56     21851
   macro avg       0.49      0.40      0.37     21851
weighted avg       0.94      0.56      0.70     21851

Confusion matrix:
[[12109  9134]
 [  472   136]]
Confusion matrix (Percentages):
[[55.416 41.801]
 [ 2.16   0.622]]
Metrics:
Sensitivity (TPR): 0.223684
Specifity (TNR):   0.570023
FPR: 0.429977
FNR: 0.776316
Balanced accuracy: 0.396854
```

ResNet Bert128

```
              precision    recall  f1-score   support
           0       1.00      1.00      1.00     10146
           1       0.92      0.97      0.94        92
    accuracy                           1.00     10238
   macro avg       0.96      0.98      0.97     10238
weighted avg       1.00      1.00      1.00     10238

Confusion matrix:
[[10138     8]
 [    3    89]]
Confusion matrix (Percentages):
[[9.9023e+01 7.8000e-02]
 [2.9000e-02 8.6900e-01]]
Metrics:
Sensitivity (TPR): 0.967391
Specifity (TNR):   0.999212
FPR: 0.000788
FNR: 0.032609
Balanced accuracy: 0.983301
```

ResNet Bert128 undersample

```
              precision    recall  f1-score   support
           0       0.99      1.00      1.00     10146
           1       0.00      0.00      0.00        92
    accuracy                           0.99     10238
   macro avg       0.50      0.50      0.50     10238
weighted avg       0.98      0.99      0.99     10238

Confusion matrix:
[[10146     0]
 [   92     0]]
Confusion matrix (Percentages):
[[99.101  0.   ]
 [ 0.899  0.   ]]
Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

ResNet Bert128 weighted

```
              precision    recall  f1-score   support
           0       1.00      0.99      0.99     10146
           1       0.38      0.98      0.54        92
    accuracy                           0.99     10238
   macro avg       0.69      0.98      0.77     10238
weighted avg       0.99      0.99      0.99     10238

Confusion matrix:
[[9996  150]
 [   2   90]]
Confusion matrix (Percentages):
[[9.7636e+01 1.4650e+00]
 [2.0000e-02 8.7900e-01]]
Metrics:
Sensitivity (TPR): 0.978261
Specifity (TNR):   0.985216
FPR: 0.014784
FNR: 0.021739
Balanced accuracy: 0.981738
```

ResNet Bert128 SMOTE

```
              precision    recall  f1-score   support
           0       1.00      1.00      1.00     10146
           1       0.86      0.98      0.91        92
    accuracy                           1.00     10238
   macro avg       0.93      0.99      0.96     10238
weighted avg       1.00      1.00      1.00     10238

Confusion matrix:
[[10131    15]
 [    2    90]]
Confusion matrix (Percentages):
[[9.8955e+01 1.4700e-01]
 [2.0000e-02 8.7900e-01]]
Metrics:
Sensitivity (TPR): 0.978261
Specifity (TNR):   0.998522
FPR: 0.001478
FNR: 0.021739
Balanced accuracy: 0.988391
```

testJTT ResNet bert128

```
              precision    recall  f1-score   support
           0       0.99      1.00      0.99      7879
           1       0.00      0.00      0.00        93
    accuracy                           0.99      7972
   macro avg       0.49      0.50      0.50      7972
weighted avg       0.98      0.99      0.98      7972

Confusion matrix:
[[7879    0]
 [  93    0]]
Confusion matrix (Percentages):
[[98.833  0.   ]
 [ 1.167  0.   ]]
Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

testJTT ResNet bert128 undersample

```
              precision    recall  f1-score   support
           0       0.99      1.00      0.99      7879
           1       0.00      0.00      0.00        93
    accuracy                           0.99      7972
   macro avg       0.49      0.50      0.50      7972
weighted avg       0.98      0.99      0.98      7972

Confusion matrix:
[[7879    0]
 [  93    0]]
Confusion matrix (Percentages):
[[98.833  0.   ]
 [ 1.167  0.   ]]
Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

testJTT ResNet bert128 weighted

```
              precision    recall  f1-score   support
           0       0.99      0.99      0.99      7879
           1       0.08      0.09      0.08        93
    accuracy                           0.98      7972
   macro avg       0.53      0.54      0.53      7972
weighted avg       0.98      0.98      0.98      7972

Confusion matrix:
[[7783   96]
 [  85    8]]
Confusion matrix (Percentages):
[[97.629  1.204]
 [ 1.066  0.1  ]]
Metrics:
Sensitivity (TPR): 0.086022
Specifity (TNR):   0.987816
FPR: 0.012184
FNR: 0.913978
Balanced accuracy: 0.536919
```

testJTT ResNet bert128 SMOTE

```
              precision    recall  f1-score   support
           0       0.99      1.00      0.99      7879
           1       0.00      0.00      0.00        93
    accuracy                           0.99      7972
   macro avg       0.49      0.50      0.50      7972
weighted avg       0.98      0.99      0.98      7972

Confusion matrix:
[[7879    0]
 [  93    0]]
Confusion matrix (Percentages):
[[98.833  0.   ]
 [ 1.167  0.   ]]

Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

ResNet bert64

```
              precision    recall  f1-score   support
           0       1.00      1.00      1.00      3914
           1       0.96      1.00      0.98        77
    accuracy                           1.00      3991
   macro avg       0.98      1.00      0.99      3991
weighted avg       1.00      1.00      1.00      3991

Confusion matrix:
[[3911    3]
 [   0   77]]
Confusion matrix (Percentages):
[[9.7995e+01 7.5000e-02]
 [0.0000e+00 1.9290e+00]]
Metrics:
Sensitivity (TPR): 1.000000
Specifity (TNR):   0.999234
FPR: 0.000766
FNR: 0.000000
Balanced accuracy: 0.999617
```

ResNet bert64 undersample

```
              precision    recall  f1-score   support
           0       1.00      1.00      1.00      3914
           1       0.96      1.00      0.98        77
    accuracy                           1.00      3991
   macro avg       0.98      1.00      0.99      3991
weighted avg       1.00      1.00      1.00      3991

Confusion matrix:
[[3911    3]
 [   0   77]]
Confusion matrix (Percentages):
[[9.7995e+01 7.5000e-02]
 [0.0000e+00 1.9290e+00]]
Metrics:
Sensitivity (TPR): 1.000000
Specifity (TNR):   0.999234
FPR: 0.000766
FNR: 0.000000
Balanced accuracy: 0.999617
```

ResNet bert64 weighted

```
              precision    recall  f1-score   support
           0       1.00      0.99      0.99      3914
           1       0.65      1.00      0.79        77
    accuracy                           0.99      3991
   macro avg       0.82      0.99      0.89      3991
weighted avg       0.99      0.99      0.99      3991

Confusion matrix:
[[3872   42]
 [   0   77]]
Confusion matrix (Percentages):
[[97.018  1.052]
 [ 0.     1.929]]
Metrics:
Sensitivity (TPR): 1.000000
Specifity (TNR):   0.989269
FPR: 0.010731
FNR: 0.000000
Balanced accuracy: 0.994635
```

ResNet bert64 SMOTE

```
              precision    recall  f1-score   support
           0       1.00      1.00      1.00      3914
           1       0.96      1.00      0.98        77
    accuracy                           1.00      3991
   macro avg       0.98      1.00      0.99      3991
weighted avg       1.00      1.00      1.00      3991

Confusion matrix:
[[3911    3]
 [   0   77]]
Confusion matrix (Percentages):
[[9.7995e+01 7.5000e-02]
 [0.0000e+00 1.9290e+00]]
Metrics:
Sensitivity (TPR): 1.000000
Specifity (TNR):   0.999234
FPR: 0.000766
FNR: 0.000000
Balanced accuracy: 0.999617
```

testJTT ResNet bert64

```
              precision    recall  f1-score   support
           0       0.99      1.00      1.00      5578
           1       0.00      0.00      0.00        29
    accuracy                           0.99      5607
   macro avg       0.50      0.50      0.50      5607
weighted avg       0.99      0.99      0.99      5607

Confusion matrix:
[[5578    0]
 [  29    0]]
Confusion matrix (Percentages):
[[99.483  0.   ]
 [ 0.517  0.   ]]

Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

testJTT ResNet bert64 undersample

```
              precision    recall  f1-score   support
           0       0.99      1.00      1.00      5578
           1       0.00      0.00      0.00        29
    accuracy                           0.99      5607
   macro avg       0.50      0.50      0.50      5607
weighted avg       0.99      0.99      0.99      5607

Confusion matrix:
[[5578    0]
 [  29    0]]
Confusion matrix (Percentages):
[[99.483  0.   ]
 [ 0.517  0.   ]]

Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

testJTT ResNet bert64 weighted

```
              precision    recall  f1-score   support
           0       0.99      1.00      1.00      5578
           1       0.00      0.00      0.00        29
    accuracy                           0.99      5607
   macro avg       0.50      0.50      0.50      5607
weighted avg       0.99      0.99      0.99      5607

Confusion matrix:
[[5578    0]
 [  29    0]]
Confusion matrix (Percentages):
[[99.483  0.   ]
 [ 0.517  0.   ]]

Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

testJTT ResNet bert64 SMOTE

```
              precision    recall  f1-score   support
           0       0.99      1.00      1.00      5578
           1       0.00      0.00      0.00        29
    accuracy                           0.99      5607
   macro avg       0.50      0.50      0.50      5607
weighted avg       0.99      0.99      0.99      5607

Confusion matrix:
[[5578    0]
 [  29    0]]
Confusion matrix (Percentages):
[[99.483  0.   ]
 [ 0.517  0.   ]]

Metrics:
Sensitivity (TPR): 0.000000
Specifity (TNR):   1.000000
FPR: 0.000000
FNR: 1.000000
Balanced accuracy: 0.500000
```

A.5. All results of vulnerability type classification

Model	Train./Val.	Test	Precision	Recall
MLP Var1	LOSARD102	LOSARD102	0.8089	0.5628
MLP Var2	LOSARD102	LOSARD102	0.8539	0.5793
MLP Var1	LOSARD102 (distinct)	LOSARD102 (distinct)	0.7545	0.8900
MLP Var2	LOSARD102 (distinct)	LOSARD102 (distinct)	0.7807	0.9022

Table A.2.: Metrics comparison of CWE classifier models on total and distinct dataset

Multi-label classification of CWE clusters - Variant 1

	precision	recall	f1-score	support
CWE-664	0.9989	0.7221	0.8383	2634
CWE-682	0.5132	0.3362	0.4062	116
CWE-691	0.9405	0.4566	0.6148	173
CWE-693	0.9845	0.8900	0.9349	2291
CWE-697	0.3333	0.2812	0.3051	32
CWE-707	0.9883	0.8369	0.9063	2716
CWE-710	0.9032	0.4164	0.5700	269
micro avg	0.9798	0.7840	0.8710	8231
macro avg	0.8089	0.5628	0.6536	8231
weighted avg	0.9776	0.7840	0.8660	8231
samples avg	0.7655	0.7547	0.7566	8231

Multi-label classification of CWE clusters - Variant 2

	precision	recall	f1-score	support
CWE-664	0.9949	0.6602	0.7937	2634
CWE-682	0.5676	0.3621	0.4421	116
CWE-691	0.9615	0.4335	0.5976	173
CWE-693	0.9827	0.9158	0.9480	2291
CWE-697	0.7500	0.5625	0.6429	32
CWE-707	0.9813	0.8682	0.9213	2716
CWE-710	0.7391	0.2528	0.3767	269
micro avg	0.9762	0.7773	0.8655	8231
macro avg	0.8539	0.5793	0.6746	8231
weighted avg	0.9709	0.7773	0.8555	8231
samples avg	0.7438	0.7367	0.7372	8231

Classification of distinct CWE clusters - Variant 1

	precision	recall	f1-score	support
CWE-664	0.9922	0.8783	0.9318	1454
CWE-682	0.6350	0.9775	0.7699	89
CWE-691	0.8158	0.9208	0.8651	101
CWE-693	1.0000	0.9091	0.9524	11
CWE-697	0.1606	0.9565	0.2750	23
CWE-707	1.0000	0.6790	0.8088	162
CWE-710	0.6780	0.9091	0.7767	132
micro avg	0.8717	0.8717	0.8717	1972
macro avg	0.7545	0.8900	0.7685	1972
weighted avg	0.9370	0.8717	0.8930	1972
samples avg	0.8717	0.8717	0.8717	1972

Classification of distinct CWE clusters - Variant 2

	precision	recall	f1-score	support
CWE-664	0.9924	0.9017	0.9449	1454
CWE-682	0.4265	0.9775	0.5939	89
CWE-691	0.8230	0.9208	0.8692	101
CWE-693	0.9091	0.9091	0.9091	11
CWE-697	0.5128	0.8696	0.6452	23
CWE-707	1.0000	0.7901	0.8828	162
CWE-710	0.8013	0.9470	0.8681	132
micro avg	0.8996	0.8996	0.8996	1972
macro avg	0.7807	0.9022	0.8161	1972
weighted avg	0.9400	0.8996	0.9112	1972
samples avg	0.8996	0.8996	0.8996	1972

List of Figures

List of Tables

List of Listings

Bibliography

[1] D. G. Altman and J. M. Bland. Statistics notes: Diagnostic tests 1: sensitivity and specificity. *BMJ*, 308(6943):1552–1552, jun 1994. doi:10.1136/bmj.308.6943.1552.

[2] Chris Aniszczyk. Open sourcing the kubernetes security audit. last checked on 2020-08-25, August 2019. URL: https://www.cncf.io/blog/2019/08/06/open-sourcing-the-kubernetes-security-audit/.

[3] Tripti Borah, Kandarpa Sarma, and Pranhari Talukdar. *Biometric Identification System Using Neuro and Fuzzy Computational Approaches*, pages 335–368. December 2015. doi:10.4018/978-1-4666-8654-0.ch016.

[4] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002. doi:10.1613/jair.953.

[5] B. Chess and G. McGraw. Static analysis for security. *IEEE Security and Privacy Magazine*, 2(6):76–79, nov 2004. doi:10.1109/msp.2004.111.

[6] Intel Corporation. Intel® Bug Bounty Program Terms. last checked on 2020-06-04. URL: https://www.intel.com/content/www/us/en/security-center/bug-bounty-program.html.

[7] MITRE Corporation. About cwe, August 2020. last checked on 2020-09-25. URL: http://cwe.mitre.org/about/index.html.

[8] CWE Content Team. CWE VIEW: Research Concepts, 2018. last checked on 2020-08-11. URL: https://cwe.mitre.org/data/definitions/1000.html.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi:10.18653/v1/N19-1423.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[11] google-research (Various Contributors). bert/readme.md, March 2020. last updated on 2020-03-11, last checked on 2020-09-25. URL: https://github.com/google-research/bert/blob/8028c0459485299fa1ae6692b2300922a3fa2bad/README.md.

[12] Ruslan Habalov and Timo Schmid. Vulncode-DB. last checked on 2020-09-25. URL: https://www.vulncode-db.com/about.

[13] Jacob A. Harer, Louis Y. Kim, Rebecca L. Russell, Onur Ozdemir, Leonard R. Kosta, Akshay Rangamani, Lei H. Hamilton, Gabriel I. Centeno, Jonathan R. Key, Paul M. Ellingwood, Erik Antelman, Alan Mackay, Marc W. McConley, Jeffrey M. Opper, Peter Chin, and Tomo Lazovich. Automated software vulnerability detection with machine learning. last checked on 2020-09-25. URL: http://arxiv.org/abs/1803.04497v2.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.

[15] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. *Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München*, 1991. Diploma thesis.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, nov 1997. doi:10.1162/neco.1997.9.8.1735.

[17] Denis Howe. *abstract syntax tree from FOLDOC*, free online dictionary of computing edition. last checked on 2020-09-25. URL: https://foldoc.org/abstract+syntax+tree.

[18] IEEE and The Open Group. The open group base specifications issue 7, 2018 edition, 2018. Revision of IEEE Std 1003.1-2008, last checked on 2020-09-28. URL: https://pubs.opengroup.org/onlinepubs/9699919799/.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, may 2017. doi:10.1145/3065386.

[20] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, dec 1989. doi:10.1162/neco.1989.1.4.541.

[21] Zhen Li, Deqing Zou, Jing Tang, Zhihao Zhang, Mingqian Sun, and Hai Jin. A Comparative Study of Deep Learning-Based Vulnerability Detection System. *IEEE Access*, 7:103184–103197, 2019. doi:10.1109/ACCESS.2019.2930578.

[22] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. last checked on 2020-09-25. URL: http://arxiv.org/abs/1807.06756v2.

[23] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In Patrick Traynor and Alina Oprea, editors, *Proceedings 2018 Network and Distributed System Security Symposium*, Reston, VA, February 18-21, 2018. Internet Society. doi:10.14722/ndss.2018.23158.

[24] LLVM. libclang: C interface to clang, 2020. last checked on 2020-09-22. URL: https://clang.llvm.org/doxygen/group___CINDEX.html.

[25] Microsoft Corporation. Microsoft Bug Bounty Program. last checked on 2020-08-25. URL: https://www.microsoft.com/en-us/msrc/bounty.

[26] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. In *Proceedings of Workshop at ICLR*, September 2013.

[27] National Institute of Standards and Technology. Software Assurance Reference Dataset, November 2017. last checked on 2020-03-01. URL: https://samate.nist.gov/SRD/index.php.

[28] NSA Center for Assured Software. Juliet test suite v1.2 for c/c++ user guide. last checked on 2020-09-22, December 2012. URL: https://samate.nist.gov/SRD/resources/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf.

[29] NSA Center for Assured Software. Release log for juliet 1.3, October 2017. last checked on 2020-09-22. URL: https://samate.nist.gov/SRD/resources/releaseJuliet1.3Doc.txt.

[30] Charles Oliveira. Iarpa stonesoup phase 3 test cases, October 2015. last checked on 2020-09-25. URL: https://samate.nist.gov/SRD/view.php?tsID=102.

[31] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985. last checked on 2020-09-28. URL: https://apps.dtic.mil/dtic/tr/fulltext/u2/a164453.pdf.

[32] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, oct 1986. doi:10.1038/323533a0.

[33] Bruce Schneier. The Vulnerabilities Market and the Future of Security, 2012. last checked on 2020-08-25. URL: https://www.forbes.com/sites/bruceschneier/2012/05/30/the-vulnerabilities-market-and-the-future-of-security/.

[34] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(nov):2579–2605, 2008.

[35] David A. Wheeler. Flawfinder Home Page. last checked on 2020-05-27. URL: https://dwheeler.com/flawfinder/.

[36] David A. Wheeler. *Flawfinder Manual*, August 2017. last checked on 2020-08-25. URL: https://dwheeler.com/flawfinder/flawfinder.pdf.

[37] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing [review article]. *IEEE Computational Intelligence Magazine*, 13(3):55–75, aug 2018. doi:10.1109/mci.2018.2840738.

[38] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer Vision – ECCV 2014*, pages 818–833. Springer International Publishing, 2014. doi:10.1007/978-3-319-10590-1_53.