Automated Buffer Overflow Detection using Deep Learning

Tristan Gaudron
Mines ParisTech - PSL University
Paris, France
tristan.gaudron@mines-paristech.fr

Jonathan Griffe
Mines ParisTech - PSL University
Paris, France
jonathan.griffe@mines-paristech.fr

Georges-André Silber
Mines ParisTech - PSL University
Paris, France

***Abstract*—According to the Center for Strategic and International Studies, cybercrime has cost almost $1 trillion to the global economy in 2020. Buffer Overflows, one of the most common types of software vulnerability, can enable attacks on computer systems. Software companies can produce huge amounts of code and thus manually checking them is too costly and practically never done. In order to reduce the amount of vulnerabilities in softwares, various automated vulnerability detection tools have been developed. However, such tools often have a high false positive or false negative rate, cannot detect precisely where the vulnerability is, and can only detect a specific set of types of vulnerability. With the advent of Deep Learning technologies, some new tools integrate Machine Learning and Deep Learning to obtain better performances. In this article, we review several of those solutions, implement one such solution and obtain experimental results.**

***Index Terms*—Buffer overflow, Automated detection, Deep Learning**

I. Introduction

As human life is more and more reliant on an increasing amount of diverse softwares, and a large majority of the security incidents result from vulnerabilities in the application layer, detecting these vulnerabilities in the production phase is becoming more crucial. One of the most common of these vulnerabilities is the Buffer Overflow (BOF).

A BOF happens when a program writes on a memory address outside of the intended data structure. Typically, when data is written in a buffer but exceeds the length fixed when the memory was firstly allocated, statically in the stack or dynamically in the heap. The main risk is that nearby data can be overwritten and thus, leading to unexpected and potentially dangerous behaviours like access to confidential data or execution of arbitrary code. Especially if the BOF happens while writing user-controlled data, these unexpected behaviours can be exploited to create such behaviours on purpose.

BOF is a type of vulnerability which occurs particularly when using low-level programming languages like C or C++. Indeed, the performances are often at high stake and would be lowered with too many safety-checks at runtime. However, these safety-checks exist and can be enabled during compilation. An example can be *canaries*, known values that are stored between different pieces of data in memory whose integrity checks can reveal BOF which would have overwritten them. There are many other such safety protocols, like random addressing, safe-stack, garbage collection, control-flow integrity (CFI) etc. [26]. Recent programming languages like Rust or Go impose safety at compile time to avoid heavy runtime support. However, BOF can still happen in projects using mixed binary as with Mozilla Firefox where CFI protected C code together with safe Rust code lead to an actual controlled overflow when providing the browser with a specially crafted SVG image [2].

Because of the huge amount of code produced, manual security testing is not realistic. To address this issue, many tools have been developed to allow for automatic detection of buffer overflows, either by static analysis (i.e. analysing the source code without running it) or by dynamic analysis (i.e. running the program, testing inputs and detecting abnormal behavior). Such tools have many flaws, starting with the high amount of false positives in static analysis tools [24] which means that while the tools reduce the amount of time needed to complete security checks, large amounts of manual analysis is still required. Similarly, the dynamic analysis tools tend to only test a small subset of possible attack angles [24], and therefore are not reliable and do not eliminate the need for manual checks of the entire source code. Both approaches also suffer from other flaws such as coarse granularity : the tool is unable to precisely detect which part of the code is involved in the vulnerability. Indeed, it is difficult to detect precisely every BOF because of the variety of forms they can have.

Because of these flaws, many methods have been tried to improve the performances, including Deep Learning. Several tools that have been developed for Natural Language Processing (NLP) applications can be used in vulnerability detection in source code, such as word2vec [25]. Along with the progress made in Natural Language Processing thanks to Deep Learning, we can reasonably expect to be able to make comparable progress in vulnerability detection by adapting the NLP methods.

The first part of this article is an overview of conventional methods for automated Buffer Overflow detection including methods using Machine Learning and then especially Deep Learning approaches. The second part is dedicated to our implementation, largely inspired by the work of [1]. We present and analyse the choices we made and our results, we then discuss ideas for future work and code representation methods.

II. Conventional Methods

A. Static Analysis

Static Analysis tools detect vulnerabilities by examining the source code, detecting statements that may be vulnerable and applying rules and methods to determine whether the statement is vulnerable. To help differentiate between vulnerable and not vulnerable statements, it can derive information from the code such as :

- Control flow graph : graph of the sequence of execution of the code. It can be used to further determine if a statement is vulnerable. [5]
- Context : the arguments passed in function calls, which helps differentiate between multiple function calls with different arguments. [6].
- Path : identifying BOFs statements that can be reached, whereas some statements in a control flow graph cannot be reached [6]
- Value Range : the value range that a variable can take. This can detect vulnerabilities for example when computed for buffer sizes. [7]

The first static analysis tools derived from compiler optimization tools but with the rise of security concerns, dedicated vulnerability detection tools were developed, such as Flawfinder [8]. Multiple methods of vulnerability detection have been used.

First, some methods use tainted data-flow [9]. Untrusted variables are marked tainted; these tainted variables are for example user inputs. When a tainted variable is used in a security-sensitive function such as $strcpy$, a warning is raised. A tainted variable can become untainted by being sanitized. This method can detect buffer overflows caused by unsanitized inputs being used in critical operations.

Another method uses constraints [10]. For each security-sensitive operation, a constraint is generated so that any violation of this constraint is a potential vulnerability. These constraints are then propagated through the program by using system dependence graphs or control flow graphs. For example, these constraints can be a range $[a, b]$ of buffer allocation size. Once they are all generated, a constraint solver is used to find an input which violates the constraint system. If a solution is found, it means the function is vulnerable.

A last method relies on pattern-matching [11]. It identifies library functions used in the code that are known to be vulnerable, and tries to match the source code with vulnerable code patterns that include these library functions. If the code includes one such vulnerable code pattern, a potential vulnerability is detected.

B. Dynamic Analysis

Contrary to the static analysis, during dynamic analysis the program is executed and the vulnerability detection relies on supplying inputs and identifying unexpected behavior. There are multiple techniques used in dynamic analysis.

The first is fault-injection [12]. For this analysis, the program data, such as input data or environment data is corrupted and the program is then executed with this corrupted data in order to see if the program handles it correctly. Examples of data used for such corruption are special characters and large strings.

A more straightforward method is the use of Attack Signatures [13]. Attack Signatures are strings known to cause unexpected behavior in other programs, generated from experience or gathered from security reports. These strings are then used as input in the tested program.

There are also other methods of dynamic analysis used to reveal BOFs in programs, such as search analysis [15], which takes a random input, tests it, modifies it and repeats in order to detect vulnerabilities. Different algorithms are used in order to generate new inputs. For example, genetic algorithms can be used to generate new inputs from mutated versions of the original input [14].

C. Hybrid Analysis

Both dynamic and static analysis are not accurate means of detecting buffer overflows. Indeed, static analysis suffers from large amounts of false positives, because the difference in source code between vulnerable and not vulnerable is often small, and the reason why a statement is not vulnerable is often not trivial and difficult to detect. On the other hand, dynamic analysis methods only test a narrow angle of attack, and a limited amount of test cases, and therefore suffer from a high false negative rate [24]. These issues can be mitigated by combining both static and dynamic analysis. This hybrid analysis uses static analysis to detect statements that might be vulnerable, which enable the use of a more extensive dynamic analysis on a smaller amount of test cases detected by static analysis. The dynamic analysis is used to verify if the statement can indeed be exploited and is truly vulnerable, reducing the amount of false positives. The combination of static and dynamic analysis allows for the use of techniques that can not be possible in either static or dynamic analysis alone.

For example, one such technique uses static analysis to detect a set of instructions that can write to a specific memory address and then use dynamic analysis to detect if memory addresses have been written by an instruction that wasn't supposed to [16] or detects the memory addresses where an instruction can write, and then use dynamic analysis to detect if the instruction wrote at an address where it wasn't supposed to [17].

III. Machine Learning Methods

Most of the machine learning methods belong in one of three groups of methods : anomaly detection, vulnerable code pattern recognition and software metrics data mining.

A. Software Metrics Approach

This approach is the most simple, and also the least successful. It uses data often gathered from Github repositories of open-source projects. This data is software metrics such as the number of contributors to a function, the number of lines of a function or the experience of the contributors. The goal of these algorithms is not exactly to detect vulnerabilities but to build a fault prediction model in order to focus manual checking by security experts on parts of the code that are more likely to be vulnerable [18]. Unsurprisingly, these tools reveal that functions written by less experienced contributors, more contributors and longer functions tend to have more vulnerabilities. Despite their poor results, this approach is interesting because fault prediction models are already used in software engineering, therefore building models applied to vulnerabilities prediction can be easily done by the same engineers.

B. Anomaly Detection

This method relies on a set of explicit and implicit programming patterns gathered through machine learning and text mining techniques to detect anomalies : patterns that do not conform to the learnt patterns are considered as potential vulnerabilities [19]. The biggest benefit of this method is that it relies on unsupervised learning and therefore does not require a labeled dataset, which in the case of vulnerability detection is very difficult to build. The first step is to automatically extract API usage patterns from a corpus through machine learning or text mining. The benefits of this step compared to a conventional static analysis algorithm is that the rules are not designed by a human, which means they are less likely to have errors, and that "implicit" programming rules, that a human programmer might follow without being aware, are also included. The second step is to detect code patterns that differ from the learnt API usage patterns. While this method does not specify the type of vulnerability, it has a finer granularity as it allows to detect which API call triggers the vulnerability.

C. Vulnerable Code Pattern Recognition

Like anomaly detection, vulnerable code pattern recognition also relies on extracting features from source code, but instead of extracting correct API usage patterns, it extracts the vulnerable code [20]. A drawback of this approach is that instead of taking code for the dataset and assuming most of it is not vulnerable, it uses supervised learning which means it needs a labeled dataset composed of vulnerable and not vulnerable functions. One of the easiest uses of this method is to train it on a specific vulnerability in order to detect if there are other occurrences resembling a newly discovered vulnerability. As this process is a common task for security analysts and can take a long time, this approach is very interesting. These methods can also precisely detect the location of the vulnerability.

IV. Deep Learning Methods

A. First works

The first attempt at using deep learning to analyse programs [21] uses a convolutional neural network (CNN) for program classifying tasks. It relies on a method to transform the abstract syntax tree of the input code into a vector representation, which is then fed to the CNN.

Li et al [22] then apply the deep learning approaches developed for program analysis to vulnerability detection. They developed a tool called Vulnerability Deep Pecker (VulDeePecker) based on a Recurrent Neural Network (RNN) which successfully helped in detecting zero-day vulnerabilities in software products.

More recently, Russell et al [23] leveraged Natural Language Processing (NLP) methods along with manually building a vulnerability dataset composed of millions of open-source software functions to develop a fast and powerful tool for automated vulnerability detection. The method discussed in the following section is based on this work.

B. Ground work

X. Li et al [1] proposed a method based on the approach of Russell et al [23], with many improvements to remediate to the multiple problems encountered by the previous approach.

This method is composed of four steps depicted in Figure 1. The first is a preprocessing of input code. The second trains the model of vector representation of code. The third trains a CNN to extract high-level features, and the last one trains a classifier to detect vulnerable and not vulnerable codes from these high-level features.

1) Medium Intermediate Representation : [1] suggests to use a preprocessing step which consists in building security slices to get rid of long dependencies. Indeed, a BOF may be the result of the interaction between lines that are far from each other in the code, making it difficult to detect. This problem is solved by building security slices : subsets of the input code that contain all the lines involved in a potential BOF. They are built by choosing keywords that could lead to BOF (like a $memcpy$ for exemple) and use a Program Dependency Graph (PDG) in order to conserve only the code from which the sink (the instruction containing the keyword) depends. Finally, when security slices are built, they are split into blocks of constant size called Medium Intermediate Representations (MIR) that are easier to work with.

2) Vectorization : In order to apply the deep-learning machinery, one may want to represent codes in a proper way that will be easier to handle than raw text. Hopefully, a common practice in NLP is to transform words or groups of words into vectors, which are easier to process, on many different levels. The solution [1] chose to deal with this issue is to learn a projection of each word based on their contexts on a low-dimension space which would be "well organised" based on the assumption that two similar words would appear in similar contexts. *In-fine*, similar words would be close in this representation space and one could imagine vector operations like $vec(\text{"king"}) - vec(\text{"man"}) + vec(\text{"woman"}) \approxeq vec(\text{"queen"})$.

3) Network architecture : After obtaining the vector representations, a neural network is used. To detect different features from different context size, three different CNNs are used, each with a different window size, which means each CNN will take into account a context of a different size. Multiple filters are used to extract multiple features. During the CNN training phase, the CNN is followed by a max-pooling

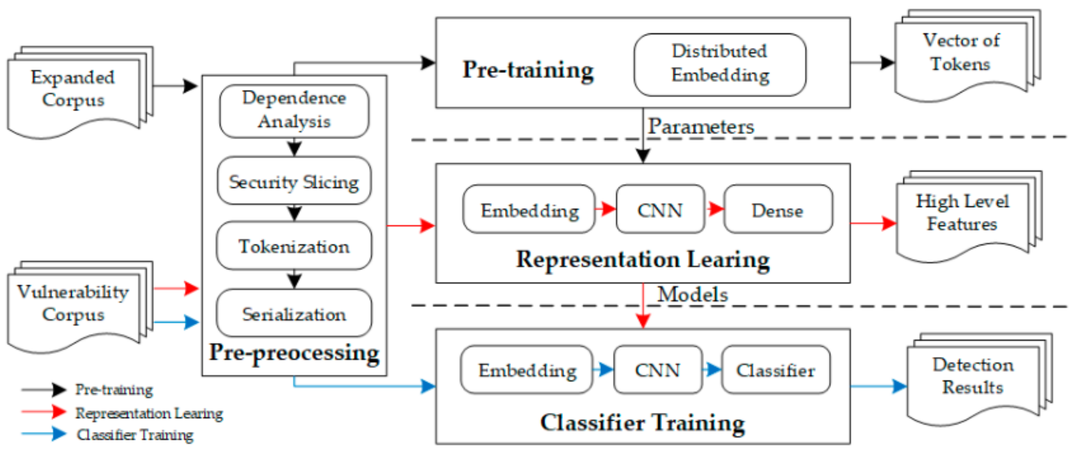
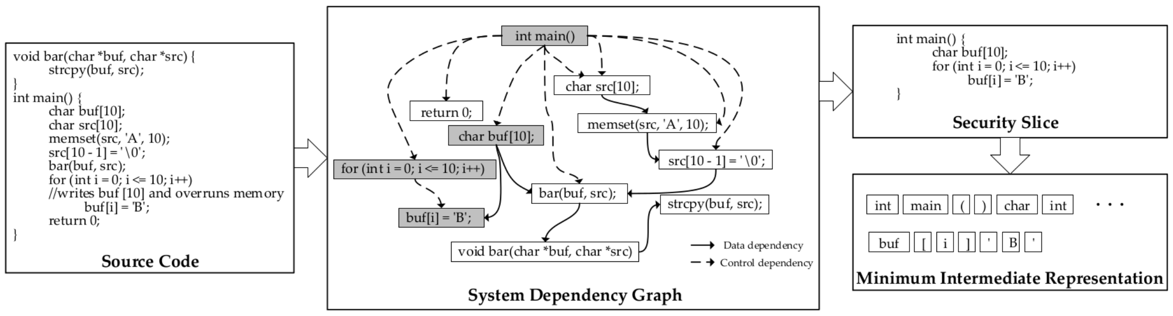


Fig. 1. Detailed model proposed in [1]



```
void bar(char *buf, char *src) {
        strcpy(buf, src);
}
int main() {
        char buf[10];
        char src[10];
        memset(src, 'A', 10);
        src[10 - 1] = '\0';
        bar(buf, src);
        for (int i = 0; i <= 10; i++)
        //writes buf [10] and overruns memory
                buf[i] = 'B';
        return 0;
}
```

Source Code

```
int main() {
        char buf[10];
        for (int i = 0; i <= 10; i++)
                buf[i] = 'B';
}
```

Security Slice

Fig. 2. Pre-processing step from [1]

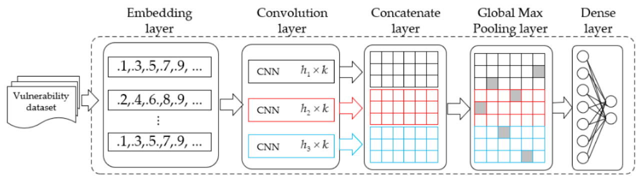
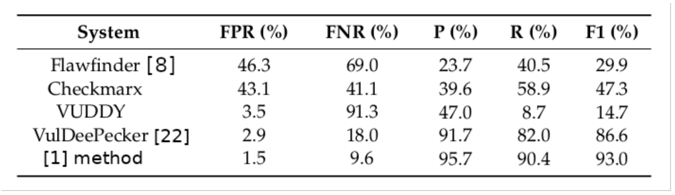


Fig. 3. architecture from [1]



System	FPR (%)	FNR (%)	P (%)	R (%)	F1 (%)
Flawfinder [8]	46.3	69.0	23.7	40.5	29.9
Checkmarx	43.1	41.1	39.6	58.9	47.3
VUDDY	3.5	91.3	47.0	8.7	14.7
VulDeePecker [22]	2.9	18.0	91.7	82.0	86.6
[1] method	1.5	9.6	95.7	90.4	93.0

Fig. 4. results from [1] method in comparison to other BOF detection softwares

along the features dimension to guarantee the extraction of high-level features while keeping the spatial information, and then two dense layers.

Once this model is trained, the max-pooling and dense layers are removed, and only the CNN is kept. A classifier is then added to the CNN and trained to classify vulnerable and non-vulnerable codes using the CNN output. The CNN weights are frozen during the learning phase of the classifier. Six classifiers were used : Logistic Regression, Naive Bayesian, Support Vector Machine, Multi-Layer Perceptron, Gradient Boosting Decision Tree and Random Forest. The classifier which gave the best results was Random Forest.

4) Results: The dataset used in [1] is quite huge (thousands of samples) and comes from the Software Assurance Reference Dataset (SARD) and the National Vulnerability Database (NVD). Their results are very good since they outperform the state-of-the-art methods in 2020. Figure 4 compares their results with other BOF detection softwares. The metrics they use are False Positive Rate (FPR), False Negative Rate (FNR), Precision (P), Recall (R) and F1 score.

V. Implementation and Experiments

We decided to implement an approach similar to the one presented above, composed of a pre-processing of the source code to get a minimum intermediate representation transformed to a vector representation with word2vec, a single

CNN layer followed by a max-pooling and finally two dense layers.

A. Code Generation

While the training of word2vec is a unsupervised step, the training of the rest of the model is supervised and therefore required a labeled dataset.

Gathering the dataset is a complicated task, and often one of the most crucial steps in similar approaches. To be able to have a somewhat representative dataset, it is important to include vulnerable code from many different types of software. While there are vulnerability databases available online, there is no labeled dataset. Therefore it would be necessary to manually search for vulnerabilities in databases, select those that belong in an open-source project, search the relevant code and manually assemble the dataset.

This process would have taken far more time than we had, which is why we decided to opt to automatically generate code in order to build a dataset. To ensure a training with a good quality, we defined some requirements in order to have a decent training :

- Include several Buffer Overflow patterns to guarantee the detection of different types of BOFs
- Include patched version of vulnerabilities to ensure that the method can differentiate between vulnerable and not vulnerable versions of the same API call
- Create other instructions that contain no vulnerabilities to make sure the solution can identify vulnerabilities anywhere in the code
- Disperse the vulnerability pattern in the code to simulate a real code, in which the BOF might rely on a interaction between instructions that are dispersed in the code.

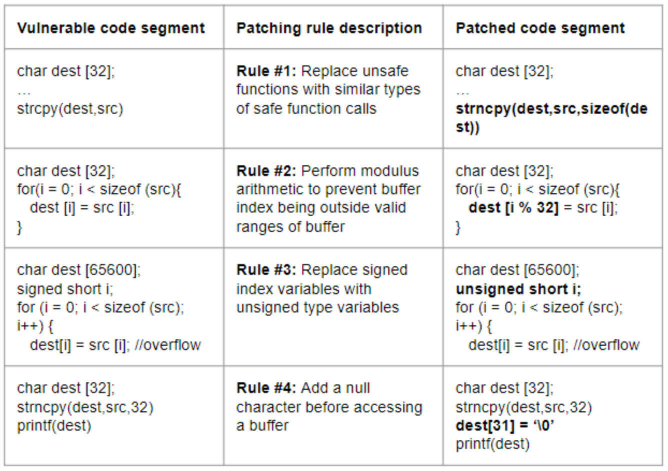
The solution that we used to meet all these requirements was to write multiple vulnerable patterns and their patched version, which can be seen in Figure 5, along with other patterns of code, for example the declaration of two variables and then an operation with them. To generate a code, the program simply randomly draws one vulnerable or not vulnerable pattern, and a random amount of other patterns of codes. All these patterns are then mixed together.

We then generated 10 000 vulnerable codes and 10 000 not vulnerable codes to constitute our dataset. While this solution doesn't allow us to build a tool capable of detecting vulnerabilities in real software, it allows us to train and test our implementation.

B. Security slices

Now that we have a dataset to learn on, we need to segment the codes in smaller blocks (because we can't pass a list with millions of tokens at once in the network). To do so, we used the security slices and MIR detailed in IV-B1

Generating a PDG can be a very hard task, all the more if many user-crafted functions are called. To turn the task into something easier, we chose to focus only on a single user-crafted function and use a tool named Joern [4] to generate the PDGs. Then, we computed the security slices using the



Vulnerable code segment	Patching rule description	Patched code segment

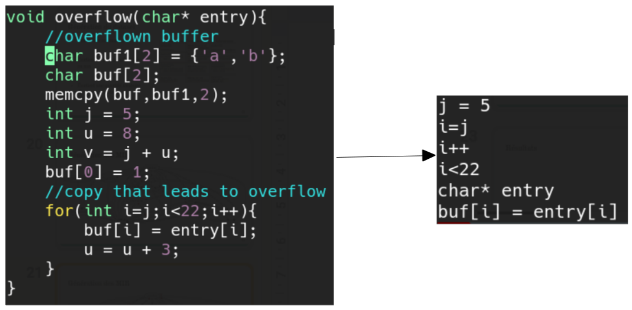
char dest [32];
...
strcpy(dest,src)
	Rule #1: Replace unsafe functions with similar types of safe function calls	char dest [32];
		...
		strncpy(dest,src,sizeof(de st))

char dest [32];
for(i = 0; i < sizeof (src){
dest [i] = src [i];
}
	Rule #2: Perform modulus arithmetic to prevent buffer index being outside valid ranges of buffer	char dest [32];
		for(i = 0; i < sizeof (src){
		dest [i % 32] = src [i];
		}

char dest [65600];
signed short i;
for (i = 0; i < sizeof (src);
i++) {
dest[i] = src [i]; //overflow
	Rule #3: Replace signed index variables with unsigned type variables	char dest [65600];
		unsigned short i;
		for (i = 0; i < sizeof (src);
		i++) {
		dest[i] = src [i]; //overflow

char dest [32];
strncpy(dest,src,32)
printf(dest)
	Rule #4: Add a null character before accessing a buffer	char dest [32];
		strncpy(dest,src,32)
		dest[31] = '\0'
		printf(dest)

Fig. 5. Buffer Overflow patterns used in the code generator. Four different patterns were used, each in their vulnerable version for vulnerable codes and in their patched version for not vulnerable codes



```
void overflow(char* entry){
    //overflown buffer
    char buf1[2] = {'a','b'};
    char buf[2];
    memcpy(buf,buf1,2);
    int j = 5;
    int u = 8;
    int v = j + u;
    buf[0] = 1;
    //copy that leads to overflow
    for(int i=j;i<22;i++){
        buf[i] = entry[i];
        u = u + 3;
    }
}
```

```
j = 5
i=j
i++
i<22
char* entry
buf[i] = entry[i]
```

Fig. 6. Security slice example generated from the sink : buf[i] = entry[i]

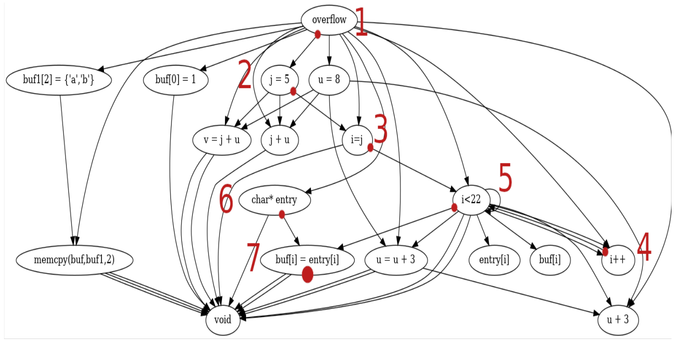


Fig. 7. PDG generated from the last code example using Joern [4]. The nodes pointed with red dot is the connected component leading to the expression $buf[i] = entry[i]$

latter, which was a much easier task.
To compute the security slices, we need to find the connected component (using only indirect edges directions) leading to the sink. Then, to keep lines in a coherent order, we need to use a topological sort on the connected component (see Figure 7). We computed the topological sort together with the generation of the connected component using a DFS (Deep-First Search) starting from the sink and using reversed edges direction. By stacking vertices the last time they are reached, we get a topological sort which, when printed, is the final security slice (see Figure 6).
These security slices, when split into constant-size blocks (if too big), are the MIR that are easier to work with.

C. Vectorization

The last step is to vectorize these MIR with a tool like Word2vec (as discussed in IV-B2). The latter takes a text as input which is a list of legal words from a dictionary $\mathcal{D} \in \mathcal{A}^{\mathbb{N}}$ constructed from an alphabet $\mathcal{A}$. In natural languages, words are separated with spaces, which is not how most programming languages are built. This is why we needed to define the concept of "token". A token is, like a word, built from the alphabet and an element of the list of tokens which will represent the code.
The first goal will be to separate tokens from each other in order to generate a list of tokens from the initial code. (*eg* int i = f(3) $\rightarrow$ [int, i, =, f,(,3,)]). Hopefully, this task is already achieved by the parser during code compilation. We decided to use Pygment, a python multi-language parser, usually used for code highlighting. Furthermore, this module allowed us to select the different types of tokens that we want to keep and especially to get rid of code commentaries.
Now that we have a list of tokens, the goal is to turn it into a list of vectors that will be fed into a neural network. A group of solutions consist in building constant-sized blocks from the list of tokens and representing each block by a vector. A historical solution is the bag-of-word method that consists of counting for each token of the dictionary their number of occurrences in the code. The main issue is that by doing so, permutations like "i = j" and "j = i" would have the same representation although a totally different meaning . A solution is the N-grams algorithm which counts the number of occurrences of every group of N words of the dictionary. The advantage of this solution is that the vector size doesn't depend on the block size but the main drawback is that the size of a vector is $|\mathcal{A}|^N$, which is too large in our case where $|\mathcal{A}| \geqslant 1000$. Hence, we decided to use the approach detailed in IV-B2 using Word2vec.

To achieve this goal, we used the CBOW (Continuous Bag Of Words) implementation from GenSim which goal is to predict a token from the other tokens in its context. To do so, it uses a fully connected neural network with one hidden layer which values will be used for the embedding. Thus, one can choose the size of the projected vectors by choosing the number of neurons in this hidden layer. Figure 8 is a crop of such an embedding space projected in 2D for visualisation using t-SNE. The Word2vec was trained by us to learn the 1000 most-used tokens on generic codes from Github. One can witness that indeed, tokens which tend to appear in same contexts are projected close to each other ($const$ and &, $case$ and $break$, similar brackets etc.)

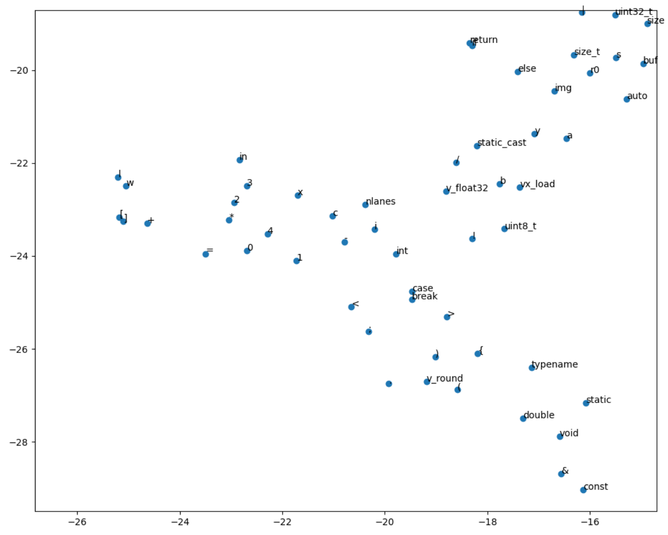


Fig. 8. Word2vec learnt on generic C++ code frome github

Now we have a vector representation of each token of the training set. There is still an issue when dealing with out-of-vocabulary tokens : tokens that didn't appear in the training set and are hence not considered part of the dictionary. This is typically the case with variable names. In our work, we decided for a first approach, to not consider out-of-vocabulary tokens. However, an easy solution to this problem when using tools like Word2Vec is to encode these tokens using the mean of the encodings of each occurrence of them based on their contexts. By doing so, we make sure to have every occurrence encoded with the same vector and to keep the vector space coherent.

D. Training Speed Optimisation

One problem we encountered is the speed of the training of the CNN. Indeed, most of the time taken by the training was not because of neural network computation, but memory access and convertion of tokens into vectors.

The former can be explained by the fact that input codes were stored in independant files, which meant assembling a batch of 100 codes required accessing 100 files dispersed in disk memory, taking a lot of time.

The convertion of tokens into vectors is also long because it requires searching the vector value of the hundreds of tokens contained in a code in a lookup table that contains hundreds of entries.

To solve this issue, we decided to convert our whole dataset into vector representations, and to store them by batch : each file contains a batch of 100 vector representations. This removes the need to convert token to vectors on the fly and reduces by 100 the amount of locations that the program

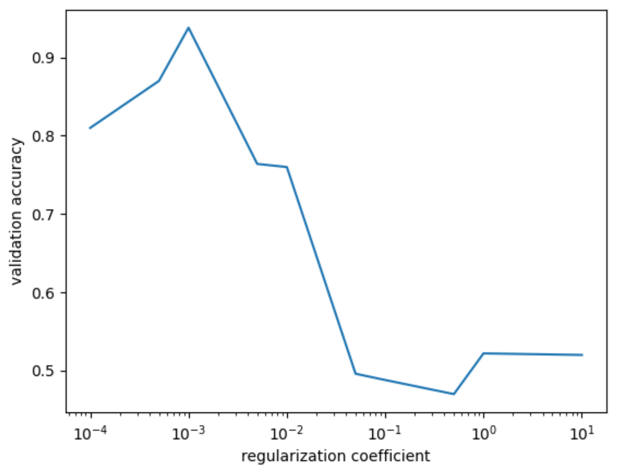


Fig. 9. Influence of the regularization coefficients on the validation accuracy

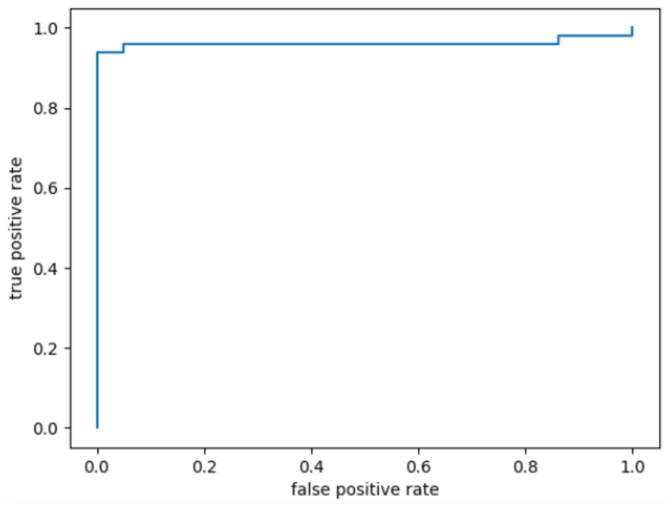


Fig. 10. ROC curve of our implementation on the generated test set

needs to access. This method almost eliminates the time taken by these two steps, and the neural network computation now represents the majority of time spent. Reducing the learning time allowed us to quickly experiment with the CNN, including testing different parameters.

E. Results

Having a working Proof of Concept (PoC) for our work was a very hard step because we faced a lot of overfitting (almost 100% accuracy on the training set and coin flip on the test set). The first step to deal with this issue was to add l1 and l2 regularizations on each single layer of the network. Further experiences revealed that each of them is crucial since getting rid of just one of them brings the overfitting back. The latter seem to help preventing weights explosion since only small coefficients are necessary as we can see when observing their influence on the learning (Figure 9). Then we proceeded to a lot of fine-tuning in order to find a working set of parameters

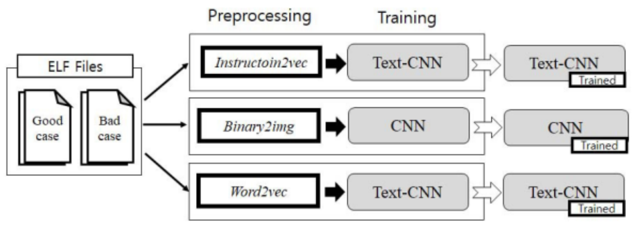


Fig. 11. [3] framework design for training

among batch-size, learning-rate and regularization coefficients. We managed to have very good results with a **recall and an accuracy of 94%** on 1000 examples from the test set (see ROC curve Figure 10).
Our concern is that we might be overfitting on the hyperparameters choice because it could work only specifically for our self-generated dataset and not scale correctly to real-life examples.

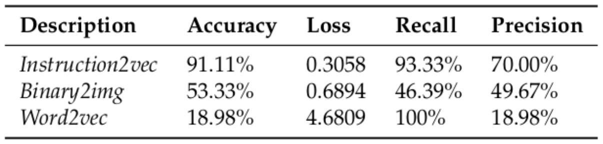
Our codes can be found on Github :

https://github.com/trasen1003/BOF_ML

VI. Code representations and future work

A. Assembly code

The work we presented until now deals with source code only. A huge issue when doing so is that the latter is often not available since companies often provide the compiled software only. Therefore, many PoC are built only for open-source projects like [3] with Mozilla Firefox. To be able to have more impact, building a tool based on other representations of the code would be an interesting approach. Compiled program analysis and especially assembly code analysis was therefore our goal in the first place. The approach wouldn't be very different from the one we presented.
Y. Lee et al [3] compare three approaches which differ only by the vectorization pre-processing since they all use a CNN afterward (Figure 11). The "Word2vec" pre-processing is very close to the one we used on source code and the "Img2vec" one consists in turning the binary file into an image by grouping each 3 consecutive bytes to form an RGB pixel and using a classical CNN image classifier. Their conclusion is that both these approaches fail to force the learning to take the specific syntax of assembly code (opcode, operand 1, operand 2) into account and hence, lead to massive overfitting. Therefore, they propose the "Instruction2vec" solution which consists in grouping word2vec-projected vectors of the opcode and both operand of each instruction in the same vector. By doing so, they orient the learning to adapt to the assembly code instructions structure.
The results of the "Instruction2vec" approach (Figure 12) are really good on a dataset built from 15 vulnerable codes from "CWE-121: Stack-based Buffer Overflow" along with 64 safe codes. These results are very encouraging to try to work with other representations than source code.



Description	Accuracy	Loss	Recall	Precision
Instruction2vec	91.11%	0.3058	93.33%	70.00%
Binary2img	53.33%	0.6894	46.39%	49.67%
Word2vec	18.98%	4.6809	100%	18.98%

Fig. 12. Results from [3] for each framework

B. Code Property Graphs

Graphs are an other quite usual way to represent codes (Figure 13) and some of them are even generated as an intermediate representation by the compiler. We already talked a lot about PDGs which is one of the most used along with Abstract Syntax Tree (AST) and Control Flow Graphs (CFG). As [4] mentions : "*Each of them provide a unique view on source code emphasizing different aspects of the underlying program*". The idea of [4] is to combine each of these 3 representations into one structure named Code Property Graph (CPG). Transversals are functions $\mathcal{T} : \mathcal{P}(V) \rightarrow \mathcal{P}(V)$ that maps a set of nodes to another set of nodes according to some predefined rules and based on the CPG. Transversals can be very simple to define but by combining them, one can detect programming patterns that can lead to security issues. For example, every call to a $memcpy$ having a size field filled with a user-controlled and unsanitized data. This approach is already widely used and could be improved by either learning proper transversals or using CPGs directly as an other representation of the code. If the former would require a huge amount of work, the latter could be an easier way to extend our work. Indeed, the vectorization step could use Node2vec, a tool already widely used in computational biology. By adding the CPG step, we could orient the learning toward this specific representation like [4] did with "Instruction2vec".

C. Future Works

Together with the different code representations that we would have tried to use if we had more time, some aspects were abandoned and could be tackled in future work :

- Expanding our work to real-life examples.
- Better understanding the learning process.
  We solved the overfitting issue at the very end of the project so we didn't have time to play with all the parameters of the learning. By doing so, we could better understand what works fine and why in order to be prepared for more generalisation.
- Classifier learning.
  [1] suggested to replace the max pooling and the dense layer by other more robust classifiers (see IV-B3) but we didn't implement that part.

D. Conclusion

In this article we analysed many different automatic BOF detection methods and focused on the work of X. Li et al [1]. We built our own implementation of the latter and detailed the choices we made during this process, especially the automatic dataset generation, the security slices generation and the training speed optimization. We managed to have some very good results, even if we fear a lack of generalisation to a real-world dataset.
Finally, we questioned the choice of working on source code and had a reflection on other types of representations of the code that could be used in similar approaches.

References

[1] X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen, 'Automated Vulnerability Detection in Source Code Using Minimum Intermediate Representation Learning', Applied Sciences, vol. 10, p. 1692, Mar. 2020, doi: 10.3390/app10051692.

[2] M. Papaevripides and E. Athanasopoulos, 'Exploiting Mixed Binaries', ACM Trans. Priv. Secur., vol. 24, no. 2, pp. 1–29, Feb. 2021, doi: 10.1145/3418898.

[3] Y. Lee, H. Kwon, S.-H. Choi, S.-H. Lim, S. H. Baek, and K.-W. Park, 'Instruction2vec: Efficient Preprocessor of Assembly Code to Detect Software Weakness with CNN', Applied Sciences, vol. 9, no. 19, Art. no. 19, Jan. 2019, doi: 10.3390/app9194086.

[4] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, 'Modeling and Discovering Vulnerabilities with Code Property Graphs', in 2014 IEEE Symposium on Security and Privacy, San Jose, CA, May 2014, pp. 590–604, doi: 10.1109/SP.2014.44.

[5] H. Shahriar et M. Zulkernine, 'Classification of Static Analysis-Based Buffer Overflow Detectors ', Secure Software Integration and Reliability Improvement Companion, IEEE International Conference on, vol. 0, p. 94-101, june 2010, doi: 10.1109/SSIRI-C.2010.28.

[6] W. Le et M. L. Soffa, 'Marple: a demand-driven path-sensitive buffer overflow detector ', in Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16, Atlanta, Georgia, 2008, p. 272, doi: 10.1145/1453101.1453137.

[7] A. Sotirov, 'Automatic Vulnerability Detection Using Static Source Code Analysis ', MSc thesis, The University of Alabama, 2005.

[8] Flawfinder. https://dwheeler.com/flawfinder/

[9] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting Format String Vulnerabilities with Type Qualifiers," presented at the 10th USENIX Security Symposium (USENIX Security 01), 2001, Accessed: May 05, 2021. [Online]. Available: https://www.usenix.org/conference/10th-usenix-security-symposium/detecting-format-string-vulnerabilities-type-qualifiers.

[10] R.-G. Xu, P. Godefroid, and R. Majumdar, "Testing for buffer overflows with length abstraction," in Proceedings of the 2008 international symposium on Software testing and analysis, New York, NY, USA, Jul. 2008, pp. 27–38, doi: 10.1145/1390630.1390636.

[11] J.-E. J. Tevis and J. A. Hamilton, "Static Analysis of Anomalies and Security Vulnerabilities in Executable Files," in Proceedings of the 44th Annual Southeast Regional Conference, New York, NY, USA, 2006, pp. 560–565, doi: 10.1145/1185448.1185570.

[12] A. A. Jorgensen, "Testing with Hostile Data Streams," SIGSOFT Softw. Eng. Notes, vol. 28, no. 2, p. 9, Mar. 2003, doi: 10.1145/638750.638781.

[13] A. Tappenden, P. Beatty, J. Miller, A. Geras, and M. Smith, "Agile security testing of Web-based systems via HTTPUnit," Aug. 2005, vol. 0, pp. 29–38, doi: 10.1109/ADC.2005.11.

[14] C. Del Grosso, G. Antoniol, E. Merlo, and P. Galinier, "Detecting buffer overflow via automatic test input data generation," Computers & Operations Research, vol. 35, no. 10, pp. 3125–3143, Oct. 2008, doi: 10.1016/j.cor.2007.01.013.

[15] P. McMinn, "Search-based software test data generation: a survey," Software Testing, Verification and Reliability, vol. 14, no. 2, pp. 105–156, 2004, doi: https://doi.org/10.1002/stvr.294.

[16] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in Proceedings of the 7th symposium on Operating systems design and implementation, USA, Nov. 2006, pp. 147–160, Accessed: May 05, 2021. [Online].

[17] M. F. Ringenburg and D. Grossman, "Preventing Format-String Attacks via Automatic and Efficient Dynamic Checking," in Proceedings of the 12th ACM Conference on Computer and Communications Security, New York, NY, USA, 2005, pp. 354–363, doi: 10.1145/1102120.1102166.

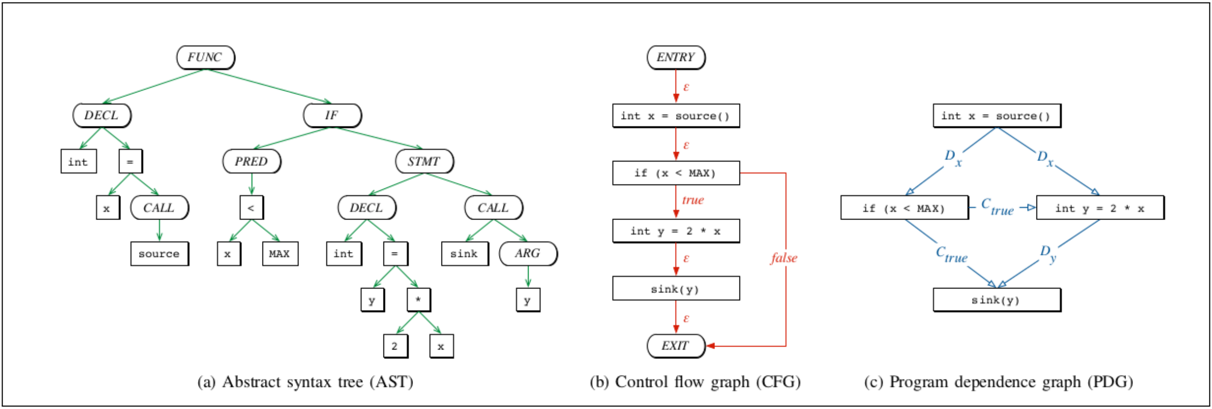


Fig. 13. Three different types of graph representations for the same code

[18] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?," Empirical Software Engineering, vol. 18, 2011, doi: 10.1007/s10664-011-9190-8.

[19] B. Livshits and T. Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," in Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, 2005, pp. 296–305, doi: 10.1145/1081706.1081754.

[20] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities using Machine Learning," Aug. 2011.

[21] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional Neural Networks over Tree Structures for Programming Language Processing," arXiv:1409.5718 [cs], Sep. 2014, Accessed: May 05, 2021. [Online]. Available: http://arxiv.org/abs/1409.5718.

[22] Z. Li et al., "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," Proceedings 2018 Network and Distributed System Security Symposium, 2018, doi: 10.14722/ndss.2018.23158.

[23] R. L. Russell et al., "Automated Vulnerability Detection in Source Code Using Deep Representation Learning," arXiv:1807.04320 [cs, stat], Nov. 2018, Accessed: Apr. 19, 2021. [Online]. Available: http://arxiv.org/abs/1807.04320.

[24] H. Shahriar and M. Zulkernine, "Mitigating program security vulnerabilities: Approaches and challenges," ACM Comput. Surv., vol. 44, no. 3, pp. 1–46, Jun. 2012, doi: 10.1145/2187671.2187673.

[25] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," arXiv:1301.3781 [cs], Sep. 2013, Accessed: May 07, 2021. [Online]. Available: http://arxiv.org/abs/1301.3781.

[26] L. Szekeres, M. Payer, Tao Wei, and D. Song, 'SoK: Eternal War in Memory', in 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, May 2013, pp. 48–62, doi: 10.1109/SP.2013.13.