Using Machine Learning for Vulnerability Detection and Classification

Tiago Baptista
Centro Algoritmi, Departamento de Informática, University of Minho, Braga, Portugal

Nuno Oliveira
Checkmarx, Braga, Portugal

Pedro Rangel Henriques
Centro Algoritmi, Departamento de Informática, University of Minho, Braga, Portugal

Abstract

The work described in this paper aims at developing a machine learning based tool for automatic identification of vulnerabilities on programs (source, high level code), that uses an abstract syntax tree representation. It is based on *FastScan*, using *code2seq* approach. *Fastscan* is a recently developed system aimed capable of detecting vulnerabilities in source code using machine learning techniques. Nevertheless, *FastScan* is not able of identifying the vulnerability type. In the presented work the main goal is to go further and develop a method to identify specific types of vulnerabilities. As will be shown, the goal will be achieved by optimizing the model's hyperparameters, changing the method of preprocessing the input data and developing an architecture that brings together multiple models to predict different specific vulnerabilities. The preliminary results obtained from the training stage, are very promising. The best *f1* metric obtained is 93% resulting in a precision of 90% and accuracy of 85%, according to the performed tests and regarding a trained model to predict vulnerabilities of the injection type.

2012 ACM Subject Classification Security and privacy → Vulnerability scanners; Computing methodologies → Machine learning

Keywords and phrases Vulnerability Detection, Source Code Analysis, Machine Learning

Digital Object Identifier 10.4230/OASIcs.SLATE.2021.14

Funding This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

Acknowledgements Special thanks to Search-ON2: Revitalization of HPC infrastructure of UMinho, (NORTE-07-0162-FEDER-000086), co-funded by the North Portugal Regional Operational Programme (ON.2-O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF).

1 Introduction

Nowadays, information systems are a part of almost every aspect in life and furthermore, almost every company is dependent on the liability, safety and security of a software application. So, it is essential to have the capability to identify and correct pieces of code that contain known vulnerabilities in order to prevent the software from being compromised.

Cybernetic attacks are a constant and present a real threat to companies and people in general, since nowadays almost every device has an Internet connection. As a consequence, devices are exposed to external threats that try to exploit vulnerabilities.

A vulnerability is a flaw or weakness in a system design or implementation (the way the algorithms are coded in the chosen programming languages) that could be exploited to violate the system security policy [13]. There are many types of vulnerabilities and many approaches to detect such flaws and perform security tests. All the known approaches present

© Tiago Baptista, Nuno Oliveira, and Pedro Rangel Henriques;
licensed under Creative Commons License CC-BY 4.0

pros and cons but none of them stands as a perfect solution. Static analysis is one of the approaches and it can be defined as the analysis of a software without its execution. Static Application Security Testing (SAST) is an application security testing methodology that allows detecting vulnerabilities at the early stages of software development. It is adopted by many companies, such as Checkmarx[1]. These methodologies have many strengths such as the ability to find vulnerabilities without the need to compile or run code, offering support for different programming languages and being able to easily identify common vulnerabilities and errors like Structured Query Language (SQL) injections and buffer overflows. Despite this, there are still problems with the referred approach, mainly in the production of a great number of false positives, in the lack of identification of the vulnerability type and even performance issues.

There are many tools that implement the concept of static analysis and apply it to vulnerabilities detection. Some tools rely on only lexical analysis like *FlawFinder*[2] [10] but have the tendency to output many false positives because they do not take into account the semantic [4]. Other tools like *CxSAST*, Checkmarx SAST tool[3], overcome this lack by using the Abstract Syntax Tree (AST) of the program being evaluated. In this context, another challenge is to create and apply the same tool to different languages, so that one can clearly identify vulnerabilities with high accuracy and have good performance with big inputs.

To overcome the flaws identified in the *SAST* approach, decreasing the number of false positives and the processing time, a new approach came to the researchers mind: to integrate machine learning techniques. The idea can be realized by altering and tuning open source projects, namely *code2vec* and *code2seq* in order to try to identify accurately and efficiently than other tools like *CxSAST* [9]. *code2vec* main idea is to represent a code snippet as a single fixed-length code vector, in order to predict semantic properties of the snippet. On the other hand *code2seq* represents a code snippet as a set of compositional paths and uses attention mechanisms to select the relevant paths while decoding [1, 2]. The resultant approaches relying on *code2seq* and *code2vec* are called *FastScan* and were not one hundred percent success but opened the path to further investigation [9].

It is clear that a good analysis tool can help spot and eradicate vulnerabilities, furthermore, it is becoming a part of the development process. But, there is still room for improvement and all the research work done in this area can be of uttermost relevance for the industry.

With all of the previous in consideration the main contributions expected from the research project that will be discussed along the paper are:

- **The improvement of the *FastScan* approach**:
  - Find the best hyperparameters for each case study;
  - Use these hyperparameters to improve evaluation metrics for the models (precision, recall and *f1*);
- **The development a specific model for each type of vulnerability**, that is capable of identifying if a code snippet has a vulnerability or not (Boolean model) of a given type;
- **The design of a proper architecture to develop a general model** capable of identifying the occurrence of vulnerabilities and their type, given a code snippet;

Since it has different objectives and it is an evolution of *FastScan*, from now on, when referring the work, it will be called ***new FastScan***.

[1] https://www.checkmarx.com/
[2] https://dwheeler.com/flawfinder/
[3] https://www.checkmarx.com/

This paper is divided in seven sections, in the first section will be presented the context and motivation, in the second section the current state of the art in vulnerability detection, the third section will detail the proposed approach, including architecture, the fourth section with more details about the developed work, the fifth section refers to the specification for the first phase (that will be presented in Section 3), then section six will approach the results and the last section with conclusions and future work.

2 Vulnerability Detection

The main spark that ignited the work described in this paper, and the work from specialists around the world, is software vulnerability. It is necessary to understand the concept of vulnerability and other main concepts related to them in order to understand all the methodologies presented next. A vulnerability is defined as a flaw or weakness in a system design, implementation, or operation and management that could be exploited to violate the system's security policy [12]. These flaws might be system design, development, or operation errors and its exploit can lead to harmful outcomes. This perspective of harm or loss of information is normally called risk. Nowadays, these concepts have become common in any software development process because with the presence of electronic devices running software in almost every area of our society, there is the need to eliminate, or at least minimize, the occurrence of situations that can risk the security of data or operations.

There are different types and classifications of vulnerabilities, provided by different contributors. They expose discovered vulnerabilities, exploits, and solutions in online databases[4]. Even though there are many sources and knowledge about vulnerabilities, exploits still occur.

Open Web Application Security Project (OWASP)[5] foundation, that works towards a more secure software development process and applicationss, has a list of the top ten security risks on applications. Since one of the objectives of this paper work is to correctly identify types of vulnerabilities using machine learning models, first it is necessary to know them and understand how they use security breaches to harm systems.

Since this top ten is intended to identify the most serious security risks on web applications, and it is important to know them in-depth in order to know how to identify them, manually or automatically, and even more realise what consequences can they bring to the system. Since the results presented are related to a specific vulnerability it is important to know it more deeply.

2.1 Injection attack

An injection attack refers to an attack where untrusted data is supplied as input to a program. This input is then processed and changes the application's expected behaviour. Normally, this vulnerability is related to insufficient user input validation. Since this is a well known and one of the oldest exploits, that has some automatic tools in order to exploit without having much knowledge, makes this one of the most common and dangerous vulnerability. There are many types of injections, namely Code injection, Email Header Injection, Host Header Injection, Lightweight Directory Access Protocol (LDAP) Injection, SQL injection among others.

[4] https://cve.mitre.org/
[5] https://owasp.org/www-project-top-ten/

Next is shown an example for SQL injection, where an attacker sends `OR 1=1--` instead of a valid id, as shown in Figure 1. Without an input validation, this query will return all data contained in the table in this case the table `accounts`.

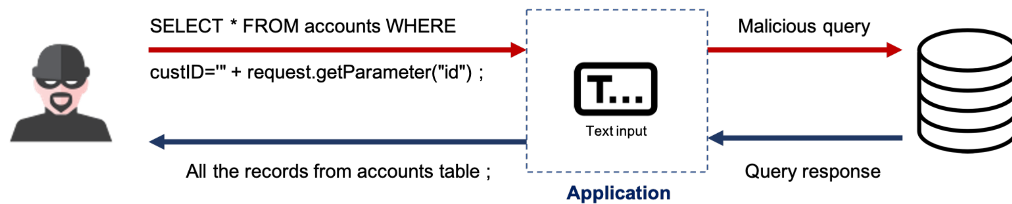

Figure 1 Schema explaining injection.

In many of these affected software development projects, there is no application of formal and systematic methods to execute source code auditing and testing. Auditing can be defined as the process of analyzing application code (in source or binary form) to uncover vulnerabilities that attackers might exploit [7]. This process is important because allows scanning the source code covering all the paths that normal testing might not cover in order to try to detect possible vulnerabilities.

2.2 Detection techniques

Having in mind the big issue in software development presented before, there are many solutions that try to eliminate it by making an in-depth code auditing. Starting with the concept of defensive programming, that refers to the practice of coding having in mind possible security problems that can occur [5]. This is clearly a good practice but one that does not solve the vulnerability related problems. Even good and experienced programmer cannot know how to prevent all the exploits created for certain Application Programming Interface (API), library or other. Also, languages by construction were not build thinking on the way an attacker would take advantage of certain nuances. Taking as an example, the buffer overflow exploits, one the most common problem reported in vulnerabilities databases. This can occur due to mishandling of inputs but in its core is allowed by the construction of the language. Has it is easily confirmed by C or C++ that do not provide any built-in protection against accessing or overwriting data in memory [6]. Since good intentions and practices are not enough to solve this problem, there was the need to apply and develop other techniques that will be briefly presented next:

- **Static analysis for security testing (SAST):** Static analysis methods are a resource for identifying logical, safety and also security vulnerabilities in software systems. They are normally automatic analysis tools and intend to avoid manual code inspections in order to save time and avoid the investment of resources in manual tasks that could be fruitless [11].
- **Dynamic analysis for security testing (DAST):** In order to find vulnerabilities, testing seems to be the easiest path to follow, and that is what DAST stands for. In DAST, a program is executed in real-time in order to test and evaluate it. In DAST the tested software is seen as a black box, and the tools or person performing the tests only interact with the application or software as users that have no knowledge of its internal operations.
- **Interactive analysis for security testing (IAST):** Contrarily to DAST, IAST aims to find vulnerabilities by analysing software from within such as SAST. But contrarily to SAST and similarly to DAST, IAST executes the analysis with the software running.

IAST uses different instruments to monitor a running application in order to gather information about the internal processes. These tools try to mitigate SAST's and DAST's limitations, namely, identify the specific place where a bug/vulnerability is located. By running from the software's inside has leverage over DAST and allows testers to define what paths or functionalities to cover, this can lead to miss-handled bugs/vulnerabilities but if well thought can lead to gains in time and work efficiency contrarily to SAST that has full coverage over the software.

- **Machine Learning in vulnerability identification:** Artificial intelligence and more specifically machine learning and deep learning systems are being used to automate many different tasks and in different areas, with success. For example in image recognition, disease behaviour prediction, traffic prediction, virtual assistant, among others. The area of software security is not an exception, as referred on the previous sections, current vulnerability identification tools and in general, all security tools have flaws and many rely on a great amount of manpower and are very time-consuming. In order to try to tackle such limitations and with the rise of many machine learning applications, there were many investigation focused on applying such techniques in the software security area. There are many examples of such applications, but for this paper purposes, the next sections focus on the application of machine learning and deep learning for vulnerability identification [3].

3 Proposed approach

In this section, it is detailed the proposed approach in order to fulfil the objectives detailed in Section 1. It will be presented the system architecture and a high-level representation of the system flow of data in order to better perceive all the involved processes.

3.1 System Architecture

In order to understand why *code2seq* was chosen in *Fastscan* and remained in *new FastScan* it is first necessary to understand *code2seq* structure and applications. *code2seq* creates a representation of source code using AST and then uses it to infer properties. An AST represents a source code snippet in a given language and grammar. The leaves of the tree are called terminals and the non-leafs are called non-terminals. It represents all the variables declarations, operator, conditions and assignments. In order to represent code in this structures it is necessary to create sequences of terminal and non terminal nodes and consider all possible paths between nodes.

This representation has some significant advantages over the use of simple code tokenisation, when compared in terms of code comparison. Namely when trying to find two methods that have the same functionality but different implementations. Having the AST enables a better comparison, since both functions paths will be similar, as represented in Figure 2 . So functions will have different token representations but similar path representation only differing in the *Block* statement.

In a simplified overview *code2seq* uses encoder-decoder architecture that reads the AST paths instead of the tokens. In the encoding end, there is the creation of vectors for each path using a bi-directional Long short-term memory (LSTM) and the extraction of tokens, where the AST terminal nodes are transformed into tokens and these tokens are split into subtokens (for example, an ArrayList is transformed into the tokens Array and List). In the end a decoder uses attention models to select relevant paths.

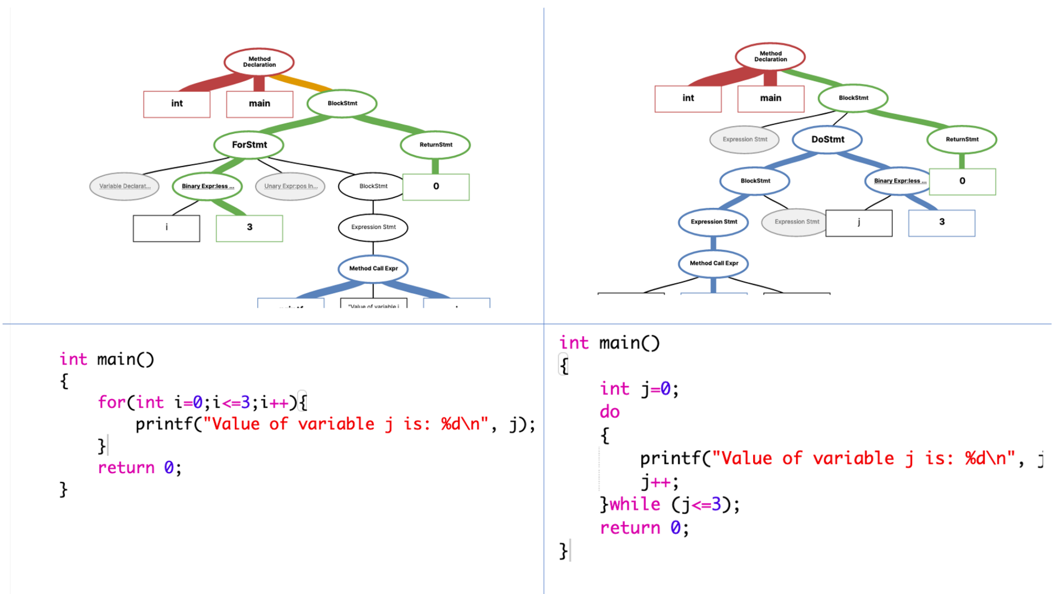

Figure 2 Clarification *code2seq* AST representation.

To accomplish the development of a specific model for each type of vulnerability, that is capable to identify if code snippet has a vulnerability or not (Boolean model), it is proposed to follow the architecture and flow presented in Figure 3. This approach is a refinement of Ferreira's work with *code2seq* named *FastScan* [9].

In the first phase, it is essential to use *CxSAST* in order to obtain the input for the preprocessing – since it converts several languages into a generic internal representation, allowing to create a tool language independent. Since the focus is on creating a model capable of identifying certain vulnerabilities then it is necessary to filter *CxSAST* output in order to correctly train the model. Then use *code2seq* normal pipeline in order to obtain a model able to identify a specific vulnerability. The goal is to have at least a model for each of *OWASP* top 10 vulnerabilities.

Having completed the first phase, it is necessary to develop a general model capable of identifying if there are vulnerabilities and it's type, given a code snippet. In order to complete this objective it is proposed to follow the architecture and flow presented in Figure 4. In order to take advantage of the work done in the first approach it is proposed to use the previously trained models and combine them. This combination might be done using ensemble technique or only by running *code2seq* testing phase in parallel.

In order to obtain the best performance wise results, it is expected to test the developed pipeline using the most powerful hardware resources available at the University of Minho's cluster[6].

[6] http://www4.di.uminho.pt/search/pt/equipamento.htm

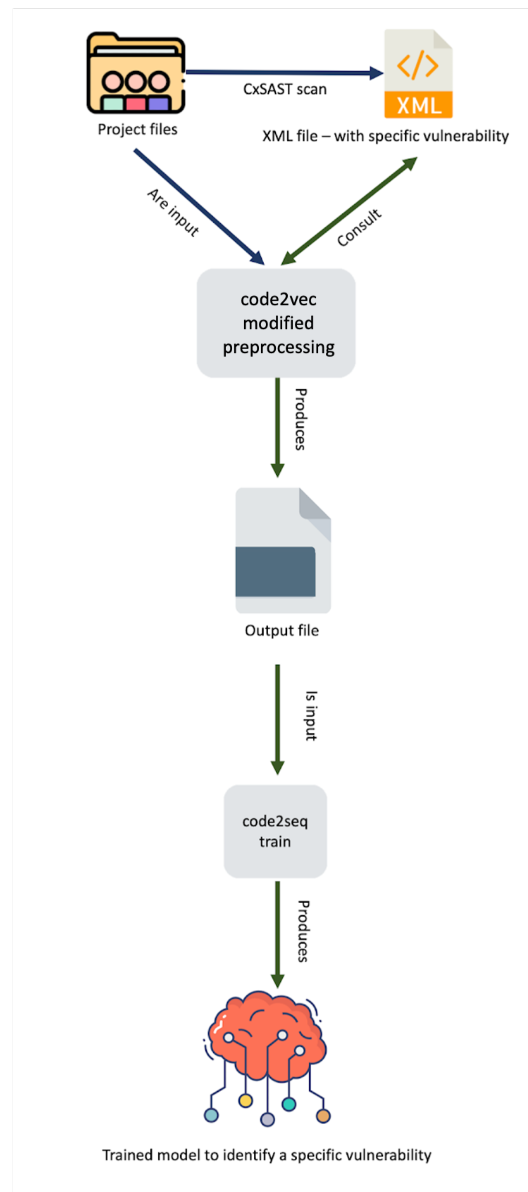

Figure 3 First phase architecture.

4 Development

The work that led to this paper, *FastScan*, had promising results and showed the potential on using machine learning for detecting vulnerabilities in source code having as a base the open source project *code2seq*. Nevertheless, it left open for investigation some issues, namely the ones that are a part of this paper objectives. To tune *code2seq* parameters and to develop a way to identify specific types of vulnerabilities in source code.

To address this issues, the first phase began with the the *FastScan* code as the base. On which it was build a solution that it is able to produce a model capable of identifying if there are vulnerabilities of a specific type given a source code input. The conceptual approach is the same as in *FastScan* but with the modifications to direct the prediction to a specific type of vulnerability and it was done the hyperparameters optimization, applied to the input dataset, in order to improve the model accuracy.

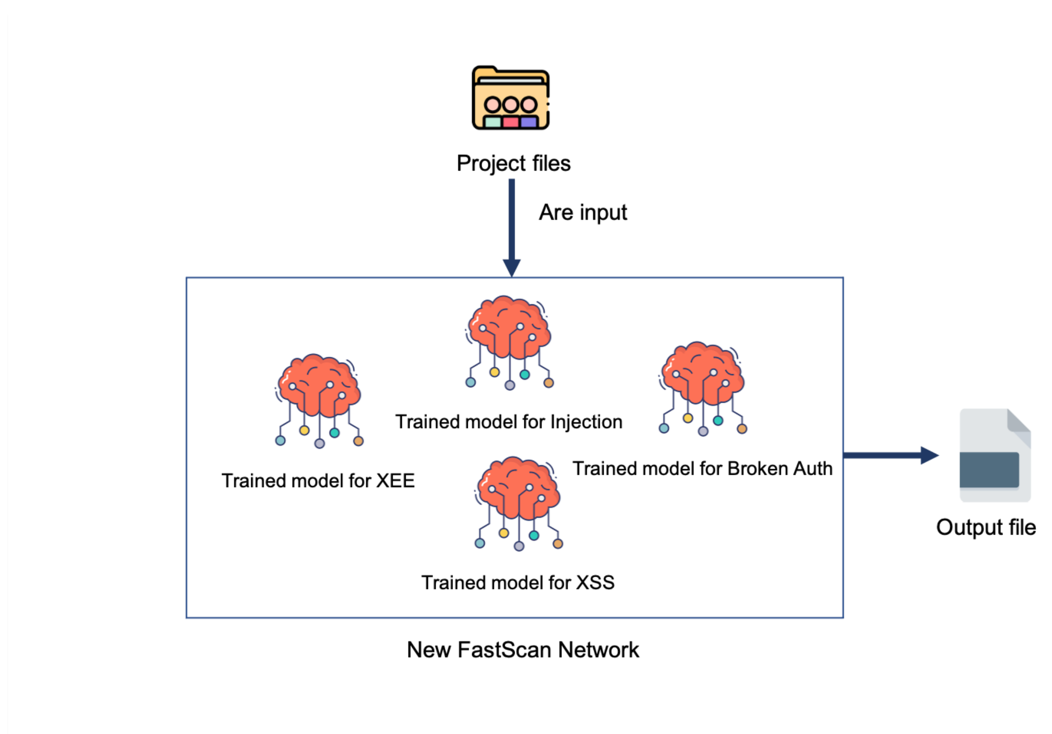

Figure 4 Second phase architecture.

With all this in mind, this section will describe the different phases that were explored during the presented work. The first section will describe the datasets, then the hardware and technical details on which the experiments were made and the following sections will detail the work done in the first phase. Explaining the steps in each one, the challenges, experiments and the obtained results for further analysis in the next section.

4.1 Datasets

The datasets take a major part in this project, because all the developed work has no utility unless there is enough good data to train the models.

The first dataset which, from now on, will be referenced as *dt01*. *dt01* is composed by **43** different projects, there is the original source code and for each project there is an XML file with detected vulnerabilities. This XML file was provided by *Checkmarx* and it is the output of their static analysis tool *CxSAST*. With one important detail, the output was validated by humans which means that there are no false positives.

4.2 Hardware and technical details

Since in *FastScan*, the low computational power was identified has a major barrier and setback to the developed work, since preprocessing and training tasks were slow and implied much waiting time, it became clear that the use of a regular personal computer was not enough to achieve the desired results in the experiments of this paper [9].

This barrier was overcame with the use of the University of Minho cluster[7]. More specifically using a node with the following specifications:

- **CPU as seen in Figure 5:**
  - Intel® Xeon® Processor E5-2695 v2[8];
  - twelve cores ;
  - twenty four threads.
- **RAM:** Sixty four GB as seen in Figure 6 ;
- **GPU:** Two NVIDIA® TESLA® K20M[9].

```
vendor_id       : GenuineIntel
cpu family      : 6
model           : 62
model name      : Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
stepping        : 4
microcode       : 0x428
cpu MHz         : 1228.500
cache size      : 30720 KB
```

Figure 5 CPU details.

```
[a75328@compute-662-1 ~]$ cat /proc/meminfo
MemTotal:        65769424 kB
```

Figure 6 Memory details.

There were problems installing all the needed dependencies to run *new FastScan* natively, because of the security constraints in the cluster system. Namely, there were no root permissions, so the installation of dependencies such as *Java*, python and its libraries were a great technical obstacle. In order to overcome this, it was used docker and all its potentialities, since docker presented a good solution because it is consistent and makes all this installation process easy by using a base image with *Java* and adding all the dependencies needed without further concerns.

It is important to notice that this solution was followed after the confirmation that there was no significant overhead or performance loss introduced by the use of docker and that it was a better solution than a virtual machine [8].

5 First Phase

In this section is explained the first phase referred previously in Section 3.1. This phase intends to receive as input a certain dataset containing *Java* projects and an Extended Markup Language (XML) file from CxSAST containing identified vulnerabilities to output a trained model for a specific and predetermined vulnerability .

5.1 Data filtering

This step was of great importance and it was not present in *FastScan*. Since the objective of new *Fastscan's* the first phase is to create a model capable of detecting a specific vulnerability, it is firstly needed to filter the original dataset input data by vulnerability type. So that it is possible to train a model to predict only a specific type of vulnerability.

[7] http://search6.di.uminho.pt
[8] https://ark.intel.com/content/www/us/en/ark/products/75281/intel-xeon-processor-e5-2695-v2-30m-cache-2-40-ghz.html
[9] https://www.techpowerup.com/gpu-specs/tesla-k20m.c2029

Before knowing how it was achieved, it is important to understand the two different components from the dataset:

- Source code files: This are the original files from several different projects;
- XML files: This are the files generated by the *CxSAST* from *Checkmarx*. There is one file for each project, on which are registered the vulnerabilities detected by the tool.

This filtering process was developed using *python* and aims to filter the XML files of each project, in order to keep only the ones that refer to the vulnerability that is desired to train a model. In order to achieve this, it was developed a python program that is able to read and parse the XML (provided by *CxSAST* – and it is built to be applied to its specific structure) and filter the vulnerabilities entries by the vulnerability name. It is not a literal search, it can be send arguments to the program with the range of words/strings to be searched in the vulnerability name field of the XML entries. In the end, the output is a set of XML files only with the entries for a specific vulnerability and ready to be ingested in the next step – Preprocessing.

5.2 Preprocessing

This step was not changed from the *FastScan*, but it is different from the original *code2seq*. The preprocessing (which is performed on the train and test data) returns a file with the dataset labeled and the *AST* that represents the input dataset. The source code files and XML files are parsed , in the preprocessing, which is built in *Java*. There is the creation of the *AST* from the input of the source code files and this was not modified from the original *code2seq*. But there is another step required to obtain the label for each method, the parsing of the XML files.

There is also performed the parsing from the *XML* file (a new step added in *New FastScan*). In this second it is relevant to refer that it is stored the vulnerabilities registered in the file, namely the name of the vulnerability, the start and end line and column in the file and also the filename and path within the project.

After the parsing stage, for each method (that resulted from the source code parsing) there the verification if it occurs in the register of vulnerabilities obtained by the *XML* parsing, the search is achieved by comparing the filename and path within the project which is stored in the result of the *XML* and also the source code parsing. Then it is registered the presence or not of vulnerabilities, with a *boolean*.

This combination of the parsing of the *XML* (with the results of the *CxSAST*) and the original source code leads to the output of the preprocessing, that is constituted by a text file that has a line for each method, on which the first element is the label indicating the presence of vulnerabilities followed by the *ast* paths.

On the original *code2seq* preprocessing, in the output file the first entry was the function's name – the first entry is the label that will be used in the training phase. But *New Fastscan* preprocessing was modified in order to make the first entry of each line a boolean that indicates the presence or not of a vulnerability in the method. This change was enough to modify the prediction of the model obtained in the next training phase, since the training is now gonna be done with the boolean has the prediction label instead of the previous string that was the method name.

5.3 Hyperparameter optimization

This step was important to tackle a problem detected in *FastScan*, without the optimization of the hyperparameters the final model performance might not be the best possible [9].

In order to understand the following section, firstly it is important to know the notions of accuracy, precision, recall and *f1*.

- Accuracy is the most known performance measure and it is the result of a ratio between correctly predicted observation to the total observations. But it can be misleading if there are not the same number of false positives and false negatives, so it is important to explore other metrics;
- Precision refers to the ratio between correctly predicted positive observations and the total predicted positive observations. This metric answers the question of from the methods labeled as having a vulnerability, how much of them actually had one? A high precision means a low occurrence of false positives;
- Recall is often called sensitivity, it answers the question of, from the methods that really had a vulnerability how many of them were labeled?;
- The *f1* metric is the weighted average between precision and recall, this is the most balanced meausure of all the preseted since takes false positives and false negatives into account. This metric is usefull when there is uneven distribution in the dataset, in the case, where there is not the a close number between vulnerable and not vulnerable entries.

To do so, it was used *wandb*[10] – It is a python library, used to monitor, compare, optimize and visualize machine learning experiments. It can be integrated with many frameworks including *tensorflow* (the python library used in the training) and it was of great importance in the development of this work. It implements different search methods in order to obtain the best parameters to increase a specific evaluation metric. This library allowed to solve the hyperparameter tuning which is a very complicated problem that normally requires experience in the field and complicated algorithms [14].

This library allowed the optimization of the models hyperparameters for a specific dataset by using *sweeps* that allow to find a set of hyperparameters with optimal performance and apply different methods to do so, such as grid, random and Bayesian[11]. It also allows to visualize performance and time metrics from the experiments, that can be consulted in Section 6.

The chosen method was the Bayesian method because it guaranteed the best possible combination without compromising time and performance. There were other methods such as random search where the parameters are being chosen randomly from a specific range – this method could be even faster but it would not guarantee the best hyperparameters since it does not cover all the combinations and it does not have a method to improve its results. Other method could be the grid search, in this method all the possible hyperparameter combinations are tested, in this case it would be of great time and processing cost given the number of hyperparameters.

The library applies the Bayesian method by building a probabilistic model that maps the value of each hyperparameters with the values to optimize – that could be accuracy, f1 and precision. Each hyperparameter value used in the algorithm iteration and the obtained results are taken into account into the next iterations. This way the search for the best hyperparameters is faster because it is being guided by the previous experiences.

[10] `https://wandb.ai/site`
[11] `https://docs.wandb.ai/guides/sweeps`

5.4 Train

This step referees to the effective training and production of the models. This is performed the same way has in the origial *code2seq*, relying on the developed approach that uses tensorflow in order to perform all the process.

6 Testing and Results

In this section will be presented the results obtained so far. The presented results were obtained using the dataset previously referred. The dataset was split in the following way:

- 75% to train = 32 projects;
- 15% to test = 7 projects;
- 10% to validate = 4 projects.

Regarding the data filtering step, it was applied the filter of injection, in order to filter the results for this specific vulnerability so that it is possible to train the model only to identify injection vulnerabilities.

Then the preprocessing was applied to the the source code and the filtered XML file obtaining the preprocessed input to train and test the final model. The total number of examples (paths) obtained from the preprocessing from the dataset was 191813.

The next phase was the hyperparameter optimization, firstly it was attempted to run the script to obtain the best parameters that maximize precision. After analysing the results in Table 1 it was observed that it was possible to obtain a really high precision but at the cost of very low recall. A model with high precision but low recall returns a low count of results, but most of the predicted labels are correct when compared to the training labels on the other hand a model with high recall but low precision returns a high count of results, but most of the predicted labels are incorrect when compared to the training labels. After this consideration it was clear that a vulnerability prediction system must have a balance between both because it is important to correctly identify a vulnerability but it is also important not to overlook one.

This balance can be obtained by using the *f1* metric, this metric is a weighted mean of the precision and recall and being so if in the optimization is focused in optimize the *f1* value then it is guaranteed the balance it is searched between precision and recall as seen in Equation 1.

$$f1 = 2 * precision * recall / precision + recall \tag{1}$$

So the optimization was done trying to maximize the *f1* metric. This was applied through the use of *wandb* sweeps, where it was defined that the value to optimize was the f1 as referred in Section 5.3.

After having the sweep results, a model was trained using the best hyperparameters that can be consulted in Table 1.

The final model for injection using the *dt01* dataset had 85% of accuracy, 90% of precision, 97% of recall leading to an *f1* of 93% and it was achieved in the first epoch as seen in Figure 7. This was the chosen, according to the criteria explained before, because it had the best score regarding the f1 *metric*. After this the model training leads to the increase of precision, as expected in a training process, but with the lowering of recall values and therefore a lower *f1* value.

Table 1 Best Hyperparameter for *dt01*.

BATCH_SIZE	BEAM_WIDTH	BIRNN	CSV_BUFFER_SIZE
127	0	true	104857600
DATA_NUM_CONTEXTS	DECODER_SIZE	EMBEDDINGS_DROPOUT_KEEP_PROB	EMBEDDINGS_SIZE
0	302	0.4162172547338106	193
MAX_CONTEXTS	MAX_NAME_PARTS	MAX_PATH_LENGTH	MAX_TARGET_PARTS
234	8	10	6
NUM_DECODER_LAYERS	NUM_EPOCHS	PATIENCE	RANDOM_CONTEXTS
1	3000	4	true
READER_NUM_PARALLEL_BATCHES	RELEASE	RNN_DROPOUT_KEEP_PROB	RNN_SIZE
1	false	0.7488847205115016	256
SAVE_EVERY_EPOCHS	SHUFFLE_BUFFER_SIZE	SUBTOKENS_VOCAB_MAX_SIZE	TEST_BATCH_SIZE
1	10000	151701	256
TARGET_VOCAB_MAX_SIZE			
27000			

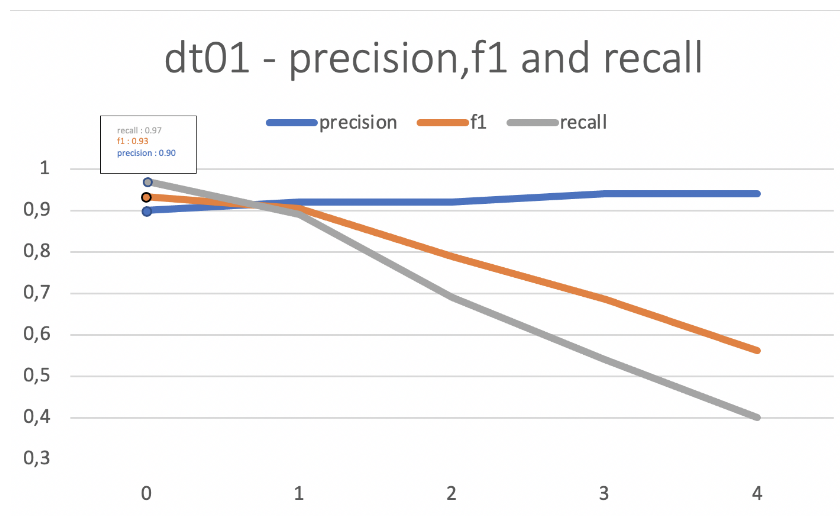

Figure 7 Model evaluation.

7 Conclusion and Future Work

This section is intended to close the paper, summarising the outcomes reached so far. The first section contains the context on vulnerability detection, the motivation and objectives of the project. The second section is a literature review on vulnerability detection. The outcomes of the reported stage provided the foundations for the proposed approach. The third section presents and discusses the our working proposal. The fourth section explains the development and includes the presentation of the dataset used for training as well as describes the hardware details. The fifth section discusses the implementation. Finally the sixth section analyzes the training results obtained when testing models.

Taking into account the results from this first experiment, it becomes clear that the hyperparameter optimization has improved the results in the increase the precision and the other metrics. Also the train only for a specific vulnerability might as well have an influence since the train for a more strict purpose is more effective, namely in this case. While *Fastscan* attempts to predict the presence of many types of vulnerabilities, *new Fastscan* aims at creating models to predict a single type of vulnerability, gathering the parts into a global analyzer in a final system.

Applying this model as a scanner that verifies projects before it goes through a *SAST*, other tool or manual verification could represent a great improve in terms of spent time, processing and manual work since it could eliminate the projects without vulnerabilities from further scanning. Comparing to the traditional tools this approach requires less results verification and promises better accuracy.

Summing up, the concept of vulnerability was presented and explained, the different methods to identify them were discussed. Moreover, the improved approach and its first promising results were described and analyzed. Also, it is important to highlight the effort that done in the adaptation and installation of algorithms and programs to run the system in parallel platform offered by the SEARCH cluster available in the Informatic Department of University of Minho to be possible to train the models in acceptable time.

The next step planned to carry on the project is the training of models for other vulnerability types in order to accomplish the second objective, as listed in the Section 1. After that, it will be necessary to implement the architecture presented in Section 3.1, in order to obtain a system capable of predicting the presence of vulnerabilities of different types.

It also is important to investigate if there is the possibility to reduce the computational cost with the mix of traditional and machine learning techniques.

References

1. Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint*, 2018. `arXiv:1808.01400`.
2. Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.
3. Philip K Chan and Richard P Lippmann. Machine learning for computer security. *Journal of Machine Learning Research*, 7(Dec):2669–2672, 2006.
4. Brian Chess and Gary McGraw. Static analysis for security. *IEEE security & privacy*, 2(6):76–79, 2004.
5. Brian Chess and Jacob West. *Secure programming with static analysis*. Pearson Education, 2007.
6. Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.
7. Mark Dowd, John McDonald, and Justin Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.
8. Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 171–172. IEEE, 2015.
9. Samuel Gonçalves Ferreira. Vulnerabilities fast scan - tackling sast performance issues with machine learning. Master's thesis, University of Minho, 2019.
10. Rahma Mahmood and Qusay H Mahmoud. Evaluation of static analysis tools for finding vulnerabilities in Java and C/C++ source code. *arXiv preprint*, 2018. `arXiv:1805.09040`.
11. Marco Pistoia, Satish Chandra, Stephen J Fink, and Eran Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2):265–288, 2007.
12. R. W. Shirey. Internet security glossary, version 2. *RFC*, 4949:1–365, 2007.
13. Robert W. Shirey. Internet security glossary, version 2. *RFC*, 4949:1–365, 2007. `doi:10.17487/RFC4949`.
14. Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *arXiv preprint*, 2012. `arXiv:1206.2944`.