MASTER PROJECT IN COMPUTATIONAL SCIENCE & ENGINEERING
MATH-598 | MATHEMATICS SECTION | EPFL

Carried out in the Computer Science & Artificial Intelligence Lab (CSAIL), Anyscale Learning For All (ALFA) Group
Massachusetts Institute of Technology (MIT), Boston

Exploring Deep Learning Models for Vulnerabilities Detection in Smart Contracts

Student

Nicolas Lesimple
Computational Science & Engineering
Mathematics Section
EPFL

Supervisors

Prof. Martin Jaggi
Machine Learning and Optimization Laboratory
EPFL

Principal Research Scientist Una-May O'Reilly
CSAIL Alfa Group
MIT

January 28, 2020

CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Una May O'Reilly, the Founder and Principal Scientist Researcher of ALFA Group at MIT-CSAIL, whose work focuses on scalable machine learning, evolutionary algorithms, and frameworks for large-scale knowledge mining, prediction, and analytics. The door to Prof. O'Reilly's office was always open. Furthermore, Una May O'Reilly is genuinely interested in the people working in her lab and enables them to flourish as researchers, by encouraging social interaction between the members of the lab, and by making teamwork and development of soft skills one of the priorities of the laboratory.

I would also like to thank a lot Eric Hemberg, who is a Research Scientist with ALFA Group at MIT-CSAIL. His work focuses on developing autonomous, proactive cyber defenses that are anticipatory and adapt to counter attacks. He also works on predicting stop-out in Massive Open Online Courses (MOOCs) and analyzing neuronal development of autism. He was my supervisor at the MIT-CSAIL laboratory and was concerned about me every day. He always checks that my project was going forward and would have to help me if not. He consistently allowed this thesis to be my own work but steered me in the right direction whenever he thought I needed it. I always felt that my research work was valued. One of his biggest fear was that I run out of work and he thus did everything he could to prevent this event to happen. Thanks to him my adaptation to my new environment was fast and easy.

I would also like to acknowledge Professor Martin Jaggi of the Machine Learning and Optimization Laboratory at EPFL as my thesis supervisor. I am gratefully indebted to him for his precious comments on this thesis and for the support all along with the project. I'm thankful for the time he dedicated to my research and for the trust, he put in me. This support ranged from the administrative tasks in the beginning to the final grading of my thesis.

I would also like to take this opportunity to thank the expert who was involved in this research project: Ph.D. Candidate, Shashank Srikant. Without his passionate participation and input, the quality of this project would not have been the same. To be precise, we worked a lot together as he initiated the project. He has created the used dataset and has also written some codes corresponding to the modeling part of the project. He also took care of my well being in the lab and I truly enjoyed our research discussions. I am also grateful, to all the members of the Alpha Group Lab researchers who agreed to share their knowledge and gave precious advice to allow me to have a fascinating and highly exciting time during the weeks I spent with them. This made my time in the US an incredible and life-changing experience.

Finally, I must express my very profound gratitude to my parents, Michèle & Serge and my brother, Arnaud, for providing me with unfailing support and continuous encouragement throughout my life, my years of study and through the process of research and writing this thesis. This accomplishment would not have been possible without them.

Thank you.
Merci.

Nicolas Lesimple

Exploring Deep Learning Models for Vulnerabilites Detection in Smart Contracts

Nicolas Lesimple
Computational Science & Engineering
EPFL
Email: nicolas.lesimple@epfl.ch

This project was initiated and done in collaboration with Shashank Srikant

Shashank Srikant
PHD Candidate at ALFA Group
CSAIL, MIT
Email: shash@mit.edu

Abstract

Solidity is a Domain Specific Language (DSL) that has emerged from the invention of distributed ledger and that has been designed for Ethereum. Due to its specific structure, DSLs are threatened by unique bugs and vulnerabilities that could induce millions of dollars losses. In this project, a Deep Learning (DL) model used with a novel source code input representation was created for the line-level vulnerability detection task. The input's structure in combination with a specific DL model can capture intricate data and control dependencies between various program variables. Using controlled settings and well-defined experiments, an exploration of the proposed approach was achieved to perfectly understand the model and to improve the performances. The developed method has successfully classified line-level vulnerabilities using a corpus of Solidity contracts as input. Our proposed pipeline is able to capture intricate dependencies between the program's elements and to understand the overall structure of the code. Characteristics defining the optimal type of input are described to define the cases in which the proposed pipeline should be used.

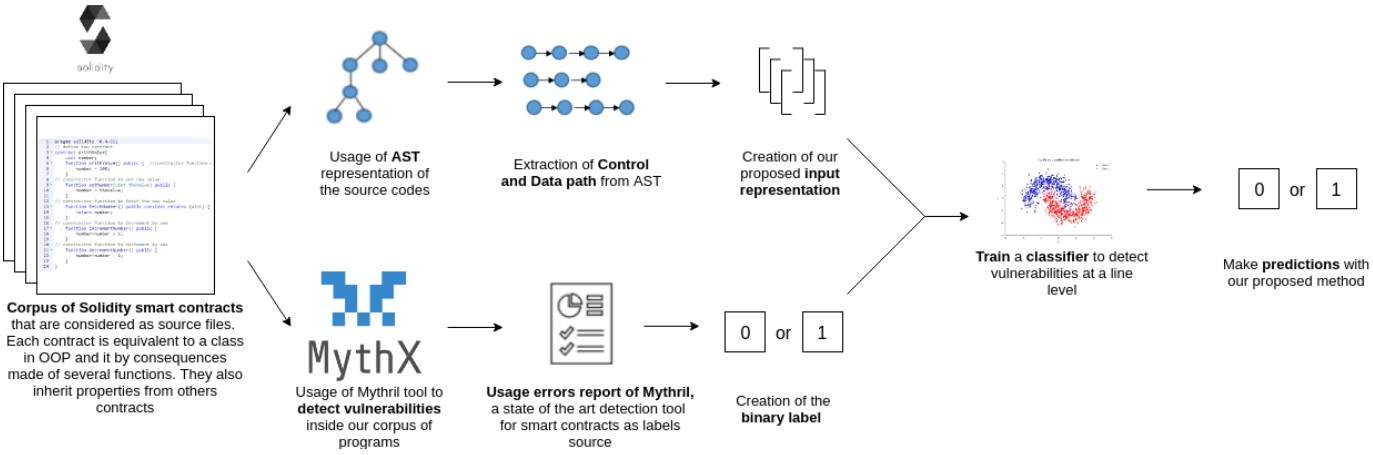
I. INTRODUCTION

A software vulnerability is defined as "the existence of a weakness, design, or implementation error that can lead to an unexpected, undesirable event compromising the security of the computer system, network, application, or protocol involved" [1]. As the number of software systems increases every day, the attack surface, which is the total sum of vulnerabilities that can be exploited, follows the same expansion. This surface needs to be protected to allow safe usage of the tools. Many recent vulnerabilities issues, including the DAO (Decentralized Autonomous Organization) bug [2] or the Heartbleed bug [3], underline the impacts of these security holes that can often have disastrous effects, both financially and on the society. These different repercussions illustrate why detecting vulnerabilities in programs has been an area of research [60] and still a great field of interest.

In this project, studied programs are written in Solidity, a **Domain Specific Language (DSL)** [4] designed for model Ethereum which is a popular decentralized distributed ledger. This kind of language is susceptible to vulnerabilities due to its very unique and specific structure : the vulnerabilities implicated in the DAO bug [2] are different from the ones inducing the Heartbleed bug [3]. These structural weaknesses are expendable to all DSL and are responsible for recent bugs that resulted in multi-million dollar losses. In the case of the DAO bug, the size of the attack surface scaled with the amount of lost money, showing the importance of reducing the number of vulnerabilities. In this specific case, the adversaries were able to manipulate smart contract execution to gain profit, by executing a complex sequence of precise actions. This process used to take advantage of the existing vulnerabilities showed how complex it can be to realize that a piece of code is vulnerable and thus underlines the interest of developing a software detection tool that could save millions of dollars.

Vulnerabilities are often due to a consequence of complex interactions between different parts of a program and largely occur because of a mismatch between what a developer expects the program to do and what it actually does. As it is impossible to guarantee the absence of vulnerabilities in a piece of code during its creation, then it is necessary to create methods to detect them. While there are existing tools ([9], [10], [11], [12], ...) for static (before running the program) or dynamic (during the runtime) analysis of programs, these tools typically only detect a limited subset of possible errors based on predefined rules. Despite the rich literature that addresses this topic (see Section *III.Related Work*), vulnerabilities and bugs escaping recognition persist. Our designed method, based on the recent widespread availability of open-source repositories, and on big-data-driven techniques aims to increase the amount of detected vulnerabilities. A corpus of programs is used as input, to discover the global patterns of bug manifestation, which is something that static checking cannot do. The key point to succeed in this task is to create an input structure able to quantify how variables interact in the context of program statements and execution flow. Our proposed approach uses paths build with **Abstract Syntax Tree (AST)** structure [47], which is mainly a compiler tree-based representation of the code. A specific DL model is then designed to take full advantage of the information given by the input structure : it uses a combination of control paths and data transformations information which means that it uses the context and interactions between elements forming the code.

Fig. 1: Overview of the proposed framework



The final goal of vulnerabilities detection is to fix them. This main objective explains that the state of art detection tools are working at a line level and also underlines the important need for understanding the causes of each vulnerability. Consequently, an interesting software vulnerability detection tool must give interpretability about its predictions. However, lack of model interpretability of deep neural networks is a limiting factor. DL models can indeed achieve high accuracy but at the expense of high abstraction. Finding features involved in modeling is not straightforward because of the inherent structure of the neural network : a neural network model corresponds to a structure that learns representations and patterns in the provided data by applying different transformations to the inputs and consequently gives birth to a task-specific output. In this work, the model is dissected in detail to understand how the latter reacts to different types of input, in fixed experimental settings. The main goal is to demystify the implemented network by understanding the causes of the different model's behaviors and by finding the features imply in the modeling. This gain of insights would also help to improve accuracy.

Fig. 2: Table of the Requirements needed to achieve accurate performances and to fully understand our approach for the vulnerability detection task

Requirements to detect vulnerabilities in smart contracts	***Solution***
Discovering **statistical patterns of bug manifestation**	Usage of a **corpus of programs** as input (big-data approach)
Quantify **how variables interact in the context of program** statements and execution flow.	Usage of **AST**
Add context information and **capture data dependencies within the code**	Usage of our **DL model with look up**
Assess the success of our method	Implementation of **baselines**
Understand the model to improve it	Usage of **defined settings** in differents **experimental processes**
Add **interpretability** to the model	**Attention weights** analysis

Contributions : The proposed framework is described in Figure 1. In this project, the following contributions are made to answer the requirements, described in Figure 2, needed to obtain accurate vulnerabilities predictions :

- A corpus of Solidity smart contracts was created and used as code input. The unbalanced attribute of this collection of programs was counteracted by a bench of manipulations described in *IV.A.Preprocessing* section.
- A new highly interpretable representation of a corpus of code was created to enable our model to understand complex dependencies between the variables within programs. This input structure uses AST representation to model the combination of control paths and data transformations. This structure is independent of any specific class of vulnerabilities and can be used for different tasks using source code as input.
- Several baselines using different kinds of input information were implemented to assess the success of our approach and to understand our models.
- A DL model was implemented to capture patterns that induce vulnerabilities and to take full advantage of the representation of the corpus of code. This designed architecture, described in section *IV.Method*, allows the understanding of the program's context and dependencies.

- A rigorous study was conducted on our designed models to be able to understand their behaviors in different experimental settings. Models with different architecture combined with inputs of different wealth in terms of information were tried and analyzed. A causal understanding of the accuracy of our models was consequently achieved.
- The usage of attention in the DL model, which is a kind of vector of importance, was also required to be able to add interpretability to prediction. This process enables us to understand the pattern of code responsible for vulnerabilities creation.

Therefore, this master thesis focuses on the understanding, the optimization and the assessment of the performance of a novel natural code processing approach, based on AST paths, that would enable the detection of software vulnerabilities. This research could lead to the creation of a program preventing vulnerabilities that could preclude disastrous financial and societal consequences.

II. BACKGROUND

A. Solidity and Ethereum

In this project, studied programs are written in Solidity, a DSL [4] designed for model Ethereum which is a popular public, decentralized, distributed ledger. A DSL is a computer language specialized in a particular application domain. This is in contrast to a general-purpose language that is broadly applicable across domains. A distributed ledger is a consensus of replicated, shared, and synchronized digital data geographically spread across multiple sites, countries, or institutions [66]. The main goal of this kind of ledger is to keep track of any transactions or contracts made in different parts of the world. The huge advantage of a ledger is the elimination of the need for a central authority which allows the entire system to be more robust against cyber-attack. In fact, to succeed an attack, the distributed copies stored in each node of the network need to be simultaneously targeted as safety correspondence algorithms between the nodes are used. The other main property of Ethereum is that all information stored in it becomes immutable and public. Public means that anyone can use the public function created on each contract pushed on Ethereum. Consequently, it allowed to maintained transparency into the trading exchanges. To be more concrete, Ethereum is the largest and most well-established, open-ended decentralized software platform. This particular ledger is powered by Ether that can be considered either as a crypto-currency either as a fuel to run command or application on it.

While Ehtereum enables the deployment of smart contracts, Solidity is used for implementing these programs involved in high-stakes transactions. A smart contract is a program that runs on the block-chain and has its correct execution enforced by the consensus protocol [67]. Each smart contract possesses its own attributes like for example its value and the address of the owner. By definition, these smart contracts are also immutable and can be shared by several actors thanks to the blockchain platform. A smart contract is equivalent to a class in an object-oriented language : it thus consists of multiple functions. From these contracts some vulnerabilities can arise due to their unique and specific design : in fact, functions inside programs are calling other functions and need to be well designed. They should take into account, for example, parameters like the wealth of the actors in a transaction or the time of the transaction, and it should be fast.

B. Creation of the dataset

To create the Solidity corpus of code, real-life codes were scraped from the website https://etherscan.io and dated from 2015 to 2018. Only files verified by Etherscan to be source codes corresponding to their byte codes available on the Ethereum blockchain were used. This set was made of 28052 unities. Then, codes that were not compilable were filtered out which gave us a collection of 25813 programs. Another filtering step was also used : only programs that had at least two transactions recorded on Ethereum were kept in the dataset. This served as a proxy for filtering contracts involved in genuine transactions. At the end of the entire process, our corpus of Solidity smart contracts was composed of 19023 files which represent 69599 contracts.

To conclude, thanks to scrapping, a corpus of Solidity smart contract was created and became our input. Our proposed approach kept the collection of smart contract dimensions which means that the model is taking several programs as input, and classify each line sequentially from the beginning to the end of the contract. During the modeling process, each line of code of every contract was treated as an input to the model. The precise format of the input is described in more details in Section *IV.B.Usage of Abstract Syntax Tree*.

C. Labeling Process

This step is a key part of the building of the Solidity dataset. Unlike tasks related to images or natural language processing with simple text as input, these codes and even more these vulnerabilities are hard to discriminate. This specific task is not achievable by all human beings. In fact, the labeling process is not straight forward as it required a qualified expert. That's why the usage of the error reports made by the state of the art vulnerability detection tools for Solidity was used as an input source. This method has the advantage to be scalable, automated and provides a robust way to analyze the strength of our approach. However, this method may inject some noise into the data. If the tool is wrong, our model is trying to learn patterns without the right labels.

Investigation was conducted on seven state of the art tools that have been developed to analyze Ethereum smart contracts : Mythril [9], Zeus [10], Oyente [11], Manticore [12], Solgraph [13], Solium [14] and Smart Check [15]. The most robust and popular of these tools (Mythril and Manticore for example) use symbolic analysis which is the state of the art method for preemptive detection for a wide class of bugs according to [58]. Several criteria were set to choose which tool could fit to fill our labeling process and only two, Mythril and Oyente, passed the following requirements :

- The tool should be based on sound technology and detect non-trivial issues
- The tool should be open-source, popular, and preferably well maintained
- The tool should be able to detect issues in smart contracts without having to modify/adding additional conditions, asserts, etc

These two tools appeared to detect different numbers of vulnerability types (with 3 in common). Thus, both tools were used on the corpus of Solidity codes and it appears that their predictions were similar in 90% of the cases. This indicates a possible presence of noise injected by our labeling process. Then, to be able to choose one of the two libraries, the specificities of each one were studied. As Mythril detects a higher number of vulnerabilities and as it gives more granular information compared to Oyente, the Mythril state of the art library was set as the label source.

Unfortunately, Mythril is not perfect and its limitations need to stay in our mind. As an example, it has been shown that until July 2018, Mythril did not detect overflows on values less than 2^{256} [16]. Some of the 11 vulnerability-types detected by the tool had shown some buggy implementation or too general warnings. To be precise, over the 11 types, only 3 of them can be used due to their occurrence in our dataset and due to the previously described issues. Besides, the creation of the corpus has shown several weaknesses and errors of Mythril tool that have been reported on Github to contribute to Mythril's support and community. It means that the tool can be, in some cases, kind of error-prone.

To conclude, the state of the art vulnerability detection tool for Solidity called Mythril was used on our scraped input codes. Mythril's error reports were set as labels source for the entire dataset. The tool work at a line level, meaning that each line of the input code has a label. The main advantages of this method are that the process is scalable and fully automated. Besides, it suppresses the need for a domain-expert. However, the usage of this external tool injected some noise inside the ground-truth data. In fact, Mythril is not reliable at 100% meaning that wrong labels are added in our ground-truth data which is definitely harmful to our implemented models. Indeed, according to private correspondence with Mythril authors, the dataset possesses an inherent false-positive rate of 10-15%. To answer this issue, a homemade synthetic dataset was used to analyze the behavior of our model on noiseless data.

D. Vulnerabilities in Solidity programs

Fig. 3: Distribution of the entire dataset in term of labels

Labels	Count	Proportion	Meaning
-2	939021	53,449%	Timeout
0	805922	45,873%	No vulnerability (represent 98,5% if filtering Timeout)
1	4350	0,248%	Integer Overflow
2	2331	0,133%	External Call To Fixed Address
3	4574	0,260%	Exception State
4	308	0,018%	Multiple Calls in a Single Transaction
5	103	0,006%	Unchecked Call Return Value
6	8	0,000%	Unprotected Selfdestruct
7	88	0,005%	Dependence on predictable environment variable
8	84	0,005%	Unprotected Ether Withdrawal
9	0	0,000%	Delegatecall Proxy
10	26	0,001%	Use of tx.origin
11	24	0,001%	Dependence on Predictable Variable

Figure 3 displays the number of apparitions of each vulnerability type in the entire dataset taking into account *Timeout* and *No Vulnerability* categories while Figure 4 displays the distribution of labels corresponding to lines of code having a vulnerability. From these tables, several conclusions can be drawn :

- The *Timeout* label is the most represented one. The corresponding data are unusable because Mythril was not able to classify them. In fact, a time limit of 5 minutes per label was set for the tool's analysis. This amount of time dedicated to the task could be set to any limit depending on business interest. This limit was chosen because it was an optimal setting considering the computational time/number of founded vulnerabilities trade-off. As it can be seen, 53,5% of the overall scraped data failed to produce a verdict within that time limit. This subset of data corresponds to unlabeled lines of code.

Fig. 4: Distribution of the dataset only corresponding to vulnerable lines in term of labels

Labels	Count	Proportion	Meaning
1	4350	36,57%	Integer Overflow
2	2331	19,59%	External Call To Fixed Address
3	4574	38,45%	Exception State
4	308	2,59%	Multiple Calls in a Single Transaction
5	103	0,87%	Unchecked Call Return Value
6	8	0,07%	Unprotected Selfdestruct
7	88	0,74%	Dependence on predictable environment variable
8	84	0,71%	Unprotected Ether Withdrawal
9	0	0,00%	Delegatecall Proxy
10	26	0,22%	Use of tx.origin
11	24	0,20%	Dependence on Predictable Variable

- Our dataset is unbalanced. The negative class, which means the class corresponding to *No Vulnerability* is much more present in the data. After filtering out *Timeout* labels, the proportion of *No Vulnerability* lines can be calculated : this negative subset represents 98.55% of the entire dataset. It means that the lines corresponding to any type of vulnerabilities, which is defined as the positive subset, represent only 1.45% of the data. To counteract the negative effect of the unbalanced property, preprocessing steps were conducted.
- Inside the positive set, 3 dominant vulnerabilities can be observed. The other types are nearly not existing and thus can't be predicted by a model. The 3 dominant labels were used as targets for our models where their union represents the positive set, while the other labels were not included in this subset.

The entire project's goal was to classify these 3 different types of vulnerabilities using a binary classifier. It means that the positive set of labels, formed by the union of the 3 types of vulnerability, is representing all lines having one issue. On the opposite side, the negative set represents the lines without any vulnerability. To be precise, 4350 *Integer Overflow*, 2331 *External Call to Fixed Address* and 4574 *Exception State* vulnerabilities, which correspond to 11265 malicious lines in total, were used in the dataset. The Solidity vulnerabilities of interest [17] are described below :

Integer Overflow [18]: Label 1 : Integer overflow vulnerabilities occur when a computed value is too large for the type attached to the value. Let's take the example of an integer : one integer has a maximum value which is $2^{256} - 1$. If the user tries to use an integer of a value near its maximum value to declare another variable of a different type that possesses a smaller maximum value, the overflow occurs. The operations that can cause overflow are "add", "subtract", "multiplication", and "exponentiation" instructions. The main consequence of this type of vulnerability is that simple addition or subtraction operation does not produce intended results. Thus, if this operation is used in a conditional statement, the program would have unexpected behavior. Several hackings using this kind of vulnerability have been observed and are described in [2].

External Call To Fixed Address (Unchecked Call Return Value) [19] : Label 2 : An external contract can take over the control flow due to an unchecked call of the return value. The consequences of such an issue are that an attacker could force the call to fail which would induce an unexpected behavior of subsequent program logic that could be used by an adversarial for his own profit. As this may cause different invocations of the function to interact in undesirable ways, the execution is resumed even if the called contract throws an exception.

Exception State (Assert Violation) [20] : Label 3 : In Solidity's design, assertion are use to assert invariants. A code flow should never reach a failing assert statement otherwise it means that it's not working properly. In the case of this issue, a failing assert invariant statement is reached, meaning that a bug exists in the contract or that assert is used incorrectly.

III. RELATED WORK

Research about bug detection and consequently about program repair has been a major research area for the last decade [60] and is currently becoming more and more important with the emergence of DSL in several domains like cryptocurrency. Repair program corresponds to the fully automated process of fixing bugs into code while bug detection is only the first step of this entire task. Two main challenges need to be overcome to achieve accurate predictions for vulnerability detection task: the representation of source code used as input needs to capture the intrinsic semantics of a program and the relations of variables within a program (long term dependencies between variables ...) while the model needs to be designed to take full advantage of the proposed input structure. Research about source code representation, about the optimal model to capture variables interactions and about the combination of both have been achieved to create methods reaching good performances.

A first reflection has occurred about the best input for vulnerability detection tasks. Some researchers have tried to use Github information. In [32], vulnerability detection relied on the information of commit messages and bugs reports. Other methods, that do not need any source code data, were also proposed as in [52] where researchers focused on a dynamic taint analysis by performing binary rewriting at run time. Another way of seeing the problem was to directly consider the transaction made by smart contracts instead of the code : to take advantage of vulnerabilities, hackers mainly focus on functions creating a mismatch between the actual transferred amount and the amount reflected on the contract's internal stored data. In [49], this vision was used to detect irregular transactions due to various types of adversarial exploits. However, it has quickly been shown, with the rise of DL, that the best input for this kind of task is a corpus of several codes containing some vulnerabilities. The bench of described papers in the following paragraphs illustrates this fact. As said previously in Section *II.C.Labeling Process*, obtaining labels corresponding to vulnerabilities in source code is a difficult step as it requires an expert. One solution is to automatically create bugs from well-functioning code. In [25], a name-based bug detection method was developed and showed that learning from artificially seeded bugs yields bug detectors that are effective at finding bugs in real-world code. Another solution is to use existing detection tools as done in [27]. Our work uses this approach, where a corpus of source code was used as input and state of the art statics analyzers were used as labeling source.

Another reflection has been made on the representation used to describe source code in combination with the model architecture. The older vision was based on static analysis and fuzzing to detect vulnerabilities. In fact, traditional program analysis techniques were used in [50] where string-based vulnerability signatures were generated with an automated based string analysis framework. Another paper [54] tried to reduce the task to a **Natural-Language Processing (NLP)** problem by using a statistical language model. Then, different NLP techniques were developed and used to understand source code in various tasks. Experts considered code as text and used NLP knowledge to infer vulnerabilities presence. In [22], the inference of generative models was used to expose bugs in compilers. In [24], neural machine translation techniques were used. The extraction of bug fixes pattern was proposed in an Encoder-Decoder model to infer the presence of buggy code and fix it. The same approach was also used in [30] focusing on JavaScript code.
More recently, with the widespread of DL and open-source code, a data-driven approach based on neural networks with a corpus of codes as input, was proposed and studied for the vulnerabilities detection task. In [27], a method relying on deep features representation learning to be able to interpret source code directly was developed. Static analyzers were used to obtain labels and a CNN bag of lexicalized tokens in combination with a random forest classifier formed the proposed method. In [29] DL is used to take advantage of the specific structure of source code. Labels were also created with a static analyzer. In this work, source-based models showed better performances compared to models applied to artifacts extracted from the building process. The same method was used in [46] where it has been shown that this approach was able to infer bug presence in JavaScript source code. In [37], it has been proven that a DL model using Long Short Term Memory, partly inspired by human memory, was able to learn long term semantic links appearing inside source code. In fact, this work has shown that language models such as n-grams for software code often fail to capture long context dependencies compared to DL models. Being more specific, the paper [33] focusing on buffer over-runs vulnerability detection reached the same conclusions. Considering all these studies, our method was designed to take advantages of this data-driven approach combining a corpus of codes as input with DL.

The majority of previously described methods, if it's not all of them, can't capture the syntax and the semantics of the input source code due to their design while our work focuses on the creation of highly interpretable features which capture variable interactions. This ability is key to build accurate and interpretable models. To succeed in this task, the AST representation was used. The pioneer paper [47] about the combination of DL with the usage of this kind of representation confirmed the feasibility of this approach while [44] underlining the advantages of such a representation using 'bag-of-path' structure as training input. In fact, even without the usage of DL, [48] illustrated the ability of AST path representation using only statistical models. In [56], the two terminal nodes of AST are used which allows the understanding of the control and the data-context of those nodes. However, the proposed model was not able to capture long-range dependencies. To overcome this weakness, in [34], an architecture composed of a **Recurrent Neural Network (RNN)** used in combination with the AST representation of source code was proposed. The output vector representation was then used for different program comprehension tasks and proved its value.

In [26], this algorithm able to learn the intrinsic structure of source codes was used to detect bugs with a **Long Short-Term Memory (LSTM)** instead of an RNN. It showed its higher ability to learn features to infer the presence of vulnerabilities. The retained design for our work is by consequence a DL model, made of LSTM networks using AST path as input representation. Our proposed design, working at a granularity of lines in a program is able to model information flow from one line to another.

Another work [35] investigated even further. This work proposed a representation based on a collection of AST paths aggregated smartly into a code vector used to predict the semantic properties of the snippet. The paper [42] also underlined the advantages of using a recursive aggregation of AST paths as input in a DL model. Besides, [43] has shown that this kind of model, using both AST paths and DL, could be used in combination with a hierarchical attention mechanism to add interpretability to the predictions and increase performances. Consequently, these two concepts were retained to be used in our designed approach : a smart aggregation of AST path is used to add more information in the input while the attention layer with analysis of the corresponding weights is used to increase the performance and the interpretability of our method. Other approaches were also invented with the usage of AST paths as in [40], where program dependence graph and data flow graph were used in combination with the local information extracted from the path on the AST to model the context and the semantic inside source code. Another method [38] used a tree-based convolutional neural network, in which a convolution kernel is designed over programs'AST.

The reflection about the optimal representation of the source code also resulted in different input structures that aim to enhance the capability of the model to understand the code's intrinsic structure as in [55] where linear models were run on program graphs-based features. Several other works were based on a graph-approach : in [39], [45] or [36], graph representation was used to better capture long-range dependencies and graph-based DL algorithms were proposed to analyze the program comprehension on different tasks. Researchers also tried to use an attention neural network mixed with a convolution neural network in [53]. This method was able to detect local time-invariant and long-range topical attention features according to the paper which again underlines the importance of attention that is used in our design.

In this quest of meaningful source code representation, even the execution information and the binary version of a program were studied. In fact, in [28], AST paths are not considered as the optimal way to represent source code. The preferred representation relied on program execution traces, used into an RNN. This method was also used to automatically repair code in [31]. Also, some researchers in [23] tried to use the compositional analysis method based on symbolic execution. Interpretability was added to the results thanks to the usage of various heuristics. Another work [41] used a deep neural network that learns from program execution traces. Some researchers even tried to directly work on the binary representation of the code. In [21], the binary representation was directly used and a method that relied on a maximal divergence sequential Auto-Encoder, which tried to maximize the divergence between vulnerable and non-vulnerable latent code was developed. Another work [51] tried to extract security-related properties from binary programs to build a unified binary analysis platform. Even if these works showed good results, their approaches were not retained for our design.

To conclude, according to the studied related work, our designed method was created and corresponds, from our point of view to the most promising approach.

IV. METHOD

Figure 5 will guide you throughout this entire method section by summarizing the choices that were made and why they were made.

Fig. 5: Table summarizing the choices made during the design of the proposed method with an overview of the causes and answers given to the faced issues

Issues faced	*Solution*
Noisy Input Data	Creation of **synthetic** data
Unbalanced Dataset	**Subsample** the **negative** data. Use weighted loss in the model. Focus on the more represented positive classes.
Code as Input	Use Abstract Syntax Tree (**AST**)
Add **context information** corresponding to each line and **link variable to their previous usage**	Addition of Endpoints / Previous line embeddings
Focus on the more important part of the information contained in each line	Use **Attention** Layer
Translate the **sequential properties** of text	**Bi-LSTM Network**
Long computation Time	Allows **batches** computation
Treat Data with **different shape**	Create batches **with programs of same length**

A. *Preprocessing*

Preprocessing's goal is mainly to counteract the negative effects of the intrinsic properties of our raw input. In fact, as *Timeout* corresponds to unlabeled data, the first step of the preprocessing pipeline is to filter out all the *Timeout* labeled lines. The creation of the labeled dataset is then achieved and data can be used in a supervised machine learning algorithm. This transformation induced the suppression of 9253 files which corresponds to 49% of the total collection.

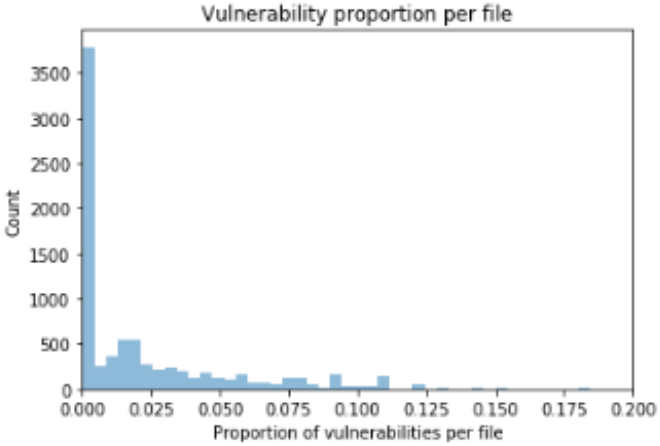
Fig. 6: Preprocessing pipeline :
Table indicating the proportion of vulnerabilities and the number of smart contracts in the dataset after each preprocessing's steps

Step in the Preprocessing	Percentage of positive labels being 1, 2, or 3	Number of different program
Count of Labels (lines) Raw Data	0.64%	19015
Count of Labels (lines) After Filtering of Timeout	1.37%	9762
Count of Labels (lines) After Filtering smaller and bigger program (<10 and >160 lines)	1.78%	8172
Subsampling of 0.5	3.55%	4086

The second step of the pipeline consists of suppressing smart contracts considered as outliers. Outliers mean programs with a too small or too large number of lines. To be precise, smart contracts having less than 10 lines or more than 160 lines of code are thrown away. This process suppresses 1590 programs which correspond to 8% of the entire corpus.

The last preprocessing stage is the one designed to decrease the effect of the unbalanced property of our dataset. The main idea is to suppress data of the more represented class to increase the proportion of the under-represented class. To do that, the entire collection of the remaining smart contracts was ranked according to their proportion of positive labels, which means according to the number of vulnerabilities contained inside them. Programs having the lowest rate of vulnerability presence were suppressed first. This process allows an optimization of the number of positive labels in our dataset. In our case, 4386 files have at least one vulnerability over 8172 as can be seen in Figure 7. It means that 3786 files do not possess any vulnerabilities. That's why, a subsampling rate of 0.5 has been chosen. It induces that half of the smart contracts were suppressed according to their vulnerability presence rate. Thanks to this step, the number of programs in the dataset decreased from 8172 to 4086 elements. This diminution represent 20% of the initial entire set of contracts.

Fig. 7: Distribution of the files forming the corpus of programs according to their proportion in term of vulnerabilities presence



The resulting set of smart contracts obtained after the preprocessing pipeline (summarized in Figure 6) using a subsampling rate of 0.5, was used as our reference input source in our study. This source is made of 3.55% of positive labels which is an acceptable amount for designing a performing classifier. Consequently, our default input source is made of 235297 lines corresponding to *No Vulnerability*, 3876 corresponding to *Integer Overflow* issue, 1475 corresponding to *External Call To Fixed Address* vulnerability and finally 3329 corresponding to *Exception State* error. Usually, even 2% of positive labels could be enough to achieve high accuracy if the architecture of the model is taking into account this unbalanced property of the data. Other sources of input created by changing the subsampling rate were also studied and results can be found in the *Appendix* Section.

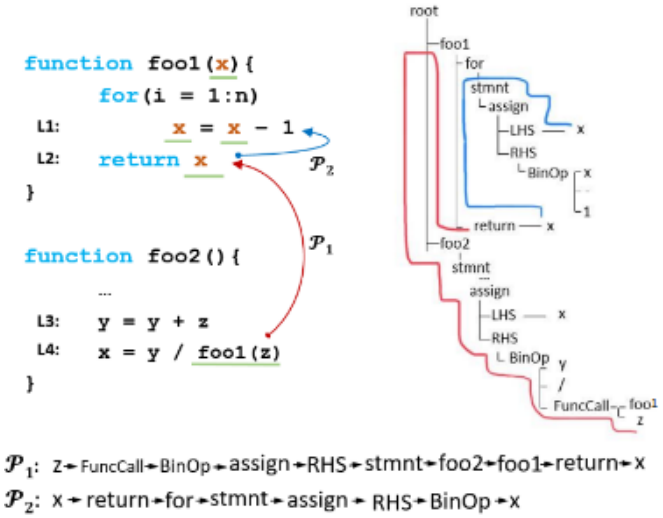
B. Usage of Abstract Syntax Tree (AST)

The main goal of this section is to describe the representation used for modeling code as input, which corresponds to an optimal structure. Optimal because the proposed architecture is able to model the dependencies of tokens inside the entire contract where tokens are defined as the elements that composed the lines of code of our input corpus. To be precise the three major components appearing in a code line are considered as tokens : variables, operators or function calls. To reach this goal, the usage of AST is required. An AST is mainly a way to represent the syntax implicated in a program using a hierarchical tree-like structure. It contained partial semantic information on the different interactions occurring between the elements in a program. This structure can be used by compilers to construct the symbol table needed for the compilation. Indeed, it aids compiler optimizations to determine the liveness of variables, etc [57]. This visualization is more focused on rules and dependencies between tokens rather than element-wise. The usage of AST paths allows our model to access information about the trace of the previous usage of a token and can help to associate all the operations it had been a part of. In this way, the representation gives enough information to our model to make him able to discriminate line-level information and thus succeed in the vulnerability classification task with high accuracy.

Thanks to AST representation, **Control and Data Path (CDP)** can be associated with each tokens forming a line. A path corresponding to one token is the part of the AST origination from the corresponding node and terminating at the previous usage of the considered token. All the nodes of an AST are strings and the ones implied in the path of interest are put into a list to create the path representation. The last node of the list is the previous variable or last function call's value used. An example of paths with the corresponding AST is displayed in Figure 8 (*P1* and *P2* at the bottom of the figure). This illustration should allow the reader to understand that a CDP is mainly built by going backward in the AST representation thanks to the 2 color lines of the figure. These CDPs can not be created for operators. In our model, operators are just ignored and are not considered as tokens. However, the term path will be used with all tokens for easier description.

Let's take an example to show how an AST path is built and that these paths could represent both the data and control dependencies. The example used is described by the code of Figure 8. Specifically, the line labeled as *L4* is studied. This line is belonging to one specific program and is formed by 6 tokens : *x, =, y, /, foo1, and z*. On the right side of the Figure 8, the AST graph of function *foo1()* and *foo2()* is displayed. In addition, the AST path corresponding to variable z of *L4* is P_1 and the one corresponding to x of *L2* is P_2. Now let's focus on the token z, where z is a variable in a programming language. In function *foo2()*, variable z is used in *L4* with a binary operator and with *foo1()* function. Thus, due to the *foo1()* function, variable z is used in a for loop, undergoes a decremental operation at *L1* and is returned by the function at *L2*.

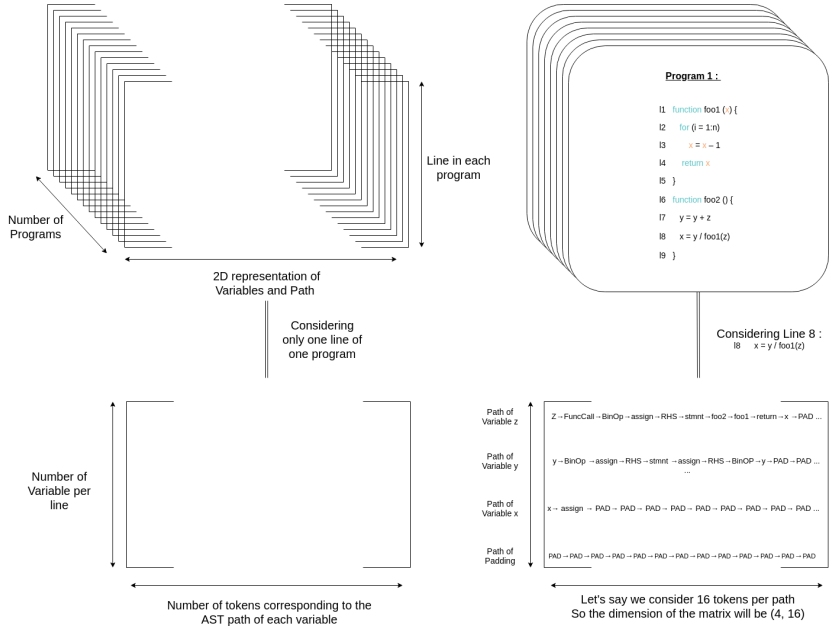
Fig. 8: Example of a snippet of code with the corresponding AST representation and with the corresponding CDPs for two variables



Consequently, this token is affected in 3 lines : *L1*, *L2* and *L4*. To be able to describe fully variable z of *L4* in the right way, our representation should reflect the previously observed behavior with the usage of z in the for loop of *foo1()*. In this case, by looking at P_1 and P_2, it can be observed that the extension of the path originating from *L4* to *L1* via *L2* can be done. The association of the paths of *L4* with *L2*, which means the association of *P1* and *P2* would be enough to describe the trace of variable z and successfully establish the full context of the variable. As you can see, the two paths described all the observations needed for variable z : binary operator, called of *foo1()* function, for loop, return statement. Thanks to this process, on one hand, explicit data dependencies were modeled thanks to the paths but also, on the other hand, the tokens were modeled as they come up in the paths. This concept is applied in our model to recursively represent the entire set of connections between tokens at different parts of the code. No other preprocessing method like program slicing is needed to obtain this information.

C. Input Representation

Fig. 9: Token Level Embedding :
Description of the 4D representation used to represent code as input



Thanks to Figure 9, the description of the chosen representation is illustrated and the corresponding terms and notations are described in the following list to allow an easier comprehension of our work. The architecture built to model the inherent properties of code is made of 4 dimensions. The first one represents the number of programs in the corpus of Solidity smart contracts input. Each contract of this corpus is made of several functions represented by several lines. The number of lines per program is an immutable attribute of each contract and is defining the second dimension of the structure. In fact, in our model, a line-level distributed representation also called embedding is learned. Then, for each line of each program, a 2-dimensional representation is created. Thus, each line is represented as a 2D matrix. The first dimension of this line-level structure is the number *k* of tokens in the line. Then, as illustrated in Figure 8, each token is associated with a control and data path thanks to the AST representation of length *l*.

During the modeling part, the length of each program is not constant as it is a code's immutable property. However, the maximum number of tokens per line *k* and the maximum path length *l* (number of nodes composing each path) need to be constant and set before usage of our model. The default settings are 4 tokens per line and a maximum path length of 16 nodes.

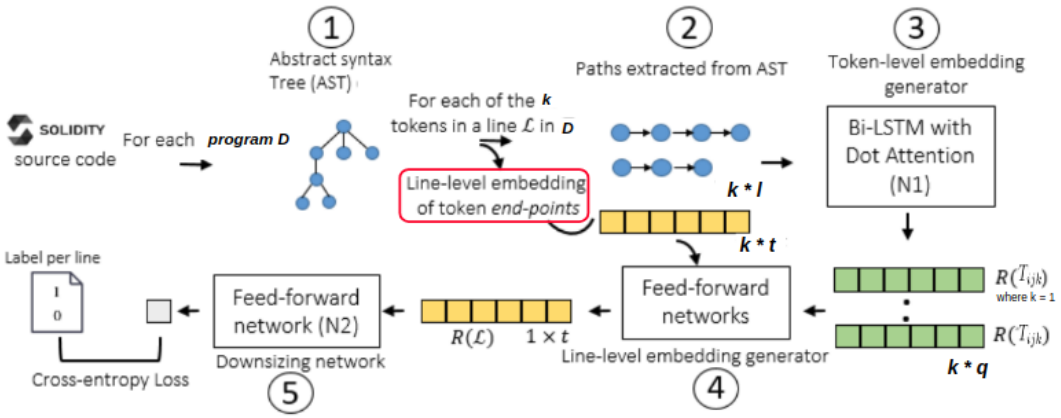
The list below summarizes the different dimension of the proposed input with their meaning and notation :

- D : Entire set of programs forming the corpus of Solidity smart contracts used as input.
- D_i where $0 < i <$ *Total number of programs in the input* : Represents one smart contract in the set D : For the specific index *i*, each program displays different length.
- L_{ij} where $0 < j <$ *Total number of lines in the contract* D_i : Each program D_i consists of an ordered set of *j* lines of code. Thus L_{ij} represent a particular line in a particular program.
- $R(L_{ij})$: Line level distributed representation which is also called line-level embedding.
- T_{ijk} where $0 < k <$ *Total number of tokens in line* L_{ij} *in the contract* D_i : Each program D_i consists of an ordered set of *j* lines of code L_{ij} in turn consists of *k* tokens. Thus, T_{ijk} represents a particular token, in a specific line, in a specific contract.
- P_{ijkl} where $0 < l <$ *Total number of node in the control and data path corresponding to the token* T_{ijk} *in line* L_{ij} *in the contract* D_i : Each program D_i consists of an ordered set of *j* lines of code L_{ij} in turn consists of a set of *k* tokens T_{ijk} and finally in turn corresponds to a control and data path of length *l*. Thus, P_{ijkl} represent a particular node in the control and data path linked to a specific token, in a specific line, in a specific contract.
- Negative set : the subset of data made by the collection of the lines corresponding to *No Vulnerabilities* labels.
- Positive set : the subset of data made by the collection of the lines corresponding to vulnerabilities labels.

To improve the model performance, the transfer learning method, using embedding already trained for a similar task has been considered. However, as our proposed structure is totally new and has never been used before, this transfer learning concept can not be used.

D. Overview of the Model Architecture

Fig. 10: Overview of Models' Pipeline



In this section, the global architecture of our model is described with the help of Figure 10. DL models taking a corpus of Solidity contracts as input were implemented to succeed in the binary classification of vulnerability detection task at a line level. It means that it outputs one label per line L_{ij}. As previously described, the AST path representation is used to create one path P_{ijkl} for each token T_{ijk} belonging to each line L_{ij} for each contract D_i. This action is represented by steps 1 and 2 of Figure 10. The reason for using AST paths is to capture the interaction between tokens inside the same line

and additionally, between tokens present in the previous lines to simulate the real execution flow of a program. However, our approach needs to be designed to take full advantage of this input information. To do that, the key point that makes our model special is the way information from the past line is used to classify the current line. During the building of the described representation of code, several pieces of information are stored per line. To be precise, what is called *end-points* indexes pointing on past lines, are stored. These indexes are used to link each token to their previous usage which aims to add context to each line. In the example of Figure 8, for tokens of the line *L4*, the line *L2* would be stored for *z*, the line *L3* would be stored for *y* and the line *L4* would be stored for *x*. This information is then used during the training phase of our model to combine information and model CDPs' dependencies. This step is what makes our model more capable of learning context and links between tokens inside a code. This process will be described in more details in the following section.

Then, at step 3 of Figure 10, the representation depicted in Figure 9 created after step 2 is used as input in an LSTM network combined with a Dot Attention Layer. The sequential properties of the text were taken into account thanks to the usage of the LSTM network as proven in [26]. In our designed model, the attention layer, invented in [5], plays a key role during the learning process and is also the building block of the interpretability of our algorithm. The goal of this attention layer is to mimic the human attention mechanism. To have more details about this concept, refer to Section *IV.E.Attention Mechanism*.

The specific network using this attention concept, defined as *N1* used in Step 3 of Figure 10, allows the creation of a token-level embedding vector, which is combined with *end-points* information to feed a simple feed-forward network. The token-level embeddings created in Step 3 are distributed representation of dimension *q* and possess the information corresponding to CDPs. The representations of each token $R(T_{ijk})$ forming a defined line L_{ij} are concatenated together as researchers did in [35]. This concatenation is then used to generate a line-level representation $R(L_{ij})$ using the feed-forward network defined in Step 4. This last line-level embedding is consequently used in another feed-forward network to downsize the vectors to finally obtain one binary label per line. *N2* network learns the relationship between the line and the attributed labels. A statistical analysis made on attention weights induces a causal comprehension of the patterns implied in vulnerabilities.

Thus our models mainly need two sources of information : the token-level AST paths and the stored *end-points* indexes. Thanks to them, two sets of embedding are created in the models : the token-level embeddings and the line-level embeddings. The creation of the last needs the *end-points* line-level embeddings which are simply the already created line-level embeddings corresponding to past lines of the previous usage of tokens forming the predicted line. In terms of comparison with NLP methods, our algorithm can be compared to the generation of embedding for each sentence and eventually each paragraph as NLP input is made of multiple tokens per line while a paragraph is a collection of sentences. In this case, paragraph-level embedding is used to infer the line-level vulnerabilities. However, no comparison with NLP techniques has been found for the combination of line embedding and *end-points* classifiers.

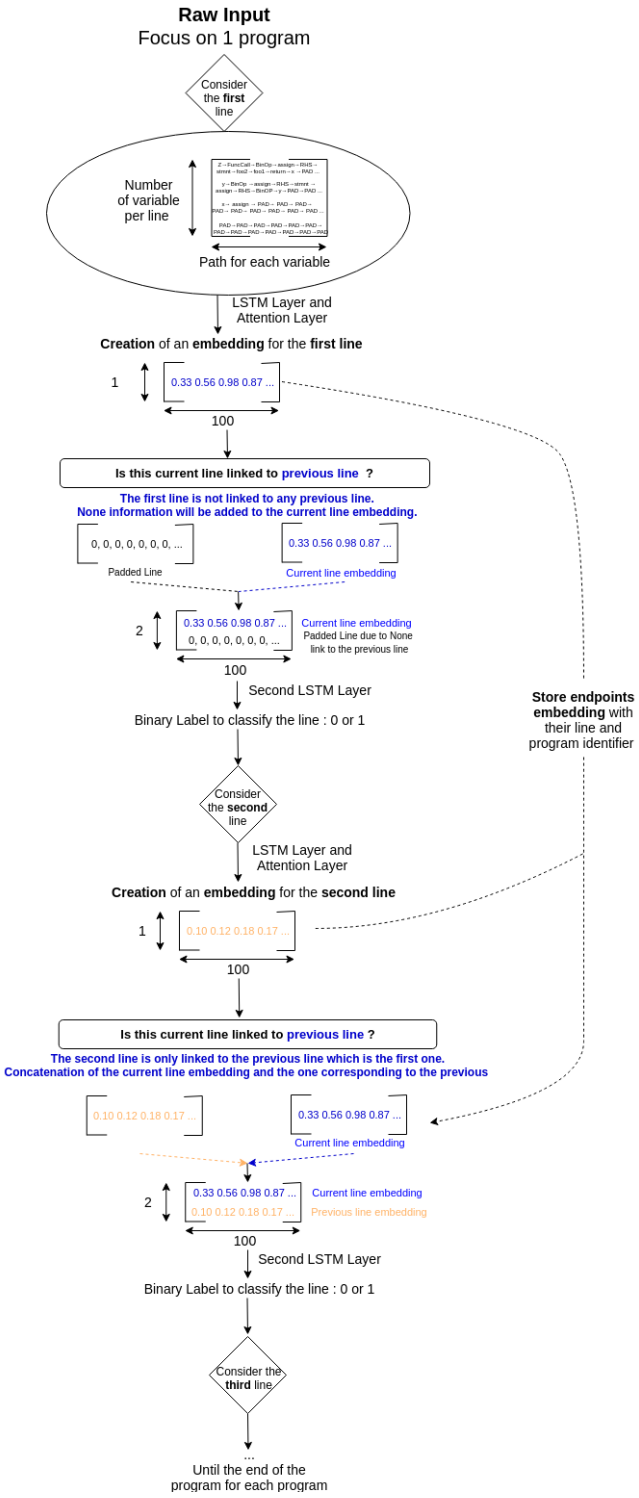
E. Attention Mechanism

The usage of the attention layer is motivated by the human attention mechanism. This human process is defined by the ability to focus on specific parts over a complex set of information, like for example focusing on important words in a sentence or on specific parts on an image. For example, in the sentence *she is eating an orange carrot*, the high attention words are *eating*, *orange* and *carrot* while the article *a* is nearly invisible for the human brain. This attention mechanism is instinctive for the majority of human but this is not the case for machine learning models. Thus, research has been done to model this useful process. In [62], the mathematical modeling of attention applied to NLP was created to help memorize long source sentences in neural machine translation. Then, different types of attention applied to machine learning corresponding to different specific tasks have been discovered. In our work, attention can be summarized as a vector of importance weights describing the power of each token in terms of discrimination of vulnerabilities. To be specific, attention creates shortcuts context vector. It associates a weight to each element forming the dictionary of tokens present in the input. These weights are the illustration of the importance of the associated tokens during the prediction process. Thanks to this vector, tokens implied in each type of issue can be identified, which allows the interpretation of the causes of the predictions. In our work, the attention mechanism invented in [7], was implemented in the same mathematical way as in [35]. In our proposed method, an attention layer composed of one dense layer was inserted between the two LSTM layers. The details of the implementation can be found in the Github link of the paper [61]. This mechanism is similar to the one published in [6].

F. Previous Line Model

The **Previous Line (PL)** model, described in this section, is built following the architecture presented in the previous Section *IV.D.Overview of the model architecture*. To illustrate the different concepts used and the usage of the *end-points* line-level embedding, the example displayed in Figure 11 is used in combination with the previously described example of Figure 8. To be precise, Figure 11 illustrates in details steps 3, 4 and 5 of Figure 10. The analysis is consequently focused on these steps.

Fig. 11: PL Pipeline :
Diagram representing the main steps of the pipeline of the Previous Line Model
with a defined maximum number of token per line of 4



Let's consider only one smart contract as input. In this case, the PL model is going through each line of this program sequentially, from the first line of the code to the last one. For each line, a collection of different CDPs with a defined length corresponding to the tokens forming the considered line is used as input. This input matrix was created as described in Section *IV.C.Input Representation* by using the AST representation. The collection corresponds to the 2D matrix of Figure 9 described earlier. This representation is used in a bi-directional LSTM network with local, multiplicative attention. Thus, this first network learns the different paths that can be associated with tokens in the scope of a program. The output of this first step is forming a line-level embedding of dimension (1, 100). This created embedding is stored in a look-up structure to be used if needed.

At that point, the model asks himself if the current computed line has a link with the previous one. A link exist to the previous line if at least one of the tokens used in the current line was used in the previous line. For example, in Figure 8, the variable *x* of *L2* is used in *L1*. In this case, *L2* is linked to L1. To know that, information was already processed during the path creation and stored in an array corresponding to the *end-points* data. By definition, the first line can not be linked to the previous line. Thus, as depicted in Figure 11, the created embedding is concatenated with a padding embedding to create a final embedding that is fed into an LSTM layer followed by two simple feed-forward networks. In this specific case, none information is added and our PL model is just a usual DL model. This network produces a logit which is then compared to the binary label using a cross-entropy loss function.

Then, the model uses the same pipeline for the second line. However, in this case, the second line is linked to the first line. For example, on Figure 8, token *x* of *L2* is linked to *L1*. In this case, thanks to *end-points* information, a lookup is done to find the line-level embedding of dimension (1,100) created previously for the first line and a concatenation of the current and previous line representation is done. Consequently, a large amount of information is added at this step. This process allows the PL model to understand short term dependency between tokens.

To summarize, the usage of the *end-points* indexes during the training phase is used to look-up and concatenate line-level embedding of the current line and of the previous line to capture more context and dependencies.

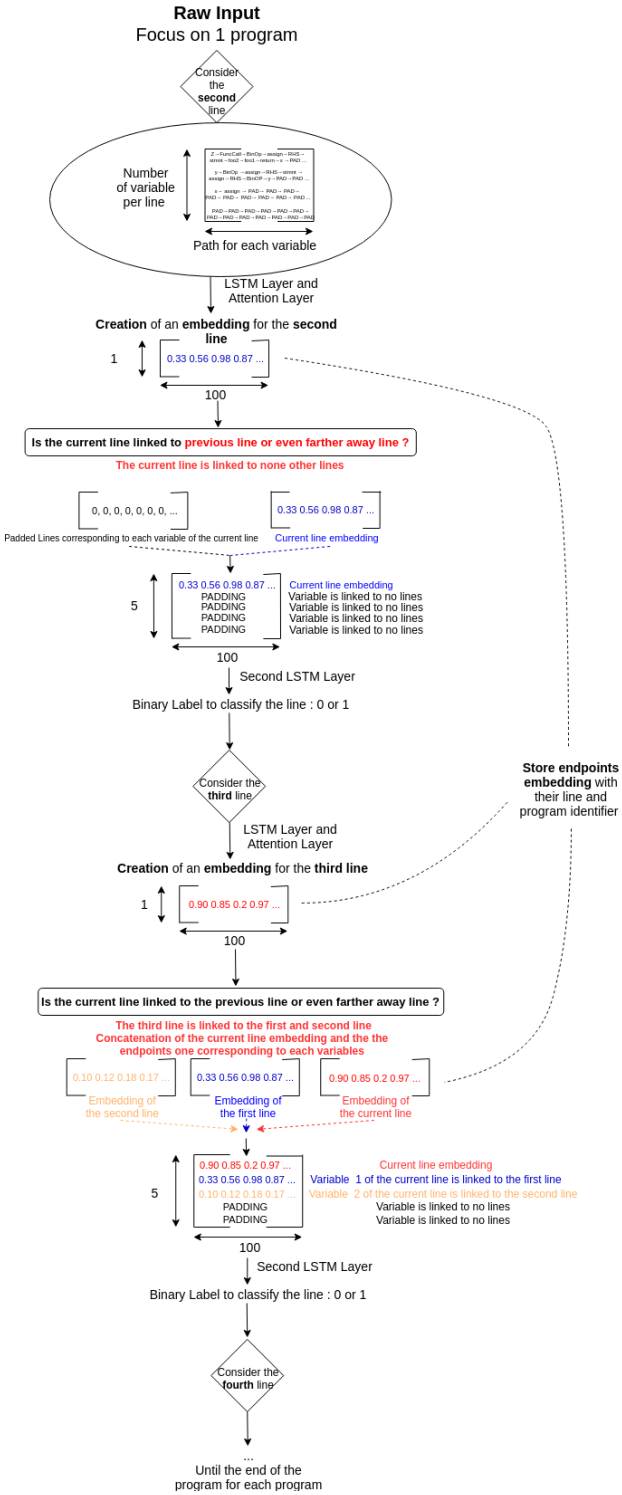
G. Endpoints Model

The only difference between the PL model and the **Endpoints (EP)** model is that the *end-points* information is not only linking tokens to the previous line but also to all the previous lines having dependencies in the entire code. Moreover, remember that using the default settings, one line is made of 4 tokens maximum that are all linked to a CDP. For the EP model, one past line embedding per token is chosen to be concatenated into the final line-level embedding, while for the PL model, only the previous line embedding was chosen if at least one of the tokens was linked to it. Thus the EP model adds maximum 4 paths while the PL model adds maximum one path to the current line's path. The Figure 12 illustrates these differences. Let's take a concrete example : *L4* of Figure 8 is made of 6 tokens : *x, =, y, /, foo1* and *z*. In this case, the *end-points* line of the tokens are respectively lines *L4*, *L3*, *L2*, and *L3*. Remember that = and / tokens can't have a *end-points* index as operators are not able to have a real CDP because they can't have any dependencies to the previous lines. If the PL model was used, *L2* would not have been chosen while in the case of EP model, *L2* and *L3* would have been chosen. Consequently, the amount of injected information is greater using the EP model than the PL model and longer dependencies are captured.

As noted in the example of Figure 8 for token *z* of *L4*, one CDP can not be sufficient in all the cases, even with our input structure, to fully capture the context of the tokens interaction in the remaining lines. It happens specifically to tokens appearing in two different lines of code. Again, let's take the example depicted on Figure 8 : the CDP of the token *z* corresponding to *P1* is not sufficient to understand that the returned *z* goes through a decrement operation on *L2*. It also needs *P2* to capture all the relations.

To summarize, adding *end-points* line-level embedding allows to capture the context and the dependencies of tokens used multiple times inside the entire program. However, the complexity also increases as the number of needed lookup follow the same variation. The choice between EP and PL models should be done wisely. The method developed in [65] could be used to evaluate the created embeddings.

Fig. 12: EP Pipeline :
Diagram representing the main pipeline of the Endpoints Model with a defined maximum number of token per line of 4



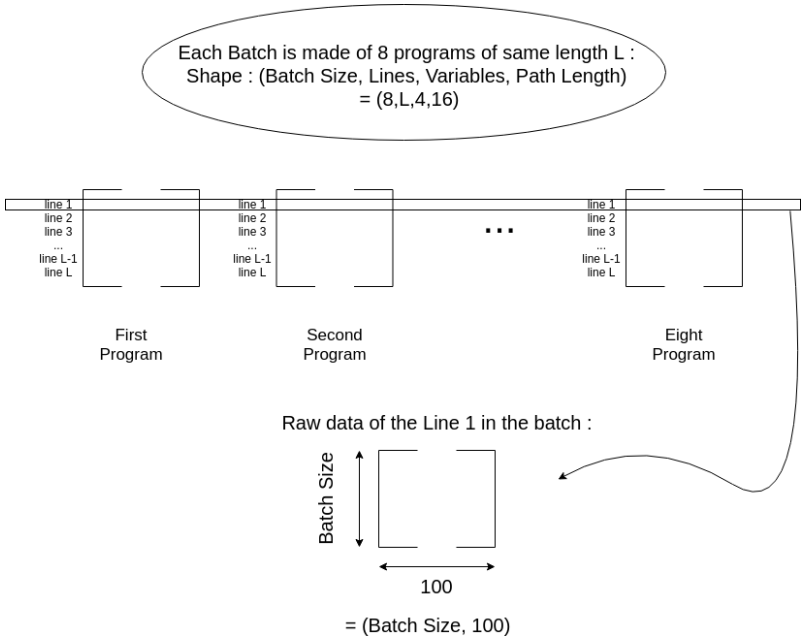
H. Batches computation

The examples studied previously are considering one smart contract at a time. The PL model and the EP models were presented as iterator on each program's lines. Indeed this is true, but to decrease the huge computational time needed for one run, an implementation adapted to the usage of batches was made.

This implementation was hard to develop due to one particularity of our representation. In fact, while the number of tokens per line and the number of nodes per path is set, the number of lines per program is not. Our input, made of real-world Solidity contracts, is by definition made of programs with different sizes.

How to create batches with different input shapes? The only possible solution is to group into the same batch programs of the same length. By doing that, each batch is ensured to have a 3D structure with a defined immutable dimension. One of the goals of implementing this batch size computation is to decrease the computation time. To succeed in this task, the filtering of the batches without a required number of programs is applied. In fact, the default batch size used is 8. If in the corpus of smart contracts, only one program has a length of 50 lines of code, one batch will be created with only one contract. These cases need to be ignored and are thus filtered out. This filtering suppresses 19% of the studied smart contracts.

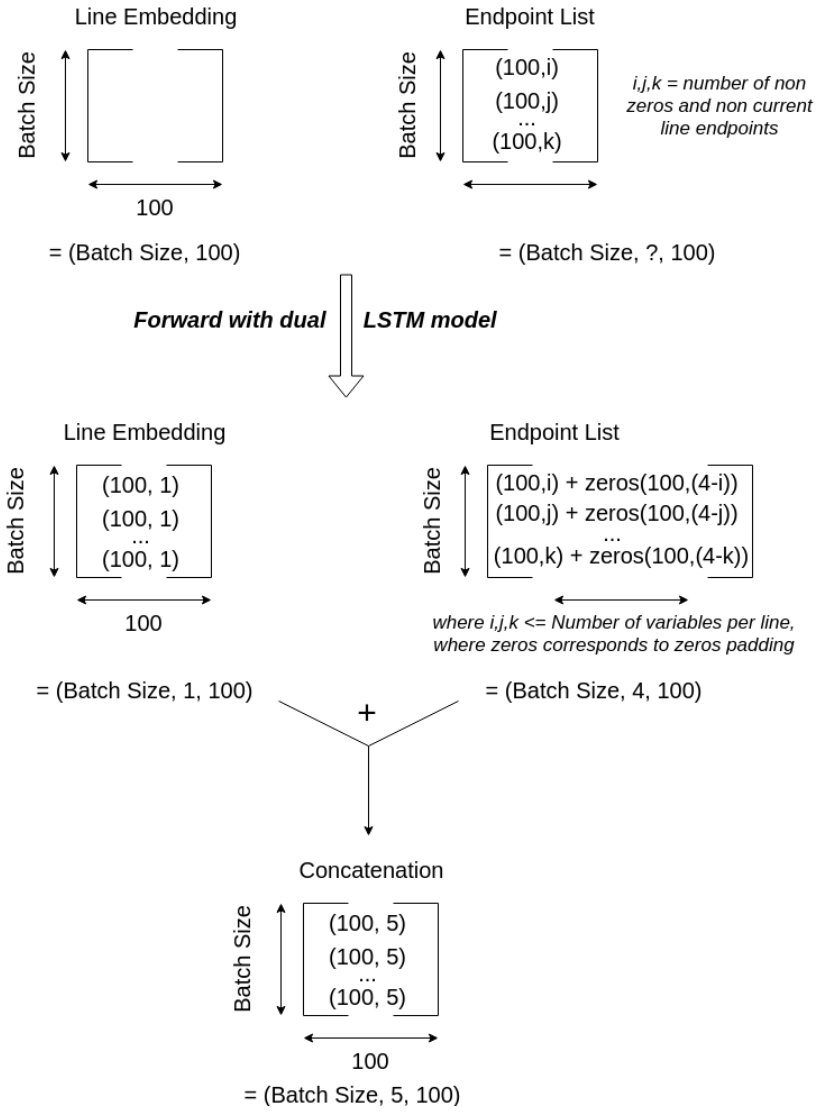
Fig. 13: Building of the Batches :
Diagram illustrating the building of batches using as a source of data the line-level embeddings previously described of dimension (1,100) created with the LSTM network



How was conducted the computation inside the model with that kind of batches ? As our models are working sequentially through the lines and as the *end-points* look-up was needed, the only way to feed the network was to build a representation by line. Figure 13 illustrate how a batch composed of several smart contracts is built. It shows that to achieve this line-level batch computation, a 2D matrix per line is created where the first dimension is the batch size, meaning that each line corresponds to one program of the batch, while the second dimension is the one corresponding to the input vectors. Thus, the overall structure is a 3D matrix of size : *(Number of lines inside each program, Number of programs in the batch (at least 8), 100)*. Figure 14 shows how the same concept was applied to the usage of *end-points*. It is also illustrating well the increase of the complexity of the model due to the increase of look-up when *end-points* data are used.

Why does each simulation still take a lot of time to be computed ? The main limitation of our model and of this batch implementation in terms of computational cost is that the different look-ups break the parallel implementation of the batch. In fact, as for each line, at least one lookup is needed, the parallelization can't be achieved in the usual optimal way. The direct consequence is the drastic increase in the time needed for the computation of the entire model. Thus, the batch process was implemented in terms of the spreading of the gradient but an effective parallelization was impossible to create due to the intrinsic method used in our model. This is the reason why the computational time required for one simulation is high even if the dataset size and the DL architecture are reasonable in term of computational cost.

Fig. 14: Building of the Batches using *end-points* :
Diagram illustrating the computation of the batches with a model applying lookup over *end-points* using as source of data the line-level embeddings previously described of dimension (1,100) created with the LSTM network



V. EXPERIMENTS & RESULTS

In this section, answers to the following research questions are described :

- *RQ1* : Can a Deep Learning model work for the vulnerability detection task?
  - *RQ 1.1* : How does the proposed approach behave with the Corpus of Solidity contracts as input source?
  - *RQ 1.2* : How does it scale if more information is added to the input ?
  - *RQ 1.3* : Is the information added in the middle of the model by the usage of *end-points* data (corresponding to previous lines) useful ?
  - *RQ 1.4* : Does the model give interpretable results?
- *RQ 2* : How can the model performance be improved?
  - *RQ 2.1* : How does the proposed program representation with a corresponding deep learning model influence vulnerability detection task? An equivalent question would be to ask how does the EP model behaves compared to baseline and similar vulnerability classifiers?
  - *RQ 2.2* : Is the increase of model complexity useful with this dataset?

To guide the reader into the comprehension of this section, Figure 15 summarizes the issues faced and the answers brought to them.

Fig. 15: Summary of the motivations and the implemented answers to the issues faced during the answering of the above research questions

Issues Faced	***Solution***
Understand the Performances of the proposed approach	**Statistical Analysis** of the data at the **token level**
Add **interpretability to the results**	**Attention Weights** Analysis
Show that added ***end-points*** **information allows an increase of the performances**	Setting of **an experiment with synthetic data** of different form
Implement a **robust comparison** to **assess the interest** of our method	Implementation of several **baselines using different kind of input**
Understand the behavior of the models with different **input data quality**	**Add operators** information to the input data

A. Metrics

As our input is qualified as an unbalanced dataset, our goal is to reach the highest precision as possible. More specifically, our main considered metric is the F1 score for the reasons explained below. To describe and understand the level of performances reached by our designed models, 7 different metrics were used. The usage of the first 5 metrics was motivated by [8] while the last two were used to analyze the importance of the threshold choice :

- False_Positive_Rate = FPR = $\frac{FP}{FP+TN}$, where FP = False Positive and TN = True negative. It corresponds to the probability of falsely rejecting the null hypothesis for a particular test. The ideal FPR is 0.
- False_Negative_Rate = FNR = $\frac{FN}{TP+FN}$, where TP = True Positive and FN = False-negative. It corresponds to the proportion of the individuals with a known positive condition for which the test result is negative. The ideal FNR is 0.
- True_Positive_Rate = TPR = Recall= $\frac{TP}{TP+FN}$. It calculates the ability of a model to find all the relevant cases within a dataset. The ideal TPR is 1.
- Precision = P = $\frac{TP}{TP+FP}$. It calculates how precise/accurate our model is out of those who are actually positive. Ideal P is 1.
- F1-Score = F1 = $\frac{2*P*TPR}{TPR+P}$. F1 Score is needed when you want to seek a balance between precision and recall scores and when there is an unbalanced class distribution. The ideal F1 score is 1.
- Average Precision = Area Under the Curve (AUC) of the Precision-Recall curve. It explains the trade-off between the true positive rate and the positive predictive value for a predictive model using different probability thresholds. The ideal score is 1.
- ROC score = AUC of ROC Curves. It summarizes the trade-off between the true positive rate and false-positive rate for a predictive model using different probability thresholds. The ideal score is 1.

B. RQ1 : ***Can a Deep Learning model work for the vulnerability detection task?***

RQ 1.1 : How does the proposed approach behave with the Corpus of Solidity contracts as an input source?

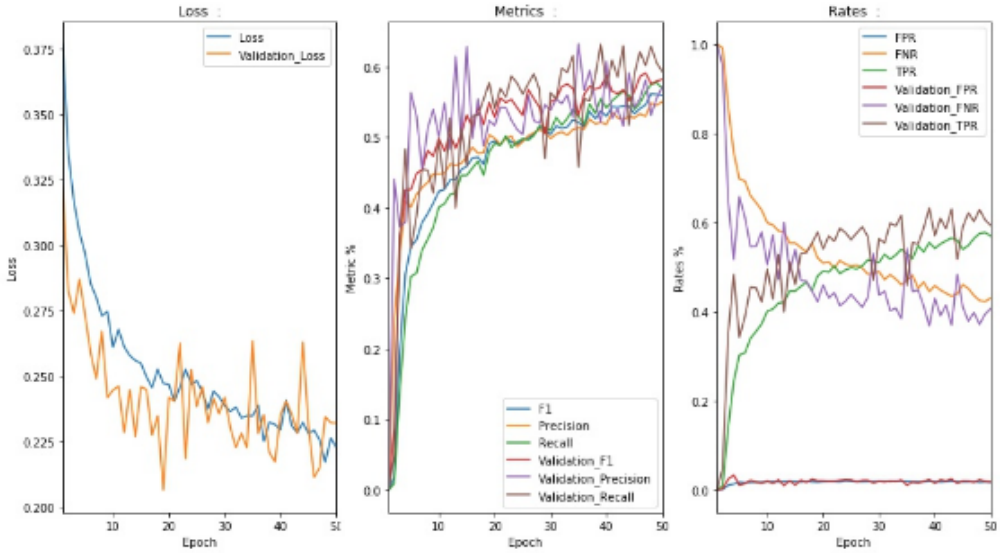
Experimental Hypothesis : *A DL model is truly learning if the train loss decrease in function of epochs, if the validation loss follows a similar evolution and if metrics curves tend towards their ideal values before reaching a plateau.*

Experiment setup :
The used dataset corresponds to the output of the previously described preprocessing pipeline using a subsampling process of 50%. This sampling process of the negative data tries to account for the highly unbalanced distribution of labels. This dataset is consequently made of 4086 smart contracts formed by 238 203 lines/labels containing 3.55% of positive vulnerability cases. These lines are flagged as being the sources of the three classes of vulnerabilities described in Section *II.D.Vulnerabilities in Solidity programs*. The task of our model is to conduct a binary classification at a line level to predict the presence or not of a vulnerability. The input is our designed code representation where main building blocks are CDPs for each token. This input was randomly split into 3 sets : the test set corresponding to 30% of this dataset. The 70% remaining is further divided into a 70%-30% training and validation set : thus the train set corresponds to 49% and the validation set corresponds to 21% of the entire dataset. The loss used during the training is the weighted cross-entropy loss, chosen for its ability to deal with unbalanced input. Our models were implemented using PyTorch version 1.0.

As explained in Section *IV.C.Input Representation*, several parameters needed to be set up to structure the input : the maximum number of tokens per line was defined as 4, and the length of each CDP corresponding to each token was set up as 16. Several other values of these parameters have been investigated and results are displayed in the *Appendix* Section. After observing different distributions of the data, these numbers have been set as the default parameters of the proposed method. Besides, as explained in Section *IV.H.Batches computation*, the filtering of batches formed with less than 8 programs of the same length is done. This number was also chosen after an investigation of the data.

Results :

Fig. 16: Learning and Scores curves during the EP model's training phase for train and validation set



To answer the research question, our EP model was trained on the corpus of Solidity contracts. Figure 16 displays different learning curves of the training phase in terms of epochs. The first graph shows the loss in function of epochs for the training and the validation set. As expected for a working DL model, the training loss is decreasing and the validation loss behaves similarly but noisier. By definition of our validation set, this experimental observation makes sense. Besides, overfitting behavior is not observed. Thus, this first figure displays an expected form of learning curve.

The two last plots of Figure 16 show the evolution of the different metrics used to evaluate the performance of our model per epoch. Again, thanks to these visualizations, the good learning behavior of our model can be observed. In fact, in the third graph, the 3 metrics follow the same evolution for both the train and validation set. The scores increase quickly during the first epochs and continue to increase at a lower rate for later epochs. It means that our model learns a method to discriminate lines with or without vulnerabilities. The fourth graph represents just other metrics that also possess the expected behavior. The FNR stays extremely low as our dataset is unbalanced. This specific metric illustrates the possible issues that can be faced in a work made on an uneven dataset.

Fig. 17: EP Model's ROC curve and Scores curves defined in function of the classification threshold

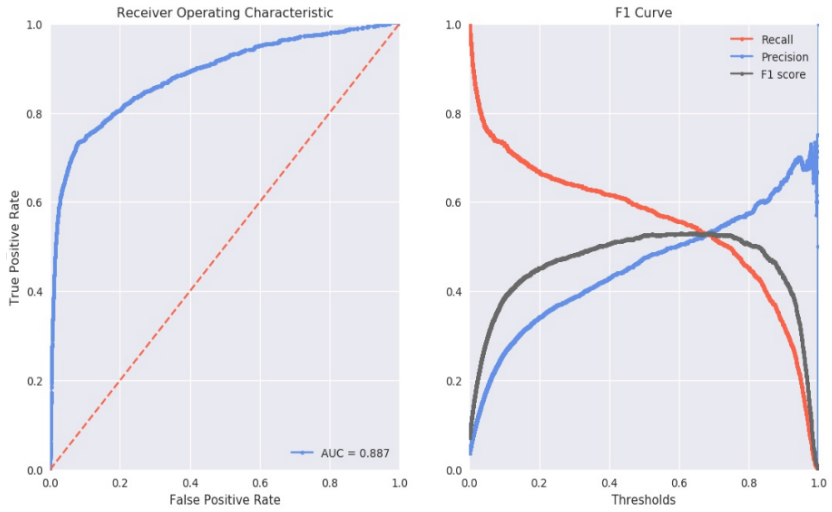


Figure 17 mainly comes from the fact that our model predicts probabilities to each class and then uses them in a softmax function to turn them into binary labels with a default threshold define as 0.5. However, probabilities may be interpreted using different thresholds. Changing this parameter can allow a change in terms of performance. This kind of approach is useful when the cost of one error outweighs the cost of other types of errors.

The first plot of Figure 17 displays the TPR in function of FPR. This ROC curve illustrates the trade-off between both rates for a predictive model using different probability thresholds. This graph also illustrates the robust learning behavior of our model when the threshold of discrimination is changed. In this type of plot, a perfect model would be represented by a line that travels from the bottom left of the plot to the top left and then across the top to the top right. A model without any learning power would be represented by the diagonal red line. As can be seen, our models possess the behavior of a skillful model.

The second graph displays F1, precision and recall metrics in terms of the threshold. The interesting conclusion that can be drawn from this graph is that the choice of a threshold superior to 0.5, around 0.7 to be precise, could help our model to improve.

In conclusion, the plots displayed by Figure 16 and 17 illustrate the skillful behavior of our model. These curves prove that our model is able to understand the input and is able to learn statistical patterns to discriminate lines with vulnerabilities from the ones without any issues.

RQ 1.2 : How does the model scale if more information is added to the input ?

Experimental Hypothesis : *If the proposed DL algorithm was working well, the performances of the model on a richer dataset in terms of amount of information should reach more accurate predictions than the ones on the default input source. It would prove that the proposed approach is able to learn context from the code.*

Experiment setup :

By using default settings previously defined in Section *IV.D.Overview of the Model Architecture*, the operators inside the line were not considered as a token. By taking them into account, a new input dataset is created. This input is qualified as augmented because more information is added to it. In this new configuration, operators just become paths of length one, where the string forming the path is just the operator itself. Padding is then used to format it as all CDPs required a constant length. Our EP model is trained with the new augmented source of data to analyze the impact of this addition of information.

To understand the changes induced by the addition of operators as tokens, a statistical investigation about CDPs is done. In fact, as described in Section *IV.B.Usage of Abstract Syntax Tree*, a line is formed by several tokens that are linked to their CDPs formed thanks to the AST representation. Thus, the label of the line can be associated with the corresponding tokens and consequently with the corresponding CDPs. It means that one label is displayed by 4 paths, as in our case, the same default conditions as the one described in Section *V.B.RQ1.1* have been used. Aggregations of the corresponding 4 paths, build as described in Figure 9, are used as input to the model. The corresponding aggregations of the 4 paths corresponding to different labels classes were stored and an analysis of the unique subset of the aggregations forming each class was done. In particular, intersection and union between the ones implied in positive labels and negative labels were studied. Figure 19 summarize our finding for the default dataset while 20 show the results on the augmented dataset. In these tables, the column called *Number of aggregation of 4 CDP paths* indicates the number of paths' structures contained in the corresponding set. The column called *Number of unique aggregation of 4 CDP paths* represents the number of unique paths' aggregation in the corresponding set over the maximum number it could have reach. For example, the maximum number of the intersection category corresponds to the size of the smallest set discriminate by vulnerability type. Besides, some statistics about the distribution of the number of tokens per line have been measured in Figure 18 and help to answer the research question.

Figure 21 shows the performance of our models tested on the default and the augmented dataset. Two settings defining the dimension of the allowed number of tokens per line were also tested. As usual, the default settings defined in Section *V.B.RQ1.1* were used.

Results :

Fig. 18: Statistics collected about the number of tokens per lines in the raw input and in the augmented dataset

	Mean	Median	Std
Dataset considering only variables as tokens	2.1	2.0	1.2
Dataset considering variables and operators as tokens	2.86	2.0	1.91

Figure 18 illustrates the impact of the consideration of operators as tokens in terms of the number of tokens per line of code. In fact, thanks to this change, the mean number of tokens per line increased by 36%. The other impact is the increase of dissimilarities between paths' aggregations forming the positive and negative set. A comparison between Figure 19 and Figure 20 shows that the percentage of unique aggregations of paths implies simultaneously in positive labels and negative labels decrease from 48% to 39% which make the classification easier for the model. It can be concluded that thanks to the addition of the operators' information, the similarity between paths' structure implies in both positive and negative labels decrease.

The increase of the discriminatory power is beneficial as the evolution of the scores of Figure 21 induces. Indeed, for both settings of considered tokens per line, when operators are added to the dataset, the F1-Score of the EP model increase from 6%. The standard deviation for each setting was calculated and the results are considered as significant. It can be concluded that the addition of operators into the set of considered tokens induce an increase of performance of the EP model by decreasing the similarity between path implies in both positive and negative labels.

Fig. 19: Statistics about the collection of CDPs' aggregations corresponding to 4 tokens in each line implied in negative and/or positive labels for the raw default input

Vulnerability	Number of Aggregations of 4 CDP Paths	Number of **Unique** Aggregations of 4 CDP Paths
Label 0	83169	8067
Label 1	1556	334
Label 2	357	173
Label 3	1287	161
Intersection of paths aggregation implied in Positive Labels	322	9 (over 161 maximum possible paths) **~ 1.3 % of the union of positive data are common between the 3 vulnerabilities type**
Union of paths aggregation implied **Positive** Labels	3200	626 (over 668 maximum possible paths)
Intersection of the paths aggregation implied in **Positive** and **Negative** Labels	2665	300 (over 626 maximum possible paths) **~48% of the data implied in positive labels are also implied in negative labels**

Fig. 20: Statistics about the collection of CDPs' aggregations corresponding to 4 tokens in each line implied in negative and/or positive labels for the augmented dataset considering operators as tokens

Vulnerability	Number of Aggregations of 4 CDP Paths	Number of **Unique** Aggregations of 4 CDP Paths
Label 0	80944	12232
Label 1	1614	420
Label 2	441	257
Label 3	1228	186
Intersection of paths aggregation implied in Positive Labels	316	7 (over 186 maximum possible paths) **~ 0.8 % of the union of positive data are common between the 3 vulnerabilities type**
Union of paths aggregation implied **Positive** Labels	3283	819 (over 863 maximum possible paths)
Intersection of the paths aggregation implied in **Positive** and **Negative** Labels	2512	324 (over 819 maximum possible paths) **~39% of the data implied in positive labels are also implied in negative labels**

Besides, the same analysis was applied to not aggregated paths (corresponding results can be seen in the *Appendix* Section) and it was concluded that 88% of the paths implied in the positive labels were also applied in negative labels. It underlines the important similarity between the different samples of the dataset which induces the low amount of information contained in the input. From the table displayed in Figure 20, it can be seen that the amount of information present in the data stays low but lager than for the default input data without operators. It also means that by adding even more information, the difference in terms of performances should also increase. Thus, one easy way to improve the model performances is to take into account even more tokens. For example, the nodes forming the AST representation that are not variables or operators could be taken into account.

Fig. 21: Comparison of the results obtained using the EP model on the raw input and on the augmented dataset considering operators as a tokens

Model Type	Max Var	F1	Std on F1 Score	Precision	Recall
Endpoints with Operators	4	0.53	1.8%	0.54	0.50
Endpoints to compare	4	0.47	1.8%	0.46	0.48
Endpoints with Operators	8	0.59	1.6%	0.59	0.60
Endpoints to compare	8	0.53	1.5%	0.51	0.55

RQ 1.3 : Is the information added in the middle of the model by the usage of end-points data (corresponding to previous lines) useful ?

Experimental Hypothesis : *If information only remains in end-points corresponding lines and if the rest of the information contained in the data is destroyed, the proposed DL approach should be able to classify a number of vulnerabilities proportional to the amount of information given by the end-points lines. According to our approach, the best results should be found with the usage of all previous lines information.*

Experiment setup :

To answer this question, a synthetic dataset has been created. The idea of this new input source is to keep the same structure as our raw data but to change the amount of information encoded inside. To build it, during the formation of our representation, the raw input data is changed. The method used is simple : the real raw data is copied, lines according to their labels are selected and a transformation at the scale of the CDPs, which means at a token level is applied. The indexes of *end-points* are not modified to be able to use them to bring the same usual amount of information. The main goal of this manipulation is to locate information to one specific category and study the behavior of our models consequently.

The analysis of the behavior of the proposed method on an input where information only remained inside *end-points* is done by designing a specific experiment. All the paths of the entire input data were randomized. This manipulation induces the destruction of all information contained inside the data. Then, an iteration on each line of the programs was conducted, and according to the labels of the line (only positive lines were transformed) a constant pattern, which was a defined constant list of paths, was injected into the *end-points* lines linked to the positively labeled current line. This action allows the information to only remains in the paths corresponding to the *end-points* lines. Be aware that, with this process, the current line is formed by only random paths. Also, in the designed experiment, a differentiation between the previous line and farther away lines has been done. The previous line represents short dependencies (usage of a token twice in two neighboring lines) between tokens while farther away lines represent long term dependencies (call of the variable into another function). Details about the difference between these two categories of lines are explained in Section *IV.Method*. As usual, default parameters described in *Section V.B.RQ.1.1* have been used. The pattern added to bring information is formed by a first path of length 16 only composed of a unique token, a second path of length 16 only composed of a unique token different from the first path, and finally composed of a third and fourth padding path. The experiment will be divided into 4 simulations as shown in Figure 22. To be clear, let's take the example of *Previous Line Experiment* : in this case, if the *end-points* corresponds to the previous line, the pattern is injected into the previous line. The *end-points* corresponding to farther away lines will be randomized as the current line. The other experiments are just defined by a change in the subset of line targeted by the injection of the pattern. Then, the EP model is trained and tested on these different synthetic input dataset where information is situated at different strategic locations.

Fig. 22: Experimental Procedure used to build the different syntethic datasets

Settings	Previous Line	Farther Away Lines
Previous Line Experiment	pattern1	random
Both Method Experiment	pattern1	pattern1
Farther Away Lines Experiment	random	pattern1
Noise Experiment	random	random

Results :

Before analyzing the performance of the model, an investigation on the distribution of the *end-points* is done to understand the amount of information brought by them.

Figure 23 displays the proportion of paths implies in different *end-points* cases. This analysis is crucial to be able to understand the results of the different simulations. A line is made of 4 tokens and each one of them is referred to one *end-points* index. This reference is called *end-points* information. If an index corresponds to the previous line, the corresponding path goes to the previous line category. If the index corresponds to a past line that is not the previous line, it goes to the farther lines case. If the index is 0, it goes to the null case. From Figure 23, the unbalanced property of the dataset can be observed. Besides, the null proportion is approximately 60% inside the positive set which means that the amount of information encoded into *end-points* is not large. The last interesting fact is that the proportion of previous lines cases is twice bigger than the farther line case.

Fig. 23: Distribution of the *end-points* paths, which means at a token level, on the raw dataset

Subset of Data	Specific Case of Endpoints information	Count	% inside the positive set	% in the entire set
Positive	Farther lines cases	4506	12,41%	0,46%
Positive	Previous lines cases	10386	28,61%	1,06%
Positive	Null cases	21412	58,98%	2,19%
Positive	Number of total positive Endpoints cases	36304	100,00%	3,71%
Negative	Total number of negative Endpoints cases	941276		96,29%
Tot	Total number of Endpoints	977580		100,00%

Fig. 24: Statistics about the type of *end-points* information encoded by the 4 CDPs for each line on the raw dataset

Labels type	Count
Previous Line	6335
Farther Away Lines	2898
Previous Line + Farther Away Lines	1871 **(~67% of overlapping information)**
Positive Labels	9076
Negative Labels	244395

Figure 24 displays the implication of *end-points* indexes at a line level while the previous figure was created at a path level. A current line is said to be implied in father lines cases when one out of 4 of the *end-points* indexes linked to it refers to a past line that is farther away than the previous line in the code. In the same way, a current line is said to be implied in previous lines cases when one out of 4 of the *end-points* indexes linked to it make referenced to the previous line in the code. The main conclusion, given by Figure 24, is that 67% of the lines using *end-points* indexes are implied in both the previous line and the farther away cases. This observation represents an overlap between the information brought by the two sets and thus decreases the positive effect of using both sources of information at the same time.

Fig. 25: Experimental results coming from EP model trained on the different synthetic datasets

Experiment	F1	Std for F1 score	Precision	Recall
Previous Line Experiment	0.23	1.2%	0.35	0.17
Both method Experiment	0.28	2.1%	0.33	0.21
Farther Away Lines Experiment	0.25	1.7%	0.37	0.19
Noise experiment	0	0%	0	0

Figure 25 shows the results for the different designed experiments. The first observation is that the score achieved by the *Farther Line Experiment* is higher than the *Previous Line Experiment*. It means that farther lines *end-points* contain more information than the previous line information even if the set of indexes corresponding to the farther lines is twice smaller. Long term dependencies form easier patterns to recognize for our proposed method and are thus classified more easily.

Besides, from the design of our model, the hypothesis is that the EP model captures more context information and thus should overcome the PL performances, modeled by the *Previous Line Experiment*. According to Figure 25, *Both method Experiment* obtains a better score than *Farther Line Experiment* and *Previous Line Experiment*. This observation corresponds to the expected behavior described in the experimental hypothesis. The difference of the scores obtained with both sources compared to the ones made with only one subpart of the *end-points* information is small. This fact can be explained by the overlap of the information described in Figure 24, which induces a high similarity inside the main part of the context brought by both sets. It can be concluded, as the EP model performs better with the father lines information and as it is even better when the entire *end-points* set is given, that the information bring by the *end-points* is definitely useful and allows a better comprehension of the code.

To conclude, even if *end-points* information is small, and even if the overlap between the previous line and the farther away line reduce the amount of information bring by this set, the addition of both stays valuable as it helps the model to learn in a more performant way and thus help it to understand the context more easily. A way to increase performances of our proposed method would be to increase the proportion of *end-points* information encoded in the input data.

RQ 1.4 : Does the model give interpretable results?

Experimental Hypothesis : *An interpretable model is a model that can explain its predictions. Consequently, in our case, if the model is interpretable, it should be able to identify some of the causes of the predicted vulnerabilities.*

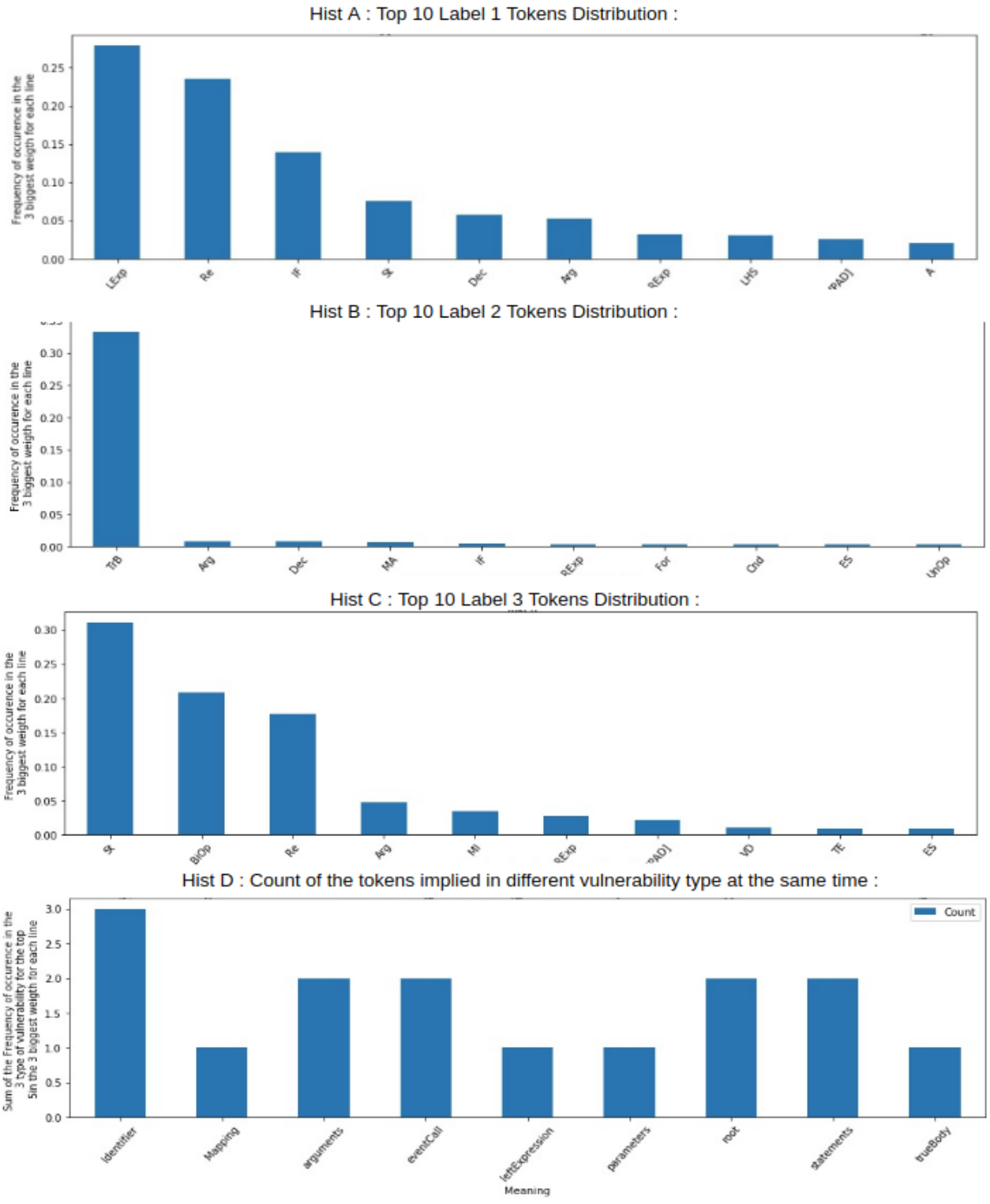
Experiment setup :
Refer to Section *IV.E.Attention Mechanism* to obtain more theoretical information about attention mechanisms. In this experiment, attention weights, which correspond to the weights created by the attention layer inside the model, were collected. To be precise, for each token inside a line, the 3 highest weights corresponding to 3 AST path nodes were selected. Knowing that the maximum number of tokens per line is 4, it means that per line a number between 1 and 12 weights were selected and linked to the node they represent with the usage of a dictionary. Besides, the entire collection of the coefficients was saved in another dictionary. Thanks to that, the normalization of the weights was achieved. Each attention weights obtained during the test phase of our model was divided by its overall occurrence in the entire dataset. In fact, all the quantities used in the production of the results correspond to a relative importance measure. This method allowed to give more importance to tokens that are less frequent in each type of vulnerability while tokens that are always present are more penalized. It means that if a token is present in a high frequency in the entire data it will be less important than a token present only 10 times. With this process, more importance is given to rare tokens able to create a vulnerability only due to their presence. Another criterion of selection was set thanks to the study of the distribution of the entire collection of weights : a minimum value needed to be reached by the weights to be selected. This threshold was set as 0,075. The goal of this manipulation was to filter out meaningless weights. Then, classic statistical analysis was conducted on the gathered collections.

Results :

Complete results are displayed in the *Appendix section*, while the more important results are illustrated in Figure 26. The histograms A, B, C represent the relative importance of the Top10 tokens implied in each type of vulnerability. It means that from these graphs, a broad idea about which tokens are causing which type of vulnerabilities can be obtained. Consequently, these 3 graphs summarize the causes of each vulnerability type which allows the model to gain interpretability. Indeed, some causes can be identified : type 1 is mainly due to *expression* and *root* nodes, type 2 is mainly due to *trueBody* nodes and type 3 is mainly due to *statements* and *parameters* nodes. Besides, a comparison was done with the relative importance of each token implied in label 0 (which means no vulnerability). All the tokens are more implied in label 0 which makes sense due to the unbalanced dataset. It means that found causes need to be nuanced because the tokens identified to create vulnerabilities are also building blocks of well-functioning code. This observation is logical as it corresponds to the intrinsic properties of the vulnerability detection process and is the reason explaining why this task is difficult to succeed.

In addition, an investigation about the implication of specific tokens into several vulnerability types at the same time was done. Histogram D is illustrating this search where a count corresponding to vulnerability type per token is displayed. This count represents the number of times each token is implied into a vulnerability. It means that the maximum score a token can have is 3 and it would mean that this token is one of the causes for the 3 different studied types of vulnerabilities at the same time. If the count is 2, it means that the token is implied into two vulnerabilities type so it could be implied in type 1 and 2 or type 1 and 3 or type 2 and 3. From this search, the importance of the identifier nodes is underlined in the 3 types at the same time. Besides, by doing the same analysis for only two kind of vulnerabilities, some common causes for both types can be identified : more important common causes of type 1 and 2 are *statements* and *eventcall* nodes, of type 1 and 3 are *statements* nodes and of type 2 and 3 are *argument* and *rigthExpression*.

Fig. 26: Attention Weitghs Analysis Results



C. RQ2 : ***How can the model performance be improved?***

RQ 2.1 : How does the proposed program representation with the corresponding deep learning model influence vulnerability detection task? An equivalent question would be to ask how does the EP model compared to baseline and similar vulnerability classifiers?

Experimental Hypothesis : *If the proposed method reach better performances than baselines, proposed program representation with the corresponding DL model influence vulnerability detection task in a positive way.*

Experiment setup :

To answer this research question, the performance of our two models, PL and EP, are compared with different baseline scores. In fact, showing that our designed models perform better than baselines on the same task would prove the advantage of using our described method made with the input representation based on AST mixed with the *end-points* information injected inside the model. Several baselines were tested on several different kinds of input. In fact, trained most classes of classifiers available on Scikit-learn were used [59]. The studied and compared baseline used to classify vulnerabilities were the Logistic Regression model, the Random Forest model, Decision Tree Classifier, Gaussian Naive Bayes model, and SVC model. In the results, only the best scores of these baselines are shown. The best performance is always reached by the Decision Tree Classifier. Scores corresponding to each baseline model can be found in *Appendix* Section.

Two different input sources were used in combination with the baselines :

- ***Bag of words - Path nodes (BOW-Node)*** : The simplest considered input baseline was build using a bag-of-tokens present in each line, which means that a dictionary of all the unique nodes forming CDPs was used. In this baseline, only raw information of CDP composed the input. There is no temporal structure that allows the evaluation of the interest in the time-related data.
- ***Bag of words - Paths (BOW-Path)*** : The other source of input is similar to the BOW-Node one. However, instead of using a dictionary of all the unique nodes, a dictionary comprising all the unique paths of all the lines appearing in the training set was used. With this process, the value of CDP information considering its temporal structure is measured. BOW-Node is made at a node level while BOW-Path is made at a path-level. This can give us a broad idea of the amount of information contained in just the path features.

A third baseline was also created to assess the skills of our model : ***Vulcan No End-points - Vul-NoEP*** : This model corresponds to our designed pipeline without any usage of previous lines information. It thus only corresponds to a DL model made of a Bi-LSTM layer with an attention layer. It corresponds to the suppression of the line-level embedding of tokens *end-points* which are input to Network 4 in 10. It allows a comparison with our model to infer if the usage of the developed line embeddings affects positively the prediction of the overall model. A lower performance for this model compared to our implementation using *end-points* is expected to prove the effectiveness of our approach.

For the PL model and EP model, the default settings described in section *Section V.B.RQ.1.1* have been used. To find the standard deviation, several simulations were run and statistics were created on their results.

Results :

Fig. 27: Scores comparison between implemented baselines on the BOW-Node input

Baseline	F1	Precision	Recall	# 0 labels	# 1 labels
Logistic_Regression	0.17	0.10	0.71	54731	18588
Random Forest	0	0	0	73319	0
DecisionTree	0.41	0.56	0.32	71833	1486
GaussianNB	0.07	0.03	0.98	3716	69603
SVC	0.33	0.61	0.22	72355	964

The first conclusion that can be drawn is that the majority of baselines have low performances with the BOW-Node input source (Figure 27) while one of the baselines stands out : Decision Tree classifier. In fact, the F1 score reach by this model is quite high for a baseline. It means that a certain amount of information is contained into AST's nodes even without temporal dependencies and is well understood by this baseline classifier. However, it is not sufficient for designing a reliable tool. According to this observation, it would be interesting to study the combination of features learned by our deep model with tree-based models as it was done in [64].

Fig. 28: Scores comparison between implemented baselines on the BOW-Path input

Baseline	F1	Precision	Recall	# 0 labels	# 1 labels
Logistic_Regression	0.32	0.20	0.75	64953	9301
Random Forest	0	0	0	74254	0
DecisionTree	0.43	0.58	0.34	72748	1506
GaussianNB	0.08	0.04	0.94	16038	58216
SVC	0	0	0	74254	0

On the BOW-Path input (Figure 28), both the Decision Tree model and the Logistic Regression model increase their F1 scores. This mainly means that the amount of information contained in paths is greater than the one contained in unique nodes. This suggests that the temporal aspect of the proposed paths is important to their predictability. Besides, the increase of the performances for Logistic Regression is higher than the one for the tree-based classifier. This result is the illustration of the importance of the creation of an adapted input representation for a particular chosen model : thing that has been done in our proposed method.

Fig. 29: Scores comparison between the implemented baselines on the different input sources and between the EP and PL models

Model	F1	Std
Only Negative Prediction	0	0%
BOW-Node F1 Logistic Regression	0.17	0.8%
BOW-Node F1 Decision Tree	0.41	0.9%
BOW-Path F1 Logistic Regression	0.32	0.9%
BOW-Path F1 Decision Tree	0.43	1.1%
Vulcan-No Endpoints (Vul-NoEP)	0.47	1.3%
Previous Line Model (PL)	0.52	1.6%
Endpoints (EP)	0.53	1.8%

Figure 29 illustrates the comparison between baselines scores for each input source and the performances achieve by EP and PL model. It can be concluded that baselines, even if using a richer bag of words as input, are significantly surpassed by our designed EP and PL models. Besides, the Vul-NoEP score is as expected lower than scores PL and EP models. The positive value of our approach that takes into account the information flow between the different lines of code is by consequence proven. Thus, our designed approach mixing representation with CDPs and DL model is capturing more information.

RQ 2.2 : Is the increase of model complexity useful with the default input dataset? In other words, is the increase of complexity due to the usage of the EP model is worth it or should we use the PL model instead ?

Experimental Hypothesis : *If the increase of the model complexity worth it, the performance of the EP model should be significantly superior to the scores obtained by the PL model. If it's not the case, it means that the input source does not have the optimal structure to take advantage of the EP model.*

Experiment setup :

To answer this question, an investigation about CDP is done. The results of statistical investigation made in *Section V.B.RQ1.2* about the number of token per line and the unique paths' aggregation on the input source considering operators were used (Figure 18 and 20).

Besides, the default simulation described in *Section V.B.RQ1.1* is used to obtain the performance of the EP and PL model. The running of each model was repeated 5 times to be able to calculate a standard deviation of the performances. Thanks to this pipeline, the metrics and the F1 standard deviation for both models were obtained. A robust comparison is thus enabled and the answer to the research question can be described.

Results :

Fig. 30: Scores comparison between the PL Model and the EP Model on simulations made with default settings on the augmented dataset

Model	F1	Std of F1 Score
PL Model	0.52	1.6%
EP Model	0.53	1.8%

The first conclusion that can be drawn comes from Figure 30 : both models have the same accuracy when used with our input source considering operators. It can thus be concluded that the increase of complexity due to the usage of the EP model instead of the PL model do not worth it. Finding the reasons for explaining this fact would help us to understand how to improve the EP model.

To find the causes of this observation the Figure 20 is analyzed in detail. Our first conclusion is that within each set of vulnerabilities, the number of unique paths' aggregations is really small meaning that the set of data does not possess a lot of diverse information. In fact, only 12232 paths' structures are forming the negative unique path set over 80944 possible, which only represents 15% of the entire set of negative paths. It means that the useful information is only contained in approximately 15% of the negative dataset. The same fact is observable for each vulnerability of the positive labels even if these subsets are made with fewer paths' embeddings. The diversity in the positive dataset seems higher than in the negative one.

The second conclusion that can be drawn is that each set corresponding to each type of vulnerability is mainly not made by a similar path's embeddings. In fact, Figure 20 shows that the intersection between the 3 sets is made of 316 paths' structure while the union is made of 3283 paths' aggregations. These numbers underline the fact that only a small amount of the concatenated paths implied in positive labels are also implied in the 3 different types of vulnerability. However, the union of the 3 subsets is made with 819 unique different paths' aggregation which means, as before, that the similarity between the positive paths is high.

The fact that both negative and positive paths set are made of similar concatenated paths could be an advantage that our model could use to be able to achieve high classification performances because it could be easier to detect the properties of each set and thus to discriminate them. However, the last line of Figure 20 investigates the overlap of information between the positive and the negative set. The huge part of paths' embeddings implied in positive labels is also part of the one creating negative labels. To be precise, 39% of paths imply in any positive labels are also implies in negative labels. The same kind overlap has been found (Figure 24) inside the *end-points* data that induces a decrease of the expected added amount of information when using the EP model.

This global similarity property of the dataset can be explained by a higher level of the representation : the number of tokens per line. In fact, in our settings, the maximum number of considered tokens per line is 4. However, according to Figure 18, the mean and median number of tokens per line is respectively 2.8 and 2, numbers which are definitely too small to be able to take full advantage of the EP model compared to the PL.

Consequently, these different observations have shown that the same information is creating negative and positive labels, which makes our data extremely hard to discriminate, even for a clever model. The overlapping of the information explains the equal performance of the PL and EP models. In fact, the amount of information added by using the EP model is small due to the inherent similar data contained in the dataset. To conclude, even when the augmented dataset, which is richer than the default one, is utilized as an input, it's not useful to use the EP model and consequently, to add complexity, compared to the PL model. In fact, the current dataset does not have enough discriminatory power. Thus, this described investigation method allows the user to know when the EP model could be used in optimal conditions. This is really useful to be able to deal with the complexity/accuracy trade-off. In addition, as shown in *Section V.B.RQ1.2*, adding information into the dataset could easily improve the performances of the model. EP model could be used with some DSL that fills the described requirements (high number of tokens per line and diversity in the path). In this case, our proposed pipeline could perform extremely well.

VI. CONCLUSION AND FUTURE WORK

This work presents an important first step in detecting vulnerabilities for domain-specific languages and analyzing programs written in Solidity. The semantic-rich features method introduced captures intricate control and data dependencies and successfully classifies 3 types of vulnerabilities. The representation introduced using the AST paths combined with a model using *end-points* information has been shown to outperform baselines scores. The intake of the *end-points* information has been proven with the design of an experiment using synthetic data and with baselines. It allows a better comprehension of the natural structure of the code. Our designed pipeline is thus able to capture more intrinsic code information than other models. Information from program tokens, although semantically incapable of capturing vulnerabilities, increases the accuracy of models. Interpretability was added to the model thanks to usage of the attention mechanism.

Taking into account the operators has shown a significant improvement in the performance. This observation induces that, by adding even more information in the data(with for example the consideration of the function name as tokens) the difference in terms of performances should also increase. Another way to increase the amount of significant data inside the input would be to use the one-hot encoding method. In fact, operators are just paths of length one, where the string forming the path is just the operator itself. By applying the one-hot encoding technique on the entire set of operators, the amount of information should be definitely more important and the performances of the model should be increased.

It would also be interesting to use a new source of input to verify the different findings of this thesis. The source [63] could be used as it consists of source codes of 1.27 million functions mined from open source software, labeled by static analysis for potential vulnerabilities. Trying a corpus of code, with the required structure to take full advantage of the model design, with for example a higher number of tokens per line, or a corpus made of longer programs to have a higher number of *end-points* dependencies, could allow better performance than some state of the art tools. The multi-classification task could also be studied. Besides, binary classification with different label grouping criteria could be investigated. By that, it means that the positive set could be made with different vulnerabilities than the one used during this project. Doing that, it would be possible to understand which vulnerabilities are easier to classify and the reasons why this is the case.

Investigate how our models provide an advantage over extant tools for vulnerability analysis for Solidity would also be useful. To do that, the usage of *Timeout* dataset corresponding to unlabeled data, described in Section *II.C.Labeling Process* would be required. Our own model would be trained on this unlabeled dataset and the predictions would be studied and compared to gain insight about the causes of these specifics labels. A manual investigation could be used to verify our results and to analyze the potentials founded vulnerabilities. The amount of issues found in *Timeout* lines could be significant because it could provide a quick solution to detect vulnerabilities in programs which existing state of the art tools are incapable of providing in real-time.

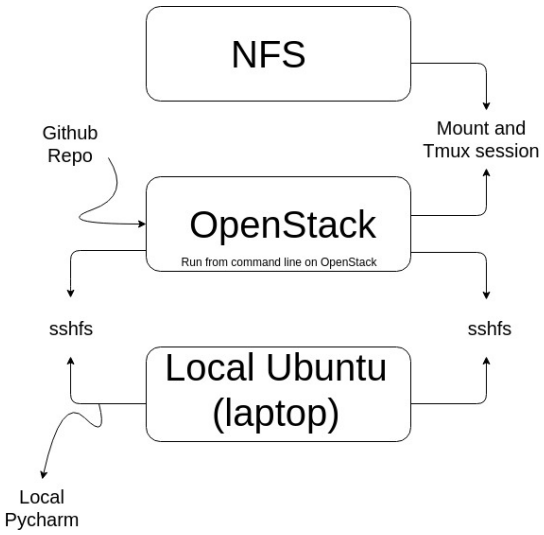
VII. APPENDIX

This section summarizes the different researches done during the entire master thesis. Some were not useful in the building of the previous thesis/paper but could be useful for the continuation of the project. These results are thus less interesting than the previously described ones because they mainly correspond to early stages results found along with the creation of the project.

A. Set up of the environment : NFS, OpenStack, Google Cloud

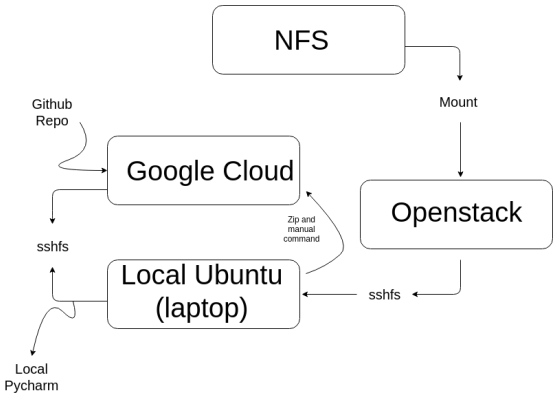
The main goal of this section is to describe the process used to obtain the data and work with it from a private computer. It is definitely useful to save this process to be able to recreate the same kind of environment in a more optimized way.

Fig. 31: Diagram illustrating the settings used with NFS and Openstack to be able to work from private computer



The first working space was made with Network File System (NFS) and Openstack, two tools given by the MIT IT service. NFS is a distributed file system protocol allowing a user on a client computer to access files over a computer network. OpenStack is a set of open-source software for deploying cloud computing infrastructure. As illustrated on Figure 32, to obtain the data stored on NFS, the owner folder of the data was mounted into a created session on one OpenStack cloud. Mounting is a process by which the operating system makes files and directories on a storage device available for users to access via the computer's file system. Thus the NFS data became available on the OpenStack cloud. Then, the GitHub repository containing the source code of our method was clone on this session. The OpenStack machine became a usable working environment to develop our tool. However, as it was more practical to work on the local computer, both directories containing the GitHub directory and the data were also mounted from OpenStack instance to the local environment.

Fig. 32: Diagram illustrating the settings used with NFS and Google Cloud to be able to work from private computer



On the OpenStack cloud, it was not possible to access to any GPU. Thus, to be able to compute our proposed pipeline using GPU to decrease the simulation time, the Google Cloud platform was used. In this case, as previously, the folder containing data from NFS was mounted on an OpenStack session. Indeed, due to the file protection policies of NFS created by MIT, it

was not possible to mount the NFS folder directly on a Google Cloud instance. The data directory was also, as previously, mounted on the local computer. Then, from our local environment, this folder was downloaded to the used Google Cloud instance. The GitHub repository was also cloned to the machine. Consequently, a working environment was set up on the instance and allowed access to GPUs. Finally, to be able to work from our local computer, the GitHub repository was mounted from Google Cloud instance to the local environment.

Thanks to these two processes, working environments using OpenStack or Google Cloud were created to allow effective work with practical settings.

B. Enable GPU usage

Our proposed model was firstly implemented on CPUs. Then, to decrease the huge amount of time needed per simulation, the GPU computation was enabled for our model. To be precise, as our model was implemented with Pycharm, the library Cuda was used. Figure 33 shows the results of the time analysis done on one simulation of our PL model using the raw data. The conclusion is that GPU computation allows a decrease of the running time : it save 66% of the initial simulation time. In fact, the diminution of time achieved goes from approximately 13 hours to 4.5 hours for 50 epochs.

Fig. 33: Time Analysis on the PL model with and without GPU

Number of epochs	CPU/GPU	F1	Precision	Recall	Loss	Time [s]
1	CPU	0.16	0.21	0.13	0.02	953
1	GPU	0.16	0.25	0.12	0.01	320
10	CPU	0.29	0.41	0.22	0.01	9002
10	GPU	0.21	0.76	0.12	0.01	3233
50	CPU	0.43	0.42	0.45	1.145e-05	46639
50	GPU	0.41	0.45	0.38	0.01	15827

C. PL model using a function level representation

In this section, the PL model was used with a different input representation compared to the previously described input structure in Sections *II.Background* and *IV.Method*. Instead of using a corpus of Solidity contract, the collection of the function forming this entire set of contracts was used. Indeed, the first dimension of the representation corresponding to each program described in Figure 9 was transformed into another one made at a function-level iterating on the different functions making these contracts. Inside each function, a maximum number of lines per function was selected. In our case, a maximum of 10 lines was selected. The negative subsampling process was also implemented at this function-level during this line selection. Thus, compared to before, the batch computation is easily enabled and the filtering out of some function due to the batch method was not required. The studied dataset is consequently bigger. Otherwise, the same kind of simulation and the same settings were used alongside this section. The raw dataset, without operators as tokens was set as input source. However, the amount of line in the data was bigger as a similar negative subsampling method was implemented with a lower filtering out percentage. The goals of this section are to analyze and understand in depth the performances of the PL model using this new kind of input. It was implemented at an early stage of the project and was used as a first step to assess and to convince ourself about the value of the proposed approach.

1) PL model analysis: The first step's goal was to create a default score to allow a robust comparison with the different simulations results found along the research process. Figure 34 shows the default results obtain with the following defined default settings :

- Optimizer : SGD
- Number of epochs : 50
- Learning rate : 0.01
- Maximum number of nodes per path called path length : 16
- Labels consider in the positive class : 1, 2 and 3
- Maximum number of tokens per line : 8

Fig. 34: Results of the PL model using the default settings

Model	F1	Precision	Recall
PL	0.47	0.60	0.39

2) Path Length Optimization: One of the crucial parameters was the maximum number of nodes allowed per path chosen for the input formatting process. As explained previously in Section *IV.C.Input Representation*, each line possesses a maximum number of tokens which is set to 8 in this section. These tokens are linked to a corresponding AST representation giving birth to CDPs. The CDPs' length was the studied parameter of this subsection. Thus, to summarize, our input was made of Solidity source code divided into functions that can possess 10 lines maximum. These lines made of 8 tokens could be linked to 8 CDP's of different length. Figure 35 displayed the results of the PL model used with the previously described defaults settings with different values of path length.

Fig. 35: Results of the path length analysis using the PL model on the raw dataset

Path Lenght	F1	Precision	Recall	label_0	label_1
16	0.43	0.52	0.37	42682	518
32	0.57	0.64	0.51	42640	560
64	0.54	0.55	0.54	42511	689

The path length corresponding to 32 is the optimal parameter. This observation makes sense as, compared to 16, the CDPs contain more information that can be understood by our model. In terms of the F1 score, the path length of 64 is approximately equivalent to the one of 32 but it was decided to not keep 32 as optimal because it required a lot more computation power. The trade-off between computation power needed and the performance of our model was not worst it for a path length of 64.

3) Randomization of the paths: In the PL model, at ths stage of the implementation, the lookup inside the model was searching for previous paths and not previous lines as did on the final implementation of the PL described in section *IV.Methods*. By doing this selection of past information at a path level, several CDPs could have been selected to corresponds to the previous path's CDP. To select the corresponding path, two methods were implemented : the first one was based on a random choice of one path in the set of previous CDPs (which correspond to the label *'WITH Randomization'* in Figure 36) while the other process was based on the selection of the closer path in term of distance achieved by looking at the proportion of common nodes in the CDPs (which correspond to the label *'WITHOUT Randomization'* in Figure 36). This random choice analysis was made with the default parameters presented previously and with different path length. Figure 36 shows the results illustrating of the influence of the combination of the path length and the randomness of the paths' selection parameters. The best configuration for this early stage PL model was found for a maximum number of nodes per path of 32 and with a non-random selection of the previous information. This configuration was set as the default one for the following simulation made during the study of the PL model.

Fig. 36: Results of the study about the combination of the path length and the randomness of the paths' selection parameters using the PL model on the raw dataset

Random Choice	Path Length	Number of epochs	F1	Precision	Recall	0_label	1_label
With	16	50	0.43	0.52	0.37	42682	518
Without	16	50	0.62	0.65	0.59	42537	663
With	32	50	0.57	0.64	0.51	42640	560
Without	32	50	0.65	0.72	0.58	42612	588
With	64	50	0.54	0.55	0.54	42511	689
Without	64	50	0.62	0.66	0.58	42607	593

4) Parameters Optimization : Learning Rate, Optimizer and Number of Hidden Unit: A study about parameters optimization was done for several of them. Results are displayed in Figure 37. Thanks to these observations, the search dimension was decreased and the combinatory grid search between the parameters was tried on a reasonable search space. The usage of designed tools for optimized grid search was consider until the paper [68] was encounter. To be precise, the learning rate was set to 0.01 and a search for the optimal parameter in term of optimizer was conducted in the following set ['SGD', 'Adagrad', 'Adam'] and in term of hidden number of nodes was conducted in the following set [100, 500, 1000]. The other parameters were kept as the defaults one. Results are summarized in Figure 38.

Fig. 37: Learning Rate, Number of hidden unit in the network and Optimizer Grid Search Analysis

Learning Rate	F1	Precision	Recall
0.1	0.03	0.66	0.01
0.05	0.41	0.41	0.40
0.01	0.47	0.57	0.39
0.005	0.41	0.42	0.39
0.001	0.38	0.42	0.35

Number of hidden units	F1	Precision	Recall
10	0.31	0.46	0.23
50	0.41	0.47	0.36
100	0.42	0.51	0.35
200	0.43	0.42	0.43
300	0.46	0.50	0.42
1000	0.44	0.43	0.45

Type of classifier	F1	Precision	Recall
SGD	0.41	0.38	0.43
Adadelta	0.35	0.52	0.27
Adagrad	0.43	0.40	0.46
Adam	0	0	0
AdamW	0	0	0
Adamax	0.01	0.16	0.01

Fig. 38: Combinatory Grid Search Analysis on Optimizer and Number of hidden nodes parameters

Optimizer	Number of hidden Units	Random Choice	Path Length	Number of epochs	LR	F1	Precision	Recall
SGD	100	Without	32	50	0.01	0.66	0.69	0.63
Adagrad	100	Without	32	50	0.01	0.63	0.64	0.61
SGD	500	Without	32	50	0.01	0.65	0.66	0.65
Adagrad	500	Without	32	50	0.01	0.65	0.67	0.64

From these results, the optimal parameters were set and considered as the default parameters for the continuation of our study: learning rate = 0.01, optimizer = SGD and number of hidden nodes = 100.

5) Weights influence: As it has been explained in the dataset description of Section *II.D.Vulnerabilities in Solidity programs*, one of the main issues to deal with is the fact that the dataset is unbalanced with too many negative labels compared to the positive ones. To deal with that, a weighted loss was used. It means that a higher weight is attributed to the part corresponding to positive labels to try to counterbalance its lower proportion. Figure 39 describes the influence of these weights on the performances of our model.

Fig. 39: Weights Influence Analysis on the PL model's performances

Weights	Random Choice	Path Length	Number of epochs	F1	Precision	Recall	0_label	1_label
None	Without	32	50	0.60	0.66	0.55	42626	574
[9,1]	Without	32	50	0.67	0.65	0.69	42548	652

The influence of the defined weights is visible and positive meaning that it helps our model to perform well. Thus, [9,1] weights were set as the default weights parameter. In addition, it can be observed that simulation without weights improves precision while creating bad recall scores which makes sense when you look at the definition of these both metrics. In fact, without a weighted loss, almost all data-points would be predicted as negative sample due to the unbalanced property of the dataset. It means that the recall score, which is the number of actual positive labels getting predicted right, would be very poor, while those few that do get predicted as positive would be right, which involves a high precision. These results has shown that our weighted loss is working fine and that its usage improve the performance of the model.

6) Label Grouping: In this section, the performances of our model on several different targets were studied. In fact, by default, the positively labeled vulnerabilities were the ones of type 1,2 and 3 cases while all the other ones were categorized as negative. Thus, an investigation was conducted to analyze the impact of the change of the defined positive targets. It has been tried for example to set all vulnerabilities as positive labels or to classify only one type as positive labels. Figure 40 summarizes the results of this analysis.

Fig. 40: Labels Grouping Analysis on the PL model

Positive Vulnerabilities	Weights	Random Choice	Path Length	Number of epochs	F1	Precision	Recall	0_label	1_label
[1]	[9,1]	Without	32	50	0.79	0.82	0.75	42950	250
[2]	[9,1]	Without	32	50	0.58	0.68	0.50	42998	202
[3]	[9,1]	Without	32	50	0.45	0.49	0.43	43057	143
[1,2]	[9,1]	Without	32	50	0.71	0.74	0.69	42713	487
[2,3]	[9,1]	Without	32	50	0.54	0.57	0.51	42843	357
[1,3]	[9,1]	Without	32	50	0.69	0.72	0.65	42808	392
[1,2,3]	[9,1]	Without	32	50	0.63	0.64	0.61	42537	663
[1,2,3,4]	[9,1]	Without	32	50	0.58	0.54	0.63	42381	819
[1,2,3,4,5,6,7,8,10,11]	[9,1]	Without	32	50	0.65	0.74	0.58	42577	623

From Figure 40, it can be concluded that the *Integer Overflow* (type 1) vulnerability is the easiest one to detect. The score obtained is really high for this task. Besides, it can be observed that detecting the *External Call To Fixed Address* vulnerability (type 2) is harder and that the *Exception State vulnerability* (type 3) is even harder. When these different labels are grouped by pairs as positive labels, the scores followed the same logic. Another fact that can be added is that the score on detecting the entire set of vulnerabilities is not bad. For the following experiments of the section, the labels [1,2,3] were kept as the default ones but these results need to stay in our mind when the interpretation of the results of the model will be developed.

7) Multi-classification analysis: During our investigation, a multi-classification task was also asked to our model. By looking at the data, it can be observed that this task should be really difficult for the model as the data is extremely unbalanced for the majority of labels' types. Three different multi-classifications were tried : all the labels were studied at the same time, and also two different subsets of labels were considered while all the remaining lines were set as negative class. The two considered subsets were the one corresponding to the most abundant classes which were [1,2,3] and [1,2,3,4]. The package *metrics* from *sklearn* was used to obtain the F1, precision and recall scores on this multi-classification task. More specifically, the macro average metric was chosen. Figure 41 displays the results for the different described settings:

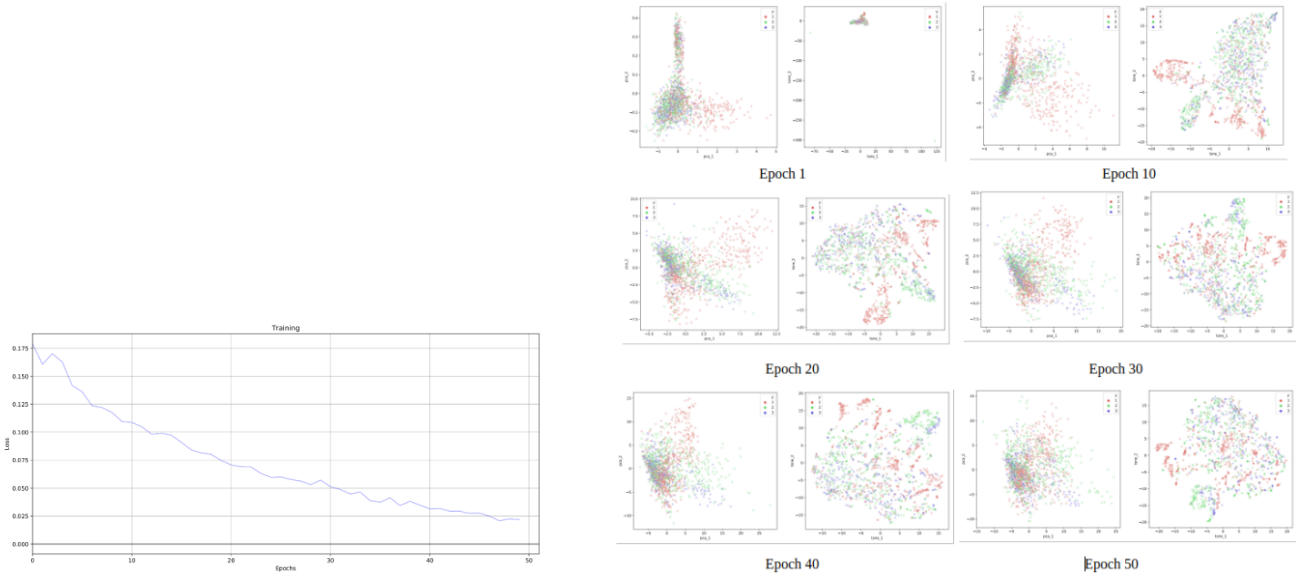
Fig. 41: Multi-classification Analysis on the PL model

Type of classification	F1	Precision	Recall
Multi for all	0.84	0.79	0.81
Multi for [1,2,3]	0.84	0.80	0.82
Multi for [1,2,3,4]	0.83	0.81	0.82

The founded results are surprisingly really good. However, these observation are explained by the predominance of the nicely classified *Integer Overflow* (type 1) vulnerability due to its higher occurrence and also by the fact that high scores are achieved easily as some classes are poor in terms of the number of vulnerabilities in the data. A change of metrics have been considered and was studied during the development of the project to overcome this issue. According to these results, it would be extremely interesting to investigate the performance of the EP model on this multi-classification task.

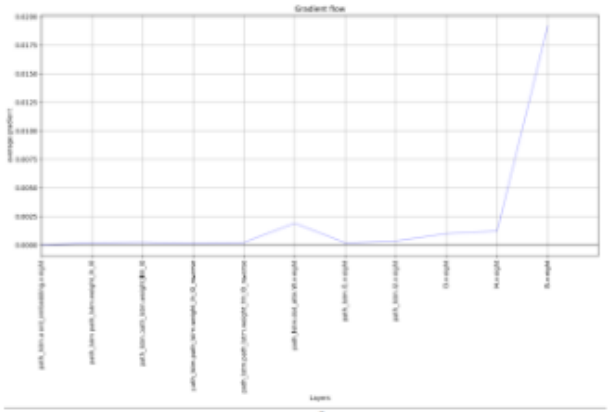
8) Visualization: To visualize and understand our approach, different methods were implemented and tried on the PL model using the default settings. Figures 42 and 43 show the different visualizations used in our study. The created plots were mainly made to verify the good behavior of our model. Indeed, the first plot, on the left part of Figure 42, corresponds to the loss in function of the epochs for the training part of our model. This curve indicates a learning model that corresponds to the wanted observation. The second plot, on the right part of Figure 42, represents the different embeddings for some specific epochs. To understand and assess the qualities of our designed embeddings, a dimension reduction was applied to them. Two plots are displayed per epoch : the left one is done with a PCA while the right one is done with t-SNE. In a perfectly working model, clear clusters should appeared near the end of the training. It's not the case here even if some clusters corresponding to *Integer Overflow* vulnerability (red ones) are more visible than the others.

Fig. 42: Visualization methods : Training Loss, Embeddings Visualization

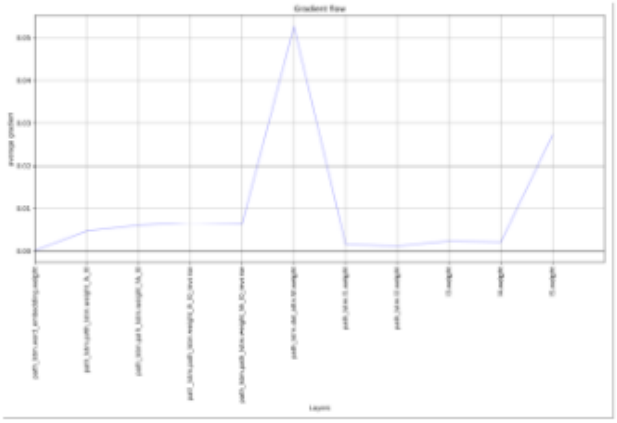


The last plots displayed in Figure 43 represent the gradient values through the different layers of our model for different epochs. These plots give us more insights to understand our model and to check his good behavior. For example, this visualization underlines the importance of the attention layer and of the last layer because, for each one of these layers, the gradient's value is higher compared to the other part of the network.

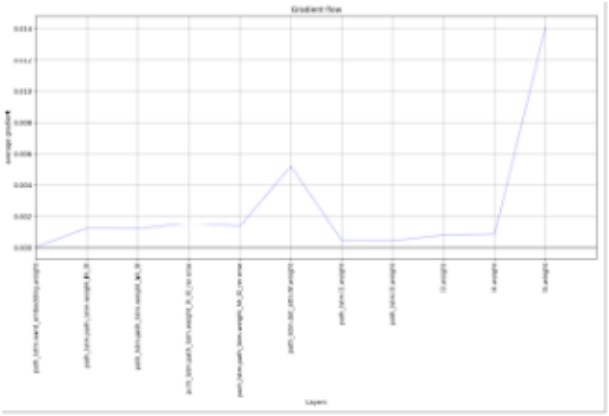
Fig. 43: Visualization methods : Gradient Spreading Curve



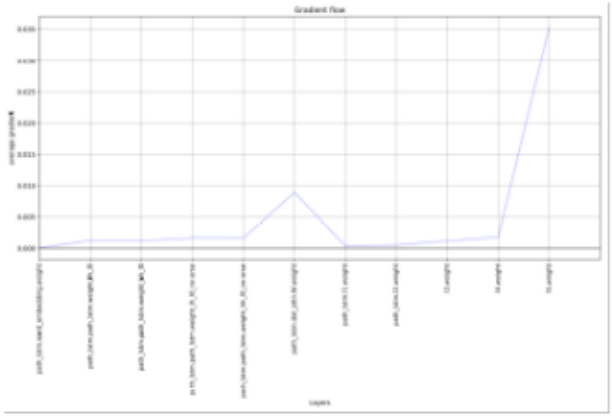
Epoch 1



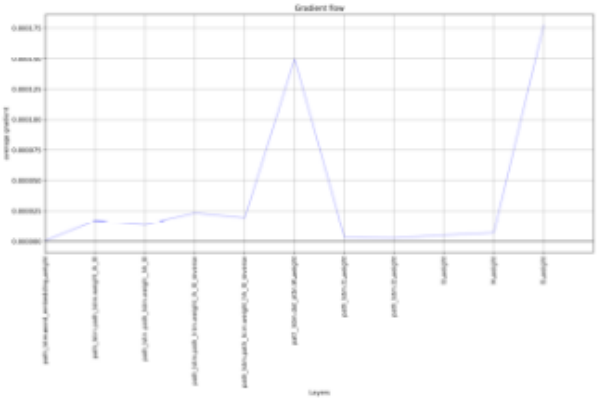
Epoch 10



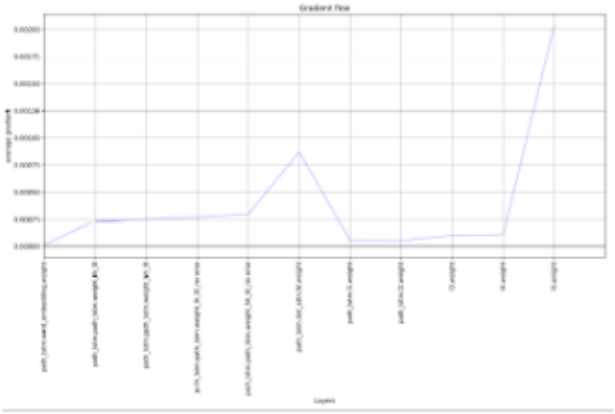
Epoch 20



Epoch 30



Epoch 40



Epoch 50

D. EP model analysis and observed behavior

In this section, the input described in Section *II.Background* and *IV.Methods* was used. The method described follows exactly the described approach of Section *IV.Methods*.

1) Learning Rate Optimization: During the investigation of the model, it has been found that the learning rate was an important parameter to be able to converge to minima. Consequently, a grid search focused on this parameter was done.

Fig. 44: Learning Rate Grid Search for the EP model

Learning Rate	F1	Precision	Recall	FPR	FNR	0_label	1_label
0.00001	0.009	0.07	0.005	0.005	0.99	20091	107
0.0001	0.04	0.11	0.02	0.01	0.97	19563	318
0.001	0.16	0.52	0.09	0.006	0.90	20314	275
0.01	0	0	0	0	1	20125	0

According to Figure 44, the most meaningful learning is achieved with the learning rate of 0.001. This learning rate is good at the beginning of the training. However, during the learning process, it sometimes happens that the model does a step back in terms of loss value around 0.35. This behavior is caused by the fact that the learning rate is too big at this phase of training. Thus, to fix this issue, the usage of a more complex optimizer compared to SGD that can adapt its learning rate and its momentum during the training phase (Adam, Adagrad) is useful. Also, the usage of an optimizer that uses a momentum can improve the speed of the optimization process in concert with the step size, improving the likelihood that a better set of weights is discovered in fewer training epochs. Consequently, Adagrad was selected as the best optimizer for our model because it allowed the usage of an adapting learning rate and momentum that help to find global minima. Other optimizers like Adam and Adamax were tried but with less success.

2) Batch Normalization: As the model is dealing with NLP input, meaning that our input matrix refers to tokens, the normalization of the raw input data could not be done as it would not make any sense. In fact, it would induce the loss of the main part of our information. In DL, normalization is used for several reasons. Let's take an example : before the normalization of the inputs, the weights associated with these inputs could vary a lot because the input features present different ranges varying from let's say 0 to 40000. To accommodate this range of differences between the features, some weights would have to be large and then some have to be small. In the case of larger weights then the updates associated with the back-propagation would also be large and vice versa. Because of this uneven distribution of weights for the inputs, the learning algorithm keeps oscillating in the plateau region before it finds the global minima. To avoid the learning algorithm spend much time oscillating in the plateau, the normalization of the input features is used such that all the features would be on the same scale. Then, since the inputs would be on the same scale, the weights associated with them would also be on the same scale. It would thus help the network to train. In our specific case, inputs are already on the same scale and consequently normalization is, by definition, not required.

One other way to apply normalization in our case is to use the batch normalization process between the layers of our model to rescale the hidden stay to help our model to learn from it. The activation values, meaning the output of the different layers, act as an input to the next layers present in the network. So, the previous potential input transformations do not matter at this stage (whether normalized or not), because the activation values would vary a lot as the information spread into the network. To bring all the activation values to the same scale, the normalization of the activation values such that the hidden representation called embeddings in our case was used to get improvements of the training speed.

Fig. 45: Batch Norm effect on the EP model's perfomances

Train Batch Size	Test Batch Size	Path Length	Percentage of Sub-sampling	Bacth Norm	Input Vocab with numbers	Epochs	F1	Precision	Recall	FPR	FNR	TPR
8	8	16	0.5	with	yes	50	0.54	0.52	0.57	0.02	0.42	0.57
8	8	16	0.5	without	yes	50	0.54	0.48	0.61	0.02	0.38	0.61

Figure 45 displays the performances of the EP model using or not the batch normalization process. In this case, the usage of the batch norm made the training more meaningful in a smoother way when looking at the training curves. The batch norm allows a decrease in precision and an increase in the recall while keeping the F1 score constant. Alongside the project, in some cases, batch normalization has been shown to be mandatory to allow convergence of the model.

3) Input Vocab: A method to improve performances of the EP model was tried : the number of different tokens in the vocabulary dictionary was increased. In fact, some numbers were added to the nodes identifiers to be able to differentiate nodes that were following themselves in the AST representation. For example, some nodes were called 'BExp0' for the first node of this type in the representation, 'BExp1'for the second node of this type in the representation and 'BExp2' for the third node of this type in the representation. This change induced the creation of a vocabulary made by 250 different nodes while the default one had only approximately 60 vocabulary words. The effect of this change was analyzed and results are shown in Figure 46.

Fig. 46: Input Vocabulary impact on the performances of the EP model

Train Batch Size	Test Batch Size	Path Length	Percentage of Sub-sampling	Bacth Norm	Input Vocab with numbers	epochs	F1	Precision	Recall	FPR	FNR	TPR
8	8	16	0.5	with	yes	50	0.54	0.52	0.57	0.02	0.42	0.57
8	8	16	0.5	with	no	50	0.53	0.51	0.55	0.02	0.44	0.55

The addition of the numbers into the nodes identifiers seems to slightly improve the model performance. The same observation is achieved with the Decision Tree Classifier model. The results corresponding to this baseline are displayed in Figure 47.

Fig. 47: Input Vocabulary impact on the performances of the Decision Tree Classifier model

Model Train	F1	Precision	Recall	FPR	FNR	TPR	0_label	1_label
DecisionTree_Test WITHOUT number in node identifier	0.438	0.589	0.348	0.009	0.651	0.348	69896	1565
DecisionTree_Test WITH number in node identifier	0.475	0.600	0.393	0.010	0.606	0.393	69688	1773

4) Usage of small batches: As previously described in Section *IV.H.Batches computation*, the implemented batches method induced a filtering out process of the batches of size inferior to the batch size. The effect of this filtering out was studied by preventing it and results are presented in Figure 48. The results illustrate the negative effect of preventing the filtering of small batches. It makes sense as some batches are made of only 1 program. In this case, the spreading of the gradient is not optimal and minima are hard to find.

Fig. 48: Effect of the filtering of small batches on the EP model

Model Type	Max Var	Train Batch Size	Test Batch Size	Path Length	Percentage of Subsampling	Batch Norm	Input Vocab with numbers	Epochs	F1	Precision	Recall	FPR	FNR	TPR
Endpoints	4	8 no filtering	8 no filtering	16	0.5	without	no	50	0.48	0.45	0.53	0.02	0.46	0.53
Endpoints	4	8 filtering	8 filtering	16	0.5	with	no	50	0.53	0.51	0.55	0.02	0.44	0.55

5) Label Grouping: The same method applied in Section *VII.C.7.Multi-classification analysis* was used for the EP model. Results are displayed in Figure 49. Just as a reminder : label of type 1 corresponds to *Integer Overflow* vulnerability, labels of type 2 corresponds to *External Call To Fixed Address* vulnerability, label of type 3 corresponds to *Exception State* vulnerability and finally label of type 4 correspond to *Multiple Calls in a Single Transaction* vulnerability.

Fig. 49: Label Grouping Analysis on the EP model

Group	F1	Precision	Recall	FPR	FNR	TPR	Loss	0_label	1_label
[1]	0.498941	0.422422	0.609314	0.0167416	0.390686	0.609314	0.137111	38125	1115
[2]	0.0875	0.132075	0.0654206	0.00116524	0.934579	0.0654206	0.0560515	39531	53
[3]	0.438679	0.41517	0.465	0.0105125	0.535	0.465	0.156497	37312	672
[4]	0	0	0	0	1	0	0.0220428	39504	0
[1,2]	0.461619	0.452577	0.47103	0.0123753	0.52897	0.47103	0.200505	42870	970
[1,3]	0.536435	0.532609	0.540317	0.0171021	0.459683	0.540317	0.220257	40208	1472
[1,4]	0.500813	0.462462	0.546099	0.0137685	0.453901	0.546099	0.154755	38849	999
[2,3]	0.376119	0.373665	0.378606	0.0129896	0.621394	0.378606	0.19146	40637	843
[2,4]	0.0187793	0.0952381	0.0104167	0.000472355	0.989583	0.0104167	0.0930875	40395	21
[3,4]	0.467815	0.52439	0.422259	0.00624917	0.577741	0.422259	0.144035	37564	492
[1,2,3]	0.527626	0.49319	0.567232	0.0231036	0.432768	0.567232	0.290232	38422	1762
[1,2,4]	0.48431	0.479793	0.488912	0.0133209	0.511088	0.488912	0.213968	37667	965
[1,3,4]	0.53897	0.519084	0.56044	0.0194605	0.43956	0.56044	0.228792	38732	1572
[2,3,4]	0.414634	0.510601	0.349034	0.00677659	0.650966	0.349034	0.186148	41138	566
[1,2,3,4]	0.521935	0.540414	0.504679	0.0175238	0.495321	0.504679	0.252996	39367	1497

It can be observed that the easiest label to predict is type 1 and then type 3. The F1 scores for label 2 and 4 are near 0. This was expected for type 4 due to its extremely low proportion in the data while for label 2, the difficulty of classifying it can be explained by the intrinsic nature of this type of vulnerability : it corresponds to an error in assertion, that is really difficult to infer. Then, when two labels are set together as positive labels, the performance follows the same relation in terms of scores. As it's easier to classify classes 1 and 3, the best scores are reached when these two vulnerabilities are labeled as positive while if one of the other types is grouped with 1 or 3, the scores decrease. Moreover, the amplitude of the decrease is proportional to the score reach by each category alone. It means that, as the score of label 1 alone is higher than the one for label 3 alone, the scores for union of label 1 and 2 is higher compared to labels 2 and 3 grouped together. Also, scores for the union of labels 2 and 4 stays low. The same relations are observed for classification with 3 labels taken in the positive set. In fact, F1 scores for [1,2,3] and [1,3,4] are nearly the same and higher than score of [1,2,4] which is higher than the one for [2,3,4]. The main conclusion is that this analysis has shown the existence of easier labels to predict. Besides, some labels have common information that helps to predict vulnerabilities because by grouping two labels as positive, the scores achieved exceeds the ones found for each label separately.

E. Grid Search Analysis on Batch Size, Subsampling rate and Path Lenght for Baselines, EP and PL model

In this section, the usual program representation described in Section *II.Background* and *IV.Methods* was used. A Grid Search analysis was performed for 3 hyperparameters : batch size (8, 16 or 32), the path length (16 or 32) and the proportion of subsampling on the more negative programs (0.8, 0.5, 0). The subsampling at a programs level allowed to obtain faster simulation in terms of time and also allowed to increase the percentage of positive samples in the dataset as described previously in Section *IV.A.Preprocessing*. To be able to understand the results of the grid search, an analysis of the different inputs induced by the subsampling and the batch methods was conducted. The table on Figure 50, describes the distribution of the data depending on their labels while the table on Figure 51 displays the quantity of data created due to the combinations of the parameters batch size and percentage of subsampling. The main conclusion is that the input created with a subsampling rate of 80% create a dataset with very few positive labels for each category.

Fig. 50: Distribution of the vulnerabilities in the different dataset made using different subsamlping methods

Proportion of sub-sampling at code level	% of positive sample	% of negative sample	Total Number of Sample	0 labels	1 label	2 labels	3 labels	4 labels
0	1.84 %	98.16 %	500960	491723	3994	1479	3578	186
0.5	3.72 %	96.28 %	238203	229335	3868	1447	3371	182
0.8	7.62 %	92.38 %	67012	61906	2444	1023	1526	113

Fig. 51: Distribution of the vulnerabilities in the different dataset made using different subsamlping percentage and batch sizes

Sub-Sampling rate	Batch Size	# label in Train	# Ignore label in Train	# label in Val	# Ignore label in Val	# label in Test	# Ignore label in Test
0.8	8	18624	13625	5656	8960	9304	10843
0.8	16	11680	19847	2624	12465	5312	15084
0.8	32	8960	24610	32	13948	1024	18438
0.5	8	79280	38551	21704	26923	40272	31473
0.5	16	48416	66545	7296	43634	17376	54936
0.5	32	23584	91365	1024	50270	5856	66104

Figure 52 displays the results of the grid search over the 3 parameters for the 3 kinds of models. It allows an easier comparison by grouping together the results for the 3 models with the same simulation settings.

First of all, an analysis only about the EP model is conducted. Results are better when the percentage of positive data in the set is higher. This makes sense as the dataset is biased and as the number of positive labels is lower for dataset with low subsampling rate than the ones having a higher subsampling rate.
For both the percentage of subsampling, the size of the batches also impacts a lot the scores. In fact, the higher is the batch size, the higher F1, precision and recall scores are. This observation can be explained by the usage of a higher batch size values that induces an higher filtering out of programs which decreases the size of the dataset.
The effect of the path length is less clear. The impact seems to depend on the set up of the experiment. For example, taking into account the trial with a subsampling percentage of 0.5 and with a batch size of 8, an increase of the path length from 16 to 32 has a bad effect on the metrics while with a batch size of 16 and 32, increasing the path length has a positive effect. The explanation is that increasing the batch size enforce our model to train on larger programs and thus, in this case, increasing the path length would make sense as more information could be took into account. Some more simulation needed to be done to prove this hypothesis.

Fig. 52: Grid Search Analysis on Batch Size, Subsampling rate and Path Lenght for Baselines, EP and PL model

Model Type	Train Batch Size	Test Batch Size	Path Length	Submapling rate	F1	Precision	Recall	FPR	FNR	TPR	Loss	0_label	1_label
Baseline	8	8	16	0.8	0.685923	0.709966	0.663455	0.022753	0.336545	0.663455	1.62525	18649	1455
Previous_Line	8	8	16	0.8	0.741078	0.74277	0.739394	0.0227028	0.260606	0.739394	0.282228	7447	657
Endpoints	8	8	16	0.8	0.703178	0.650814	0.764706	0.0344615	0.235294	0.764706	0.245426	7977	799
Baseline	8	8	32	0.8	0.694126	0.731673	0.660244	0.0203267	0.339756	0.660244	1.55653	18699	1405
Previous_Line	8	8	32	0.8	0.690534	0.722999	0.660859	0.0219515	0.339141	0.660859	0.254964	10005	787
Endpoints	8	8	32	0.8	0.732357	0.707851	0.758621	0.0258807	0.241379	0.758621	0.236174	8719	777
Baseline	16	16	16	0.8	0.686722	0.74382	0.637765	0.0184396	0.362235	0.637765	1.55653	18769	1335
Previous_Line	16	16	16	0.8	0.740365	0.752577	0.728543	0.0201782	0.271457	0.728543	0.293699	5963	485
Endpoints	16	16	16	0.8	0.742291	0.675351	0.823961	0.0335891	0.176039	0.823961	0.225439	4733	499
Baseline	16	16	32	0.8	0.689632	0.71947	0.662171	0.0216747	0.337829	0.662171	1.59432	18671	1433
Previous_Line	16	16	32	0.8	0.737819	0.70354	0.77561	0.0308614	0.22439	0.77561	0.240181	4300	452
Endpoints	16	16	32	0.8	0.715393	0.656504	0.785888	0.0344828	0.214112	0.785888	0.294708	4820	492
Baseline	32	32	16	0.8	0.677072	0.736048	0.626846	0.018871	0.373154	0.626846	1.59948	18778	1326
Previous_Line	32	32	16	0.8	0.698413	0.709677	0.6875	0.0267857	0.3125	0.6875	0.23265	2022	186
Endpoints	32	32	16	0.8	0.841121	0.762712	0.9375	0.0301724	0.0625	0.9375	0.339361	906	118
Baseline	32	32	32	0.8	0.698574	0.740821	0.660886	0.0194101	0.339114	0.660886	1.5256	18715	1389
Previous_Line	32	32	32	0.8	0.842975	0.87931	0.809524	0.0104012	0.190476	0.809524	0.289254	2034	174
Endpoints	32	32	32	0.8	0.838196	0.849462	0.827225	0.013882	0.172775	0.827225	0.236374	2022	186
Baseline	8	8	16	0.5	0.475553	0.600677	0.39357	0.0102974	0.60643	0.39357	1.13533	69688	1773
Previous_Line	8	8	16	0.5	0.552705	0.501265	0.615911	0.0260798	0.384089	0.615911	0.238644	37439	1977
Endpoints	8	8	16	0.5	0.549305	0.528267	0.572087	0.0211277	0.427913	0.572087	0.264683	36603	1645
Baseline	8	8	32	0.5	0.478747	0.606576	0.395418	0.0100938	0.604582	0.395418	1.12615	69697	1764
Previous_Line	8	8	32	0.5	0.58467	0.565517	0.605166	0.0197958	0.394834	0.605166	0.245152	38076	1740
Endpoints	8	8	32	0.5	0.523663	0.502655	0.546504	0.0232609	0.453496	0.546504	0.266233	36105	1695
Baseline	16	16	16	0.5	0.476957	0.604308	0.393939	0.010152	0.606061	0.393939	1.13002	69697	1764
Previous_Line	16	16	16	0.5	0.622886	0.55694	0.706546	0.0272877	0.293454	0.706546	0.275479	18012	1124
Endpoints	16	16	16	0.5	0.564444	0.494548	0.65735	0.0322276	0.34265	0.65735	0.304378	19820	1284
Baseline	16	16	32	0.5	0.481399	0.600552	0.4017	0.0105156	0.5983	0.4017	1.13195	69651	1810
Previous_Line	16	16	32	0.5	0.628918	0.571691	0.698876	0.0279142	0.301124	0.698876	0.263452	16496	1088
Endpoints	16	16	32	0.5	0.617053	0.529163	0.739953	0.032684	0.260047	0.739953	0.228167	16705	1183
Baseline	32	32	16	0.5	0.467498	0.587906	0.388027	0.0107047	0.611973	0.388027	1.15612	69675	1786
Previous_Line	32	32	16	0.5	0.674061	0.615265	0.745283	0.0258639	0.254717	0.745283	0.21782	9438	642
Endpoints	32	32	16	0.5	0.665936	0.584615	0.773537	0.035579	0.226463	0.773537	0.255833	5944	520
Baseline	32	32	32	0.5	0.48959	0.601508	0.412786	0.0107629	0.587214	0.412786	1.12567	69604	1857
Previous_Line	32	32	32	0.5	0.665289	0.547619	0.847368	0.0444222	0.152632	0.847368	0.192314	5780	588
Endpoints	32	32	32	0.5	0.694362	0.627346	0.777409	0.0267154	0.222591	0.777409	0.228499	5131	373

Then, an analysis only about the PL model is conducted. These results show the same effect of the different tested parameters as for the EP model. This makes sense as the PL model is a subset of the EP model. Indeed, performances are better when the percentage of subsampling is higher, which means results are better when the percentage of positive data in the set is higher. Also, the positive effect of an increase of the batch size is observed. This effect is more visible when the percentage of subsampling is smaller meaning when our model is run on more data. This makes also sense as changing the batch size when there is not a lot of data can drastically decrease the amount of programs used in our training and testing phases. This observation is in accordance with the fact that using higher batch size values induces that several programs are drop away and are don't use during training and testing.
The effect of the path length parameter is not understandable. For a subsampling rate of 0.8, the impact is negative for batch size 8, neither negative neither positive for batch size 16 and positive for batch size 32. To be precise, the simulation using a batch size of 32 and a path length of 32 is extremely good. However, this observation is due to a huge decrease in the used number of samples and can also coincide with larger programs as explained previously. It would be interesting to work on a dataset made of larger programs. Except this good result, the effect of the path length is really hard to describe and seems to be unpredictable. Consequently, a path length of 16 nodes is set as a default parameter as it decreases the amount of time and computation power needed for the computation. Remember that before, in Figure 35 path length parameter had a huge impact. But in the meantime, a bug was found and fixed in the code. By fixing it, the path length parameter lost his importance.

By doing a comparison between the previously described results obtained with both EP and PL models, it's impossible to discriminate a better one as also explained in Section *V.Results*. However, it can be seen that the EP model usually succeeds to have better recall scores while the PL model seems to succeed to achieve higher precision.

Finally, the implemented baselines were also used in this grid-search. Figure 53 describes the results of the different baselines using a subsampling rate of 0.8 and a batch size of 8, and a path length of 32. The baseline called 'Raw_baseline' does not correspond to a model but corresponds to the most basic baseline that can be created : predictions are only negative values. Similar simulations were done for each set of settings but the entire set of results are not displayed due to its lack of interest. However, the best baselines' scores for each simulation were reported in Figure 52. The best model was always the Decision Tree Classifier from the Sklearn library. Even if the results of this baseline was surprisingly good, our models always succeeded to exceed baselines' performances in term of F1 score and recall.

Fig. 53: Performances of the implemented baselines model for an input defined with a subsampling rate of 0.8, with a batch size of 8, and with a path length of 32

Model Train	F1	Precision	Recall	FPR	FNR	TPR	Loss	0_labels	1_labels
Logistic_Regression_Train	0.325018	0.207263	0.752606	0.235614	0.247394	0.752606	8.16878	34021	12887
Random_Forest_Train	0	0	0	0	1	0	2.61316	46908	0
DecisionTree_Train	0.797724	0.851118	0.750634	0.0107475	0.249366	0.750634	0.994761	43778	3130
GaussianNB_Train	0.142909	0.0769919	0.993519	0.974907	0.0064807	0.993519	31.1422	1111	45797
SVC_Train	0.485256	0.81516	0.345449	0.00641159	0.654551	0.345449	1.91514	45404	1504
Raw_baseline_Train	0	0	0	0	1	0	2.61316	46908	0
Model Test	----	----	----	----	----	----	----	----	----
Logistic_Regression_Test	0.3233	0.206442	0.745022	0.240416	0.254978	0.745022	8.34281	14485	5619
Random_Forest_Test	0	0	0	0	1	0	2.67493	20104	0
DecisionTree_Test	0.694126	0.731673	0.660244	0.0203267	0.339756	0.660244	1.55653	18699	1405
GaussianNB_Test	0.14559	0.0785743	0.989724	0.974335	0.0102762	0.989724	31.0743	492	19612
SVC_Test	0.471291	0.827419	0.32948	0.00576913	0.67052	0.32948	1.97743	19484	620
Raw_baseline_Test	0	0	0	0	1	0	2.67493	20104	0

F. Usage of Synthetic dataset

As previously described in Section *V.RQ1.3*, the creation of a synthetic dataset was implemented to be tested as a noiseless input source on our models. In this section, a description of additional experiments that were run alongside the building of the project is done. The same process as previously described in Section *V.RQ1.3* was used to build the different sources of synthetic data but the format of the added information and the categories targeted by these transformations were changed to test different settings. Figure 54 summarizes the results found on the different inputs. The default parameters used for these simulations were a batch size of 8, a path length of 16, a maximum number of tokens per line of 4 and a subsampling rate of 0.5. Remember that with this kind of settings, a line corresponds to 4 paths made by 16 nodes.

In these experiments, several types of patterns were used and injected to the path corresponding to the positive labeled lines or to the negative labeled lines. Here is a description of each type of pattern :

- *Random pattern :* This kind of pattern corresponds just to a randomization of the entire information in the implied line's paths. In this case, all the information contained in the paths is destroyed.
- *Raw data pattern :* This type of pattern corresponds to the raw data. It means that none transformation has been applied to the corresponding line's paths and consequently that the information naturally contained in them is kept.
- *Identical pattern :* In this case, the two first paths corresponding two one line were set to a list of 1 while the two last paths were randomized. In this case, strong discriminatory power is injected into the paths. Sometimes the value 1 was changed to another number but it never changed the observed effect.
- *Identical Range pattern :* In this case, small randomness is added to the created path. In fact, the paths are changed by injected a pattern of the same length containing numbers defined inside a specific range and created randomly. To be precise, for the lines labeled 1, the first path contained only 1, 2 or 3 numbers while the second path contained only 4, 5 or 6 while the third path contained only 7, 8 or 9. The fourth path was randomized. The same is applied to lines labeled 2 : the first path is made of numbers contained between 1 and 10, the second path with numbers between 10 and 20 and the third path with numbers between 20 and 30. The labeled 3 lines had also the same structure : the first path was made of numbers between 50 and 54, the second path with numbers between 54 and 58 and finally the third path with numbers between 58 and 62. The information localised in this kind of paths contained a high discriminative power but lower than the one of the *Identical pattern* structure.
- *Pattern with noise probability x :* This process corresponds to an addition of noise that can be applied to all the previously described patterns. The x number defines the probability of adding some noise to each path. Then, in the case of noise creation, a random number is generated between 0 and 4 to choose how many tokens' paths are changed. Let's say that the generated number is 2. It means that the two tokens' paths are transformed. For each one of them, two lists of random length made of random numbers are created : one corresponds to the indexes of the node in the path and the second corresponds to the new values of the corresponding nodes. This process has the consequence to add dissimilarities into the data.

Fig. 54: Performances of the models using different settings of synthetic data as input

Model Type	Pattern for Positive data	Pattern for Negative data	F1	Recall	Precsion	FPR	FNR	TPR
Endpoints	Identical Pattern	Random	1	1	1	0	0	1
Endpoints	Identical Range Pattern	Random	1	1	1	0	0	1
Endpoints	Identical Range Pattern	Raw data	1	1	1	0	0	1
Previous Line	Identical Range Pattern	Raw data	1	1	1	0	0	1
Endpoints	Identical Range Pattern	Identical Range Pattern	0	0	0	0	1	0
Endpoints	Random	Random	0	0	0	0	1	0
Endpoints	Random	Raw Data	1	1	1	0	0	1
Endpoints	Raw Data	Random	0.972	0.968	0.977	0.001	0.022	0.0977
Endpoints	Identical Range Pattern	Identical Range Pattern with noise proba 0.25	0.641	0.984	0.475	0.0003	0.524	0.475
Endpoints	Identical Range Pattern	Identical Range Pattern with noise proba 0.5	0.636	0.664	0.611	0.012	0.388	0.611
Endpoints	Identical Pattern	Identical Pattern with noise proba 0.5	0.544	0.752	0.426	0.005	0.573	0.426

From lines 1 to 4 of the table displayed in Figure 54, a hypothesis can be made : the positive data and negative data are too similar to allow our model to be able to classify vulnerabilities with high accuracy. To test this hypothesis, the same pattern for all negative and positive data was set and results can be found in line 5. By doing that, the model is run on identical data. From the results, it can be deduced that this is not the case and consequently that the data are not exactly the same between the two classes. The model is not able to learn anything which makes sense as there is no difference between negative and positive data. Similar results were obtained on our sanity check of line 6 where the entire input data was completely random.

Then, another experiment was tried : one of the categories was randomized while the other kept its raw input. With this process, it can be tested if the model is able to find patterns contained in the raw information of one class (lines 7 and 8

in Figure 54). If the model is able to classify vulnerabilities it would means that the information contained in positive and negative data could be too similar to allow classification. In fact, in both cases, the F1 scores are nearly perfect. Our model is able to classify vulnerabilities with perfect accuracy without the information contained in one of the two classes. Moreover, thanks to line 8, it can be concluded that our model is able to well-classified data with only the information contained in the positive data which represents approximately 3% of the entire dataset. It means that the interaction between the positive and negative data induces a decrease in the accuracy of the model most probably due to a high similarity. However, a perfect score is not obtained for line 8 meaning that some samples are hard to classify because they are made with similar data as the negative ones, which are just noise. It can also be concluded that information contained in the negative set is more important than the one in the positive set. This fact could be due to the unbalanced property of the dataset.

From the previous experiments, the conclusion that our model is learning on mainly similar data can be drawn. This fact was indeed proven by Figure 19 of Section *V.1.RQ1.2*. By knowing this distribution, a reproduction of the EP model behavior was implemented with the synthetic data. To do that noise was added to the paths. In fact, the guess is that negative and positive data are approximately similar but not exactly the same. So to simulate the same structure, positive and negative data were transformed into the same type of *identical range pattern* and the noise was added only to the negative data (lines 9 and 10 on Figure 54). For the noise probability of 0.5, by analyzing the different values of each epoch, a conclusion about the similarity between the simulation with the raw data and with the synthetic one is drawn. The only difference is the shape of the precision curve which is beginning quite high and decreasing during training. To finally succeed to simulate the same behavior, the design of synthetic data with an even more similar pattern was done and corresponds to line 11 in Figure 54. By looking at the learning curves, the similar behavior between the raw data and synthetic data made with identical patterns and with the addition of noise to the negative set is shown. It underlines the main weakness of our project: usage of an input without enough discriminate power between positive and negative classes.

G. Similarity analysis at path-level

As described in Section *V.B.RQ1.2*, to understand the changes induced by the addition of operators as tokens, a statistical investigation about CDPs is done. Remember that a line is formed by several tokens that are linked to their CDPs formed thanks to the AST representation. Thus, each CDP can be associated with the label of the corresponding line. It means that one label is displayed by 4 paths, as in our case, the same default conditions as the one described in Section *V.B.RQ1.1* have been used.

In the previous section, the analysis of the aggregations of the 4 paths corresponding to different labels classes were stored and an investigation of the unique subset of the aggregations forming each class was done. In this section, the same method is used on the non aggregated paths. Figure 55 summarize our finding for the default dataset while 56 show the results on the augmented dataset. In these tables, the column called *Number of CDP paths* indicates the number of paths contained in the corresponding set. The columns called *Number of unique paths* represents the number of unique paths in the corresponding set over the maximum number it could have reach.

Fig. 55: Statistics about the collection of CDPs corresponding to 4 tokens in each line implied in negative and/or positive labels for the raw dataset considering

Vulnerability	Number of Paths	Number of Unique Paths
Label 0	320448	2247
Label 1	6112	221
Label 2	1888	230
Label 3	4880	150
Intersection of paths implied in Positive Labels	8005	30 (over 150 maximum possible paths) **~ 20% of the data are common between the 3 vulnerabilities type**
Union of paths implied **Positive** Labels	12880	476 (over 601 maximum possible paths)
Intersection of the paths implied in **Positive** and **Negative** Labels	12761	417 (over 476 maximum possible paths) **~ 88% of the data implied in positive labels are also implied in negative labels**

Fig. 56: Statistics about the collection of CDPs corresponding to 4 tokens in each line implied in negative and/or positive labels for the augmented dataset considering operators as tokens

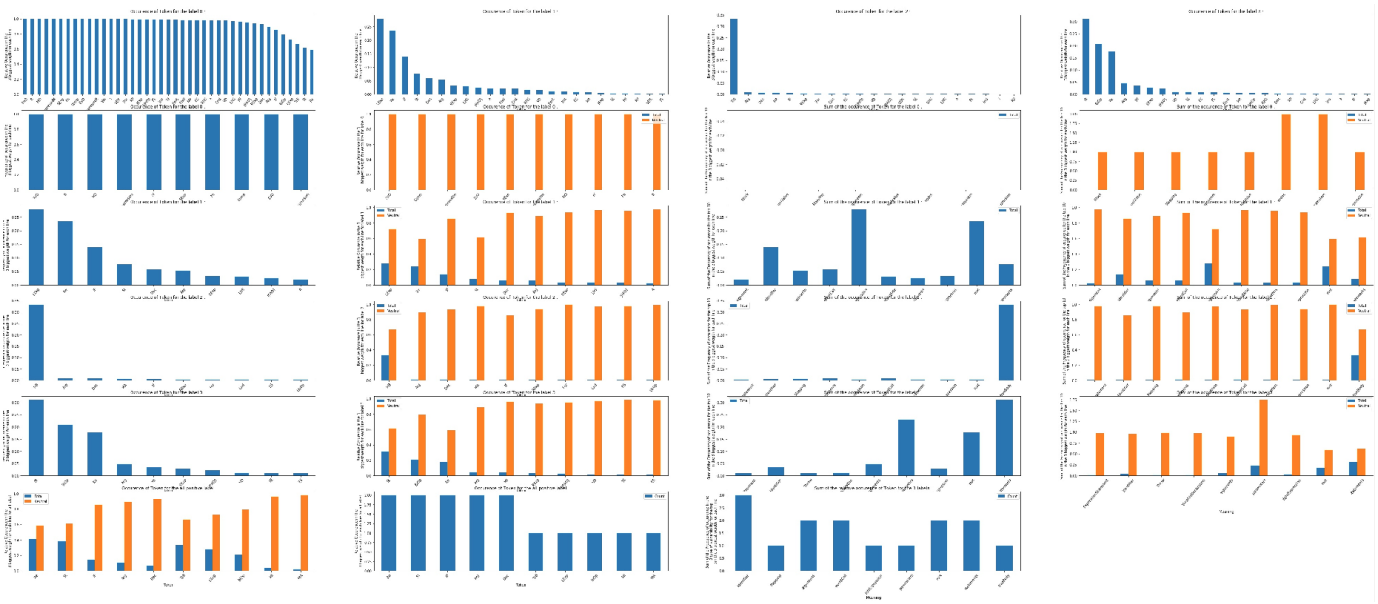
Vulnerability	Number of Paths	Number of Unique Paths
Label 0	322736	3285
Label 1	6128	308
Label 2	2028	261
Label 3	4844	206
Intersection of paths implied in Positive Labels	7573	40 (over 206 maximum possible paths) **~ 19% of the data are common between the 3 vulnerabilities type**
Union of paths implied **Positive** Labels	13000	614 (over 775 maximum possible paths)
Intersection of the paths implied in **Positive** and **Negative** Labels	12053	536 (over 614 maximum possible paths) **~87% of the data implied in positive labels are also implied in negative labels**

From these two figures, two conclusions can be drawn. First, the overlap between paths implied in a negative and in a positive label at the same time is huge. It underlines, as before, the high similarity of the two classes' data and illustrates the intrinsic lack of discriminative power of the input. The second conclusion is that, at this path-level, without the aggregation of the 4 paths forming a line, the addition of the information corresponding to the operators token has no effect.

H. Complete Attention Weights Analysis

The meaningful results of this analysis are shown and explained in Section *V.B.RQ1.4*. Below, the entire set of generated plots can be found.

Fig. 57: Distribution of token's weights implied in each vulnerability type



The plots of the first row of Figure 57 represent the distributions of the tokens for each type of vulnerability according to their weights. Each one of the graphs composing the first row are a distribution of the tokens implies in one particular label. It means that from these graphs, a broad idea about which tokens are causing which type of vulnerabilities can be created. In fact, you can take the top 10 tokens and obtain the 10 tokens that have the biggest influence on which type of vulnerability will be created. Some causes can thus be differentiated.

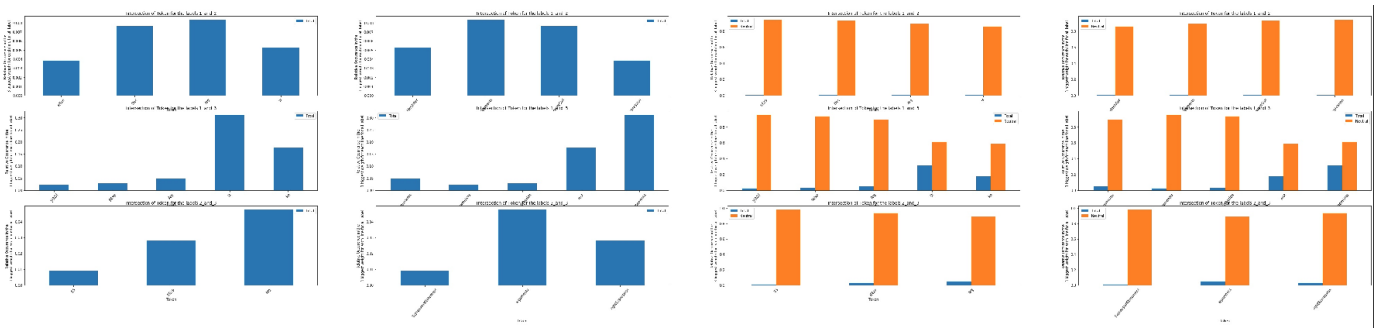
The plots of the second, third, fourth, and fifth row represent the distributions of top10 tokens implies in each label (first and second graphs of each row) and grouped by larger categories (third and fourth graphs of each row). These more general categories of tokens are used to gain interpretability and a comparison is done with the number of tokens implies in label *No Vulnerability* which are represented as orange bins. Thanks to the comparison, it can be concluded that all the tokens are implied with a higher score in label *No Vulnerability*. It thus gives us a way to see the tokens that have really huge importance: the ones that are not so present in *No Vulnerability* labels set while present in the Top10. The second line corresponds to label 0, the third line corresponds to label 1, the fourth line corresponds to label 2 and the fifth line correspond to label 3.

The sixth row corresponds to a study made on all positive data. This row is made of 3 graphs representing a condensed version of the 4 previous rows. In fact, the Top10 of the tokens implied in the entire set of positive labels is shown. The first graph represents the distribution of these Top10 tokens compared with their respective importance in label *No Vulnerability*. Using the same reasoning as before by giving more importance to the ones which are less represented in label *No Vulnerability* subset, the user is able to find the more important tokens for positive labeling. The second graph displays a count per token. This count represents the number of times each token is implied into a vulnerability. It means that the maximum score a token can have is 3 and it would mean that this token is one of the causes for the 3 different types of vulnerabilities at the same time. The third graph represents the same information for the more general categories.

The 3 rows of Figure 58 represent the intersection analysis between the more important tokens for different sets. The main idea is to analyze the tokens implied in two vulnerabilities at the same time. The first row represents the intersection of labels 1 and 2, the second row represents the intersection of labels 1 and 3 and finally, the third row represents the intersection of labels 2 and 3.

The first graph of each row is made of the tokens that are present in the intersection of the 2 different label sets. Thus a conclusion about the common causes of these two labels can be drawn. Then, the second graph of each line displays the same information as the first one but grouped by the more general categories. Finally, the two last graphs of each line represent the

Fig. 58: Intersection analysis of the more important tokens for each vulnerability type



same information but compared with the importance of each token in the negative labels.

Fig. 59: Word Cloud for vulnerability of type 0 and 1

Fig. 60: Word Cloud for vulnerability of type 2 and 3

The main idea of the representation Figures 59 and 60 is that the more important the token is for the corresponding set, the bigger it is written. The first illustration of Figure 59 corresponds to label *No Vulnerability*, the second to label of type 1. The first illustration of Figure 60 corresponds to label of type 2 and finally the second to label of type 3.

VIII. References

[1] European Union Agency for Cybersecurity, Vunerability definition, enisa.europa.eu, [En ligne], *https://www.enisa.europa.eu/topics/threat-risk-management/risk-management/current-risk/risk-management-inventory/glossaryG52 europa.eu*, (accessed January 1, 2020).

[2] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269.

[3] Wikipedia, Hearthbleed, *https://en.wikipedia.org/wiki/Heartbleed*, (accessed January 1, 2020).

[4] By Martin Fowler, with Rebecca Parsons. 2010. Domain Specific Languages.

[5] Dzmitry Bahdanau, KyungHyun Cho, Yoshua Bengio. 2016. Neural Machine Translation by jointly learning to align and translate. *ICLR 2015 as oral presentation*.

[6] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin. Attention Is All You Need. *http://arxiv.org/abs/1706.03762*.

[8] et al. Zhen Li, Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *In 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.*

[9] Mythril. 2017. *https://github.com/ConsenSys/mythril.*

[10] Kalra, S.; Goel, S.; Dhawan, M.; and Sharma, S. 2018. Zeus: Analyzing safety of smart contracts. *NDSS.*

[11] Luu, L.; Chu, D.-H.; Olickel, H.; Saxena, P.; and Hobor, A. 2016. Making smart contracts smarter. *In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 254–269. ACM.

[12] Manticore. 2018. *https://github.com/trailofbits/manticore*.

[13] Solgraph. 2016. *https://github.com/raineorshine/solgraph*.

[14] Solium. 2018. *https://github.com/duaraghav8/Solium*.

[15] SmartCheck. 2018. *https://tool.smartdec.net/*.

[16] Mythril Github, Overflow vulnerability detection, *https://github.com/ConsenSys/mythril/wiki/Integer-Overflow* (accessed December 29, 2019)

[17] Mythril Vulnerabilities List, *https://mythx.io/swc-coverage/*, (accessed January 1, 2020).

[18] Mythril Integer Overflow Vulnerability Definition, *https://swcregistry.io/docs/SWC-101*, (accessed January 1, 2020).

[19] Mythril Integer Overflow Vulnerability Definition, *https://swcregistry.io/docs/SWC-104*, (accessed January 1, 2020).

[20] Mythril Integer Overflow Vulnerability Definition, *https://swcregistry.io/docs/SWC-110*, (accessed January 1, 2020).

[21] Tue Le, Tuan Nguyen, Trung Le, Dinh Phung, Paul Montague, Olivier De Vel, Lizhen Qu. 2019. Maximal Divergence Sequential Autoencoder for Binary Software Vulnerability Detection. *ICLR*.

[22] Chris Cummins, Pavlos Petoumenos, Alastair Murray, Hugh Leather. 2018. Compiler Fuzzing through Deep Learning. *ISSTA*.

[23] Saahil Ognawala, Ricardo Nales Amato, Alexander Pretschner and Pooja Kulkarni. 2018. Automatically assessing vulnerabilities discovered by compositional analysis. *MASES.*

[24] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, Denys Poshyvanyk. 2018. An Empirical Investigation into Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ASE*.

[25] Michael Pradel, Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-based Bug Detection, *CoRR*.

[26] HK Dam, T Pham, SW Ng, T Tran, J Grundy, A Ghose, T Kim, CJ Kim. 2018. A deep tree-based model for software defect prediction. *CoRR*.

[27] Rebecca L Russell, Louis Kim, Lei H Hamilton, Tomo Lazovich, Jacob A Harer, Onur Ozdemir, Paul M Ellingwood, and Marc W McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. *arXiv preprint arXiv:1807.04320.*

[28] Ke Wang, Rishabh Singh, Zhendong Su. 2018. Dynamic Neural Program Embedding for Program Repair. *ICLR*

[29] 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, Marc W. McConley, Jeffrey M. Opper, Peter Chin, Tomo Lazovich. 2018. Automated software vulnerability detection with machine learning. *IWSPA*.

[30] Zheng Gao, Christian Bird, Earl Barr. 2017. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. *ICSE*.

[31] Jacob Devlin, Jonathan Uesato, Rishabh Singh, Pushmeet Kohli. 2017. Semantic Code Repair using Neuro-Symbolic Transformation Networks. *CoRR*.

[32] Yaqin Zhou and Asankhaya Sharma. 2017. Automated Identification of Security Issues from Commit Messages and Bug Reports *FSE*.

[33] Min-je Choi, Sehun Jeong, Hakjoo Oh, Jaegul Choo. 2017. End-to-End Prediction of Buffer Overruns from Raw Source Code via Neural Memory Networks *IJCAI*.

[34] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, Xudong Liu. 2019. A Novel Neural Source Code Representation based on Abstract Syntax Tree *ICSE*.

[35] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 40.

[36] Miltiadis Allamanis, Marc Brockschmidt, Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. *ICLR*.

[37] Hoa Khanh Dam, Truyen Tran, Trang Pham. 2016. A deep language model for software code. *CoRR*.

[38] Lili Mou, Ge Li, Lu Zhang, Tao Wang, Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing.*AAAI-16*.

[39] M. Brockscmidt, M. Allamanis A. L. Gaunt, O. Polozov. 2018. Generative Code Modeling with Graphs. *CoRR*.

[40] Y. Li, S. Wang, T. N. Nguyen, S. V. Nguyen. 2019. Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks. *Proceedings of the ACM on Programming LanguagesOctober Article No.: 162*

[41] K. Wang, 2029. Learning Scalable and Precise Representation of Program Semantics. *CoRR*.

[42] L. Büch, A. Andrzejak. 2019. Learning-based Recursive Aggregation of LSTMs for Code Clone Detection. *10.1109/SANER.2019.8668039*

[43] S. Xu, S. Zhang, W. Wang, X. Cao, C. Guo, J. Xu. 2019. Method name suggestion with hierarchical attention networks. *PEPM 2019: Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*.

[44] U. Alon, M. Zilberstein, O. Levy, E. Yahav. 2018. A General Path-Based Representation for Predicting Program Properties. *PLDI*.

[45] M. Allamanis, M. Brockscmidt, M. Khademi. 2018. Learning to Represent Programs with Graphs. *ICLR*.

[46] M. Pradel, K. Sen. 2017. Deep Learning to Find Bugs, *Technical Report TUD-CS-2017-0295TU Darmstadt, Department of Computer Science,November, 2017*

[47] L. Mou, G. Li, Y. Liu, H. Peng, Z. Jin, Y. Xu, L. Zhang. 2014. Building Program Vector Representations for Deep Learning. *International Conference on Knowledge Science, Engineering and Management*

[48] C. Omar. 2013. Structured Statistical Syntax Tree Prediction. *SPLASH*.

[49] Haijun Wang, Yi Li, Shang-Wei Lin, Lei Ma† , Yang Liu. 2016. VULTRON: Catching Vulnerable Smart Contracts Once and for All. *ICSE New Ideas and Emerging Results*

[50] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. 2009. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. *In Proceedings of the 2009 IEEE/ACM International Conference on automated software engineering. IEEE Computer Society, 605–609*.

[51] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. *In International Conference on Information Systems Security. Springer, 1–25.*

[52] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *In NDSS, Vol. 5. Citeseer, 3–4.*

[53] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. *In International Conference on Machine Learning. 2091–2100.*

[54] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. *In Acm Sigplan Notices, Vol. 49. ACM, 419–428.*

[55] Gursimran Singh, Shashank Srikant, and Varun Aggarwal. 2016. Question independent grading using machine learning: The case of computer program grading. *In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 263–272.*

[56] Srikant, S., and Aggarwal, V. 2014. A system to grade computer programming skills using machine learning. In Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, *1887–1896. ACM.*

[57] Aho, A. V., and Ullman, J. D. 1977. Principles of Compiler Design (Addison-Wesley series in computer science and information processing). *Addison-Wesley Longman Publishing Co., Inc*.

[58] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. Commun. *ACM 56, 2 (2013), 82–90.*

[59] Pedregosa, F.; Varoquaux, G.; and Gramfort, A. e. a. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12:2825–2830.*

[60] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering 42, 8 (2016), 707–740.*

[61] Code2Vec Gihub Link, Attention Mechanism Implementation, *https://github.com/tech-srl/code2vec*, (accessed January 1, 2020).

[62] Dzmitry Bahdanau, KyungHyun Cho, Yoshua Bengio. 2015. Nerual Machine Translation by Jointly Learning to Align and Translate. *Published as a conference paper at ICLR*.

[63] Labeled dataset for software vulnerability detection task : *https://osf.io/d45bw/*, (accessed January 1, 2020).

[64] 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, Marc W. McConley, Jeffrey M. Opper, Peter Chin, Tomo Lazovich. 2018. Automated software vulnerability detection with machine learning. *IWSPA*.

[65] Y. Wainakh, M. Rauf, M. Pradel. 2019. Evaluating Semantic Representations of Source Code. *arXiv:1910.05177*

[66] Ledger Definition: *https://systemsinnovation.io/blockchain-overview-article/*, (accessed January 1, 2020).

[67] Álvaro Rocha, Hojjat Adeli, Luís Paulo Reis, Sandra Costanzo. 2019. New Knowledge in Information Systems and Technologies. *Volume 2, Springer, page 85, section 2.4.Smart Contracts*.

[68] Christian Sciuto, Kaicheng Yu, Claudiu Musat, Martin Jaggi, Mathieu Salzmann. 2019. Evaluating the Search Phase of Neural Architecture Search, *CoRR*.