Literature Review & Project Proposal

Machine Learning for Automated Vulnerability Detection

Sydney Vertigan, Daniel Jones, Samantha Wise, Junfan Huang

January 2, 2022

1 Introduction

Machine Learning is an evolving aspect of science in technology. There is evidence of rapid progress in a wide variety of applications since this concept was first developed in 1959. The focus of the proposed research is to determine whether cybersecurity applications, such as the automation of detecting vulnerabilities in code, are able to reap the benefits of machine learning methods, in particular Deep Neural Networks, or are these methods over-hyped, and systems that are already implemented remain pertinent.

The security of the software we run is becoming an increasing concern, for individuals and businesses alike. Straddling the border between the outside world and our personal computers, web browsers are an important piece of the security puzzle. Over 40% of Mozilla Firefox's CVE (Common Vulnerabilities and Exposures) disclosures have been memory corruption vulnerabilities [1]. These will form the focus of this project. Figure 1 gives an example of a use-after free memory corruption vulnerability. Other examples include buffer overflows, heap corruptions and type confusion vulnerabilities.

```
  char *username;
  char *password;

  username = malloc(...);

  ...
  // memory at "username" location is free'd, making it available
  // for other programs. "username" is now a dangling pointer: it
  // points to a memory address which isn't allocated!
  free(username);
  ...

  password = malloc(...);

  // Uses the memory at "username" even though it isn't owned
  // by this program anymore. This could have anything in it!
  printf("Username: %s\n", username);
```

Figure 1: Even though the `username` pointer has been marked as 'free in memory', a pointer to it still exists in the form of the '`username`' variable. This is a *dangling pointer*, and its use indicates a *use-after-free* bug [10]. This pointer references a location in memory that might be allocated to another variable or even program. See Chris Evans' explanation [2] of how these bugs can be exploited to provide code execution.

The project will start by surveying a variety of machine-learning based approaches to vulnerability detection. The results of the survey will include the advantages and disadvantages of the approaches, involving:

1. amount of training data required,
2. ability to handle uncertainty,
3. transparency of learnt models.

We will then implement the most viable approach identified in our survey to automatically detect a simple vulnerability, and evaluate our results on its utility and potential.

2 Approaches Exploration

2.1 Constraint-based Program Analysis

Between 2003 and 2007, Stanford Computer Systems Lab made considerable progress using static analysis to detect memory corruption bugs in source code. A number of approaches were implemented, extending ideas from compiler design and formal verification to model memory leaks. The Clouseau tool used pointer analysis to model ownership of pointers throughout their lifetime [12]. Archer focused on buffer overflows and memory corruption, approximately halving the false positive rate to 35% [28]. Saturn introduced a summary-based approach to path-sensitive pointer analysis [27]. A trade-off can be seen between the false positive rate and execution time; Saturn took 24 hours to analyse the Linux kernel source code, for example [27]. Recently, tools such as Fastcheck [5] and Saber [25] have made use of sparse graph representations to give a 10x speed up over Saturn with a comparable false positive rate [25].

"What You See Is Not What You Execute" [3] argued that analysis should be performed on machine code rather than source code. In this vain, Feist et al showed good progress in finding use-after-free vulnerabilities in binaries, using a combination of dynamic and static analysis. This work resulted in the disclosure of CVE-2015-5221 in the JasPer JPEG library [11]. The Valgrind tool provides an instrumentation framework for performing dynamic analysis, providing memory modelling plugins [19].

New approaches, which augment programs and tracks memory usage at runtime have proven to be effective. Tools such as SafeCode [8], DANGNULL [16] each develop variants of this approach. DieHard [4] extends this with a probabilistic approach.

2.2 Inductive Logic Programming (ILP)

ILP is described as a symbolic machine learning technique which constructs logic programs from examples. In a cybersecurity context of code vulnerability detection, it has been shown to characterise the general features of a bug, with some flexibility around linguistic variation [18, 22]. This is the approach that has been implemented by Qinetiq when developing a vulnerability detection system to detect patterns in binaries corresponding to bugs [26].

A key strength of ILP is its ability to produce a 'human-readable' output when combined with expert domain knowledge. It is shown to be a well-suited and efficient framework when dealing with small structured datasets and supports continual and transfer learning [9]. We have witnessed glimpses of its potential through its feats in learning to detect buffer overflow attack construction strategies [18] and analysing ransomware attacks [22].

In 2003, [18] illustrated that logic programming is a convenient tool for determining the semantics of attacks. This provides a framework for detecting attacks in formerly unknown inputs. Moreover, we can generate detection clauses from examples of attacks through the machine learning approach provided by ILP. This paper conducted experiments of learning ten different buffer overflow exploit strategies demonstrating that only very few attack examples are required when producing accurate detection rules.

ILP does exhibit limitations in situations where data are either erroneous, noisy and ambiguous which is likely to be the case in this upcoming project. In [22], it was shown that without a sufficient amount of negative examples, the ILP system will tend to initially propose overly-general hypotheses which requires hand-crafting further examples and meta-constraints. This could lead to a situation of outputting a series of increasingly intricate but logically equivalent hypotheses that thwarts the user's attempts to clarify an excessively general clause.

2.3 Neural Networks

A machine-learning approach which has successful cases for vulnerability detection is the use of Neural Networks. [24] relates closely to our project goal in dealing with source code vulnerability detection. [17] utilised Neural networks to detect software vulnerabilities.

In order to perform effective source code vulnerability detection using Neural Networks, a large amount of training examples are required to ensure high accuracy. Russel et al [24] showed that duplicate functions would need to be removed from the data (open-source repositories can have functions repeated across different packages), as these can artificially inflate the data.

Since the most useful input for Deep Learning is real integer values, we would be required to encode the programs into vectors. The more general paper [17] states:

'Programs can be first transformed into some intermediate representation that can preserve (some of) the semantic relationships between the programs' elements (e.g., data dependency and control dependency). Then, the intermediate representation can be transformed into a vector representation that is the actual input to neural networks.'

It seems feature-extraction is extremely important [24], similar to that used for sentence sentiment classification with convolutional neural networks and recurrent neural networks for function-level source vulnerability classification. This approach has been well-explained in steps by the cited paper.

It seems, for best results, more than one neural network approach has to be applied. For example, Russell et al [24] achieved their best results using *'features learned via convolutional neural network and classified with an ensemble tree algorithm'*. However, for reasons explained below, we may combine our Neural Network with our ILP approach as a final model.

2.4 Combination Methods Exploration

Holldobler [13], in 1999, proved that for any type of logic programming, there exists a recurrent neural network that can approximate it. Boonserm [14] combined ILP and Neural Networks to classify Thai characters and got a significant improvement. ILP generated 77 rules through which the system could count the numbers of unmatching and matching predicates of the rules. They then used these numbers as inputs in the Neural Network for handwriting classification. However, logic programming cannot describe uncertainty which causes the vulnerability to learn noisy data. In order to solve that issue, Dantsin [6] first introduced probability distribution in logic programming expanding the space from "true or false" to the real unit interval [0,1]. Based on this, De Raedt in 2008 [7] came up with the Probabilistic Inductive Logic Programming (PILP) which improved the noise robustness of ILP.

Evans [9] introduced differentiable components by using functions mapping to represent the rules which performed well when dealing with mislabelled data. It constructs all potential clauses and gives initial values for the clause weightings, then trains the clauses by minimise the loss. However, as the arity of predicates increases, the number of potential clauses will exponentially increase. Evans' approach, therefore, limits the number of predicates to two. Rules with more than two predicates should be divided into sub-rules. This gives a trade-off between the number of rules and the number

of clauses. This wastes a lot of calculation resources and makes it difficult for large datasets.

Rocktäschel [23] came up with another approach to represent the symbols using fixed dimensional vectors and using neural networks to learn the probability. The vectors transformation method provides a new approach of a combination of source code detection with Deep Learning and ILP, once we transfer our code into a vector space by using techniques like a program dependence graph or word embedding[24].

2.5 Summary

Although the traditional methods are precise, speed and flexibility can be compromised and weakened. Another possible issue with these methods is that these don't adapt as well to change compared to more modern approaches. Statistical methods are able to be better adapted, especially when non-parametric. More complex bugs, such as use-after-frees, are computationally infeasible to model formally [29].

ILP is useful because it allows you to see how the model goes about performing classification, as its output is human readable and therefore straightforward to modify for increasing its accuracy. However, accuracy is not always very high and it helps a lot of the time to use one's own prior knowledge. Another issue could be that it is proven to be good for small datasets, but may not scale very well.

Neural Networks, on the other hand, scale very well and can help to increase accuracy, but cannot evidence why this occurs. We could know this more from ILP. Also, the paper we have read uses static analysis to train their network so would this work so well on data without this.

Neural Networks are trained for interpolation and are not so simple to extrapolate beyond the data they have been trained on. However, ILP is better at this because it has to 'explain its workings'.

The combination of more classical methods/ILP with Neural Networks seems like a good approach as it helps to balance out the problems with both models. ILP and Deep Learning actually have similarity, it is worth to explore the potential methods to combine these two to obtain a better performance of vulnerability detection. We plan to build a system that will be able to exploit the benefits of ILP and Neural networks, whilst avoiding its limitations.

3 Evaluating the Viability of our Dataset

Internet provides a large number of labelled source code data which brings out great convenience for automatic vulnerability detection. Currently, we found two appropriate datasets: SATE IV Juliet Test Suite and Draper VDISC Dataset.

3.1 Data Description

The Juliet Test Suite[20] contains 64099 individual test cases in C/C++ with 118 different Common Weakness Enumeration (CWEs) and 28881 examples in Java with 112 different CWEs. In addition, it points out the location and name of weakness for each example. Draper VDISC Dataset[15] includes 1.27 million functions labelled by static analysis for potential vulnerabilities. However, as the label is added by static analysis tools, mislabelled vulnerabilities might exist and without a accurate location.

CWE-114: Process Control on line(s): 134

```
123 void CWE114_Process_Control__w32_char_connect_socket_21_bad()
124 {
125     char * data;
126     char dataBuffer[100] = "";
127     data = dataBuffer;
128     badStatic = 1; /* true */
129     data = badSource(data);
130     {
131         HMODULE hModule;
132         /* POTENTIAL FLAW: If the path to the library is not specified, an attacker may be able to
133          * replace his own file with the intended library */
134         hModule = LoadLibraryA(data);
135         if (hModule != NULL)
136         {
137             FreeLibrary(hModule);
138             printLine("Library loaded and freed successfully");
139         }
140         else
141         {
142             printLine("Unable to load library");
143         }
144     }
145 }
```

Figure 2: A bad example in Juliet Dataset

We aim to detect vulnerability in function-level. Ideally, it will provide a precise positioning for the bug. In this case, the location and name of weakness provided in Juliet Dataset is supportive. The Draper Dataset contains millions of sample functions, some of which are mislabelled. Not only is this a good application for Deep Learning, it also provides a method to test the robustness of our model. Figure 3 describes how we plan to use these datasets.

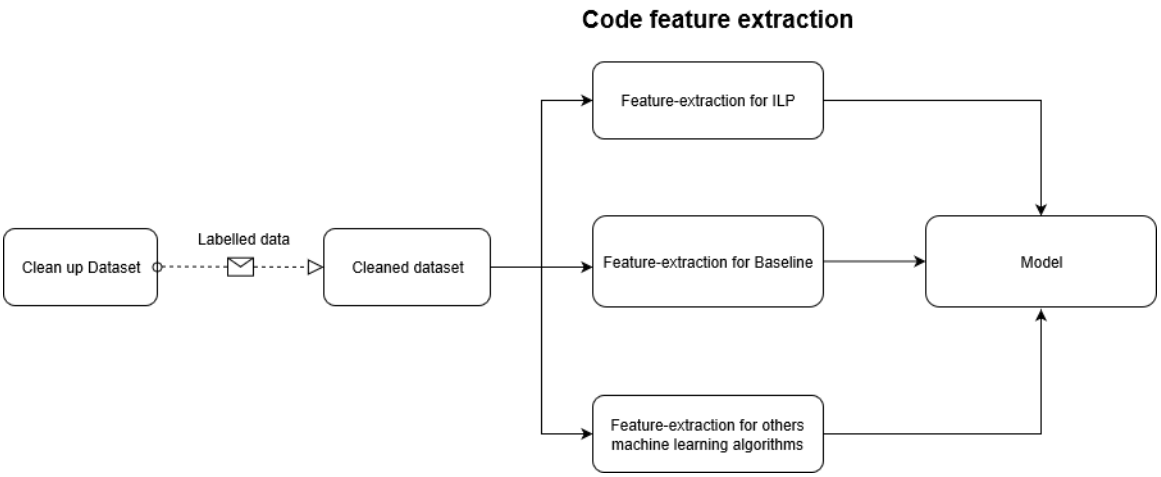

Figure 3: Diagram of the usage of datasets

4 Project Proposal

The following section develops a detailed project plan, informed by the research and analysis above.

The project has been split into a set of distinct tasks. A lead has been allocated for each task, though the specific group members working on each task is expected to change as the project develops:

- Project Setup :: Lead by Daniel Jones.
- Preparation of Juliet Dataset :: Lead by Junfan Huang.
- Development of Baseline Model :: Lead by Sydney Vertigan.
- Preparation of Big Data Set :: Lead by Samantha Wise.
- Minimal ILP System using Code :: Lead by Samantha Wise.
- Code Representations :: Lead by Daniel Jones.
- Evaluation and Test Suite :: Lead by Daniel Jones.
- Development of a Combined Method :: Lead by Junfan Huang.
- Evaluation :: Lead by Sydney Vertigan.

A detailed description of each task and its dependencies follows in the project timeline.

4.1 Estimated Timeline

The estimated time-line for this project is explained in detail here and an overview provided in Figure 5. This timeline was developed from the task dependency graph in Figure 4.1. Work on this project will commence on 17th June 2019, and complete on 6th August 2019. Each date given is the start date with preceding milestone date signalling that the original milestone has been completed.

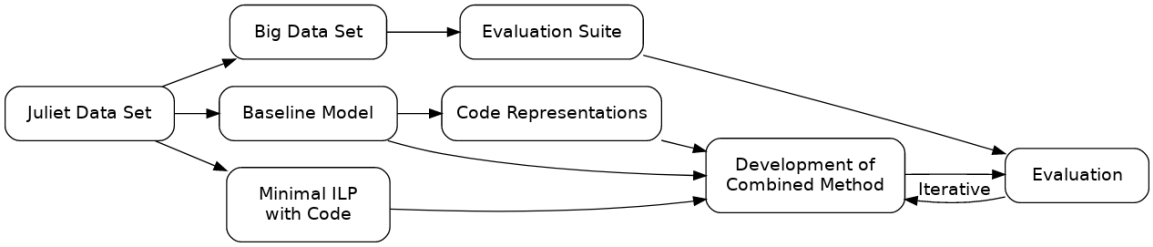

Figure 4: The dependencies between tasks in the project.

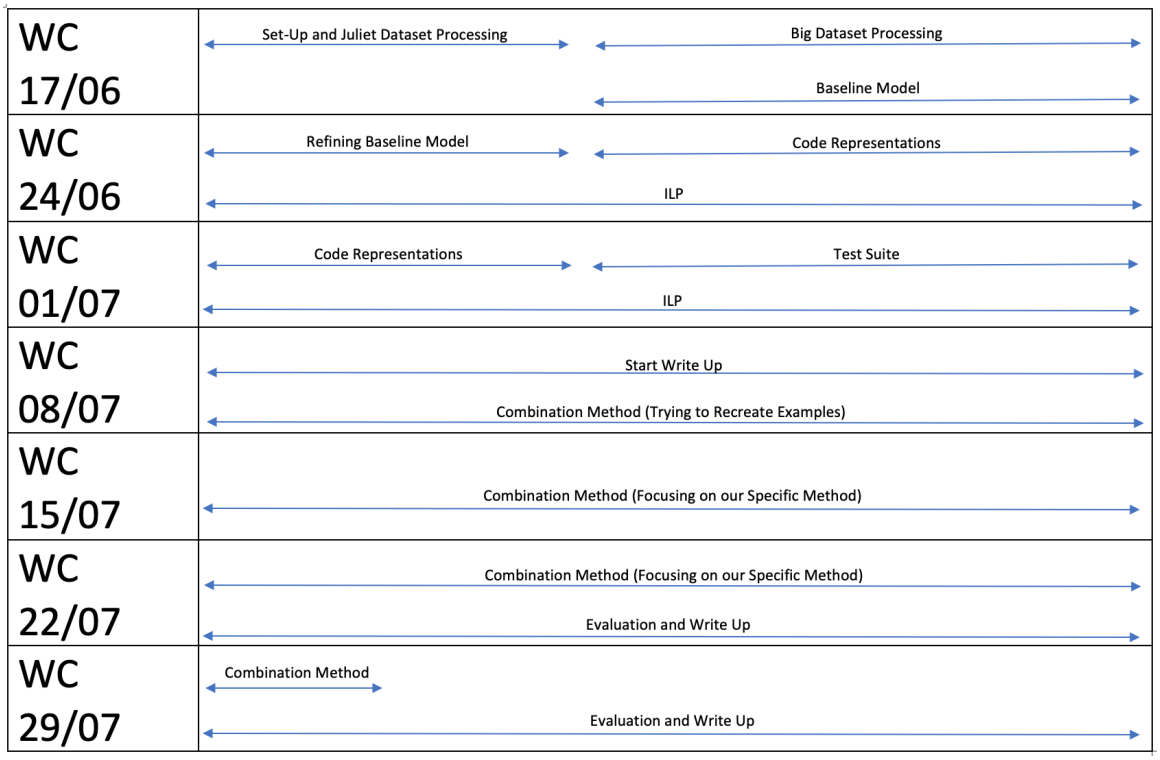

Figure 5: Project Timeline

Milestone 1 (17 June 2019): Data preparation and kick-starting the Baseline Model

Task	Deliverable
1.1 All group members work on processing the 'Juliet' dataset into a suitable format to feed into an ILP and a Neural Network model and set up the project structure in GitHub and Overleaf.	A toy dataset that can be used as an input into an ILP and Neural Network. Group project GitHub and Overleaf document is set up.
1.2 Two members start accumulating and processing our 'big dataset' involving web scraping from GitHub.	A 'Big dataset' that is processed to be fed into an ILP and Neural Network model.
1.3 Two members kickstart our baseline vulnerability detector machine learning model.	A basic machine learning model that can classify buggy code from sound code in our datasets.

Milestone 2 (24 June 2019): Refining Baseline Model, kickstarting ILP and Code Representations

Task	Deliverable
2.1 Two members work on refining the Baseline Model and playing with different code representations (including using graphical methods).	A finalised baseline model.
2.2 Two members work on figuring out how to implement ILP using Prolog and other ILP systems.	A basic understanding on how to use Prolog and other ILP systems through trivial examples.

Milestone 3 (01 July 2019): Finalising Code Representations and ILP, kickstarting Test Suite

Task	Deliverable
3.1 One member continuing to work on code representations.	A final decision on the optimal code representation.
3.2 One member working on Test Suite.	A final decision on the test suite format.
3.3 Two members continuing to work on how to implement ILP using Prolog.	A deep understanding on how to use Prolog and other ILP systems through examples in [18].

Milestone 4 (08 July 2019): Start Write Up and Combination Model

Task	Deliverable
4.1 Two members start writing the report.	A LaTeXproject report on the progress thus far.
4.2 Two members work on building the Combination Model and testing it on the examples found in [9].	A combination model that successfully works on the examples in [9].

Milestone 5 (15 July 2019): Building Combination Model

Task	Deliverable
5.1 All group members work on building our final combination model.	A sketch/skeleton of our Combination Model that works with our toy dataset.

Milestone 6 (22 July 2019): Finalising Combination Model and starting Evaluation

Task	Deliverable
6.1 Two group members working on the final model.	Added extensions to the Combination Model that can handle larger amounts of data.
6.2 Two members working on Model Evaluation and Write Up.	An evaluation table for the models tested thus far.

Milestone 7 (29 July 2019): Wrapping up

Task	Deliverable
7.1 Two members working on finalising the Combination Model.	A finalised combination model.
7.2 Two members working on Evaluation and Write Up.	A finalised evaluation table for the baseline model and our combination model. A first draft LaTeXproject report.

4.2 Risk Analysis

The risks proposed here are ranked on a likelihood 1 to 5 with 1 being the lowest and 5 the highest probability. The severity of the risk is also marked in a comparable manner.

Risk	Likelihood	Severity	Prevention
'Juliet' dataset more challenging to process than perceived.	3	2	Read more documentation on the dataset and follow implementations.
Failure to format data appropriately for baseline model in time.	2	2	Create a substantial, simpler baseline or read more documentation on how to format data for neural networks.
Spending too much time researching and trying different code representations.	1	1	Stop trying them after time allotted.
Failure to format data to be suitable for ILP.	3	3	Read more documentation on how data needs to be formatted for ILP.
Failure to set up ILP model.	3	3	Plan how we will set up our model using information found in papers.
Failure to feed the output of the first component into the input of the second component in our combination model.	3	4	Process outputs into the correct form to be inputted into the second component.
No prior experience of creating a similar model and scarce publicly available resources, not knowing what steps to take in times of crisis.	5	5	Talk to experts from CS department [21] and Qinetiq [26]. Also try to find as many more resources as we can.
Difficulty in developing unified model interface with test suite for evaluation purposes.	2	4	Developing the test suite early so we create our models to the test suite's spec.
Combination model not ready as scheduled.	3	4	Making our model simpler than we first wanted.
Combination model fails.	4	4	Work out why and therefore why this may not be the best method.
Not being able to meet word count requirements.	2	1	Working out the least important information to cut.

5 Conclusion

From our review of the current literature, we have determined that an approach combining machine learning with inductive logic programming, informed by prior work on constraint-based program analysis, is worth pursuing.

Our conversations with experts in industry [26] and academia [21] further support this. To ensure the project is viable in the given timescale, we have located sufficient data sets in the public domain. Finally, we have assimilated the aforementioned work to develop a detailed project plan, including risk analysis.

References

[1] CVE Details: Mozilla Firefox, https://www.cvedetails.com/product/3264/mozilla-firefox.html (accessed 2nd May 2019).

[2] Exploiting 64-bit Linux like a boss, https://scarybeastsecurity.blogspot.com/2013/02/exploiting-64-bit-linux-like-boss.html (accessed 2rd May 2019).

[3] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6):23, 2010.

[4] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Acm sigplan notices*, volume 41, pages 158–168. ACM, 2006.

[5] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *ACM SIGPLAN Notices*, volume 42, pages 480–491. ACM, 2007.

[6] Eugene Dantsin. Probabilistic logic programs and their semantics. In *Logic Programming*, pages 152–164. Springer, 1992.

[7] Luc De Raedt and Kristian Kersting. Probabilistic inductive logic programming. In *Probabilistic Inductive Logic Programming*, pages 1–27. Springer, 2008.

[8] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: enforcing alias analysis for weakly typed languages. In *ACM SIGPLAN Notices*, volume 41, pages 144–157. ACM, 2006.

[9] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61:1–64, 2018.

[10] Josselin Feist. *Finding the needle in the heap: combining binary analysis techniques to trigger use-after-free.* PhD thesis, Grenoble Alpes, 2017.

[11] Josselin Feist, Laurent Mounier, Sébastien Bardin, Robin David, and Marie-Laure Potet. Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, page 2. ACM, 2016.

[12] David L Heine and Monica S Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *ACM SIGPLAN Notices*, volume 38, pages 168–181. ACM, 2003.

[13] Steffen Hölldobler, Yvonne Kalinke, and Hans-Peter Störr. Approximating the semantics of logic programs by recurrent neural networks. *Applied Intelligence*, 11(1):45–58, 1999.

[14] Boonserm Kijsirikul, Sukree Sinthupinyo, and Apinya Supanwansa. Thai printed character recognition by combining inductive logic programming with backpropagation neural network. In *IEEE. APCCAS 1998. 1998 IEEE Asia-Pacific Conference on Circuits and Systems. Microelectronics and Integrating Systems. Proceedings (Cat. No. 98EX242)*, pages 539–542. IEEE, 1998.

[15] Louis Kim and Rebecca Russell. Draper vdisc dataset - vulnerability detection in source code. https://osf.io/d45bw/, 2018. Accessed: 2rd May 2019.

[16] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[17] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[18] Steve Moyle and John Heasman. Machine learning to detect intrusion strategies. In *Knowledge-Based Intelligent Information and Engineering Systems, 7th International Conference, KES 2003, Oxford, UK, September 3-5, 2003, Proceedings, Part I*, pages 371–378, 2003.

[19] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[20] NIST. Juliet test suite v1.3. https://samate.nist.gov/SRD/testsuite.php, 2017. Accessed: 2rd May 2019.

[21] Oliver Ray. Word of mouth, April 2019.

[22] Oliver Ray, Samuel Hicks, and Steve Moyle. Using ILP to analyse ransomware attacks. In *Proceedings of the 26th International Conference on Inductive Logic Programming (Short papers), London, UK, 2016.*, pages 54–59, 2016.

[23] Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. In *Advances in Neural Information Processing Systems*, pages 3788–3800, 2017.

[24] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762. IEEE, 2018.

[25] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014.

[26] Claire J Taylor. QinetiQ, word of mouth, Mar 2019.

[27] Yichen Xie and Alex Aiken. Context-and path-sensitive memory leak detection. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 115–125. ACM, 2005.

[28] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. *ACM SIGSOFT Software Engineering Notes*, 28(5):327–336, 2003.

[29] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 42–54. ACM, 2017.