Australian Government
Department of Defence
Science and Technology

A Review of Machine Learning in Software Vulnerability Research

Tamas Abraham[1] *and Olivier de Vel*[2]

[1] **Cyber and Electronic Warfare Division**
[2] **Principal Scientist**
Defence Science and Technology Group

DST-Group–GD–0979

ABSTRACT

Searching for and identifying vulnerabilities in computer software has a long and rich history, be that for preventative or malicious purposes. In this paper, we investigate the use of Machine Learning (ML) techniques in Software Vulnerability Research (SVR), discussing previous and current efforts to illustrate how ML is utilised by academia and industry in this area. We find that the primary focus is not only on discovering new approaches, but on helping SVR practitioners by simplifying and automating their processes. Considering the variety of applications already in evidence, we believe ML will continue to provide assistance to SVR in the future as new areas of use are explored and improved algorithms to enhance existing functionality become available.

Produced by

Cyber and Electronic Warfare Division
PO Box 1500
Edinburgh, South Australia 5111, Australia

Telephone: 1300 333 362

AR-017-005
October, 2017

APPROVED FOR PUBLIC RELEASE

A Review of Machine Learning in Software Vulnerability Research

Executive Summary

Computer software, like any other product, may contain faults. Some may be benign, others can have serious implications to the operation and security of the systems they are deployed on. Flaws in software that can be exploited for malicious purposes by an attacker belong to the the class of *software vulnerabilities*. Research into computer software vulnerabilities has a long and rich history, be that for preventative or malicious purposes, and has increased relevance to Defence as it tries to maintain a high level of information assurance of its systems.

In this report we review the use of Machine Learning (ML) techniques in Software Vulnerability Research (SVR), and discuss previous and current efforts to illustrate how ML is utilised by the various researchers in this area. It is intended as an accessible introduction to those not yet fully immersed in both fields of study and to also encourage the identification of and further research into the problems fitting the specific goals of the reader. For this reason, the description of individual articles is kept short to highlight only the techniques employed and their intended purposes. A short introduction to basic concepts and techniques of both Software Vulnerability Research and Machine Learning is also provided.

By conducting this review, we find that the primary benefit of ML in SVR is not only in the discovery of new approaches, but also in helping SVR practitioners by simplifying and automating their processes. Considering the variety of applications already in evidence, we conclude that ML will continue to provide assistance to SVR in the future as new areas of use are explored and improved algorithms to enhance existing functionality become available.

This page is intentionally blank

Contents

Figures

This page is intentionally blank

1. Introduction

Creating computer software is a non-trivial, complex process that can often yield code that contains some flaws and weaknesses. Verification of large code bases can be difficult, or is sometimes ignored due to costs, resulting in operational systems that can break down, or be manipulated due to the unexpected and undesirable behaviours they exhibit. Although the severity of software security errors is variable, some can be sufficiently serious to be able to be exploited to cause critical harm to users by lost productivity, loss of intellectual property or even physical damage. Dowd et al. [30] terms the subclass of software "bugs" that can be exploited for malicious purposes as vulnerabilities, although practical exploitation of a vulnerability may not always be possible in a particular context or be suitable for the goals of an attacker. Efforts to counter the issues caused by vulnerable software has created studies in Software Vulnerability Research (SVR).

Research into vulnerabilities affecting computer systems is not only limited to software. Our focus in this paper, however, is on software-based vulnerabilities, and we do not consider hardware or system architectures. Software itself can be analysed as source code or binary, providing multiple paths to finding vulnerabilities. Research processes into software vulnerabilities can be routinely differentiated into pursuits for discovery, analysis and exploitation, with mitigation included as a preventative activity [73]. Each stage explores a different facet of dealing with vulnerabilities, often requiring long and laborious input from practitioners such as code auditors. Automation plays an important part in many SVR activities, nevertheless currently it is via human interpretation that most vulnerability discoveries are made. Increasingly, Machine Learning (ML) techniques are incorporated into SVR processes to further decrease the need for manual interaction. When applied successfully, ML algorithms can guide users towards most likely solutions and potentially uncover previously unknown vulnerabilities. The goal of this paper is to catalogue efforts within SVR that utilise machine learning in order to highlight the state of current undertakings and identify possible areas where further contributions could be made in the future.

2. Background

The focus of this paper is to draw attention to the use of machine learning in software vulnerability research. To enable discussion, we introduce both fields separately with some general details. Further particulars are given as necessary in subsequent sections when reviewing individual publications. What follows here is a simple overview of the two areas of study covered without, at this stage, highlighting any connections between them.

2.1. Software Vulnerability Research

Classifying software vulnerabilities is not a simple task. The category to which a particular vulnerability belongs is usually revealed during the analysis of the software error. Sometimes a type can be anticipated during the vulnerability discovery process since some techniques implicitly target a limited range of vulnerability types. Tools such as *scanners* look for bad constructs such as obsolete library functions and other security-related errors in code. *Fuzzing*

is the process of supplying software executables with unusual input to cause unexpected behaviours like a crash or a memory leak, which can then be investigated in the corresponding source code. Comprehensive manual *code reviews* can also be used to uncover errors but they can be an expensive alternative to other vulnerability discovery approaches. *Formal verification* provides mathematical proofs of correctness, and if unsuccessful, can indicate problems. It is, however, often limited to small code segments or algorithms due to complexity and costs. *Symbolic execution*, a technique to analyse the traversal of variable values (inputs) through program branches is another discovery method although it can suffer from scaling issues like path explosion.

Some errors in source code may be easy to group although certain vulnerabilities only present themselves in combination with other factors such as the platform they are deployed on, and can therefore be difficult to initially categorise. The diversity of computer architectures, operating systems, computer languages and the existence of language-specific bugs further complicates analysis. We use the taxonomy given in the book by Dowd et al. [30] as our guide to define software vulnerability types. Language-specific issues include:

- memory corruption such as buffer overflows (stack, off-by-one, heap, global and static data)
- arithmetic boundary conditions such as numeric over- and underflows
- type conversion errors (signed/unsigned, sign extension, truncation, comparisons)
- operator misuse (sizeof(), shift, modulus, division)
- pointer arithmetic bugs
- other errors (evaluation order logic, structure padding, precedence, macros/preprocessors, typos)

Other vulnerabilities are more complex. Problem categories include those that are present in practical applications such as in relation to operating systems or application platforms, even though the underlying issues may be caused by simpler errors such as memory corruptions or pointer bugs:

- string and meta-character vulnerabilities
- OS-specific vulnerabilities (privilege problems, file permission issues, race conditions, processes, IPC, threads, environments and signalling issues)
- platform vulnerabilities (SQL injection, cross-site scripting (XSS), cross-site request forgery (CSRF), file inclusion and access, shell invocation, configuration, access control and authorisation flaws)

An alternate way of categorising vulnerabilities is from the attack perspective. For example, the Open Web Application Security Project [3] provides a list of attack types and regularly

compiles a list of top contemporary vulnerabilities [1], not all of which are related to source code or binaries. Of those that are, injection attacks (code, SQL, HTML script, remote file and shell) and control flow hijacking such as overflows (buffer, integer, string format) and heap spray are similar to those we listed above. Other authors like Bletsch provide similar categorisations that discuss how vulnerabilities can be exploited for example via code re-use (return-oriented programming (ROP), return-into-libc (RILC), jump-oriented programming (JOP)) [80].

As vulnerabilities in software get discovered, they are often shared with the wider community. The Common Vulnerabilities and Exposures (CVE) is a "dictionary of publicly known information security vulnerabilities and exposures" [4], currently maintained by the MITRE organisation. Vulnerabilities are often attached a score to describe their severity, with the Common Vulnerability Scoring System (CVSS) [40] being one of the most utilised. Services that provide access to CVEs, scores and other related information include databases such as the Open Source Vulnerability Database (OSVDB) and National Vulnerability Database U.S. (NVD), and APIs that allow real-time updates on newly discovered vulnerabilities such as VulnDB and vFeed. Statistics are often compiled annually on popular exploits based on the number of vulnerabilities found in different categories, although there may not be a correlation between the frequency and severity of attacks attributed to each type. For example, Price and Kouns [46] lists cross-site scripting, SQL injection, cross site request forgery, file inclusion, denial of service and overflow attacks as the most frequently observed abuses according to the OSVDB in 2014.

Efforts to mitigate the impact of software vulnerabilities have also produced various strategies and solutions. Although full error prevention may not be achievable, there are expectations on software producers to minimise the possibility of their product containing vulnerabilities. These include comprehensive testing and fixing errors during the development cycle, providing fixes after the release of software, and writing code in what are considered secure programming languages. A list of vulnerability discovery techniques that can be utilised prior to releasing software as a preventative measure, such as software testing, fuzzing and program verification (theorem proving, abstract interpretation, model checking) are listed in Vanegue [82]. Modelling code execution using graphs such as abstract syntax trees (AST), control flow graphs (CFG), program dependency graphs (PDG) and code property graphs (CPG) can be helpful in identifying issues during development. Additional mitigation techniques for use after the software has been released has also been provided by OS and hardware manufacturers. Data Execution Prevention (DEP)/No eXecute (NX), Address Space Layout Randomisation (ASLR), instruction-set randomisation (ISR), run-time checking (canaries, LibSafe), program shepherding, Control Flow Integrity (CFI), Data Flow Integrity (DFI) and Control Flow Locking are some of the technologies currently in use, see [80]. Software tools to perform various discovery tasks are readily available, both commercial and open source [87].

Program analysis is central to the problem of analysing software behaviour. Theoretically, the identification of a software bug is undecidable, i.e. it is not possible to write a program to represent and compute all possible executions of another program in the general case [67]. In practice,some program behaviours may be ignored as not being relevant to the current analysis. However, this can result in either under approximation—the exclusion of possible valid behaviours that may contain bugs, and over approximation—the inclusion of possible

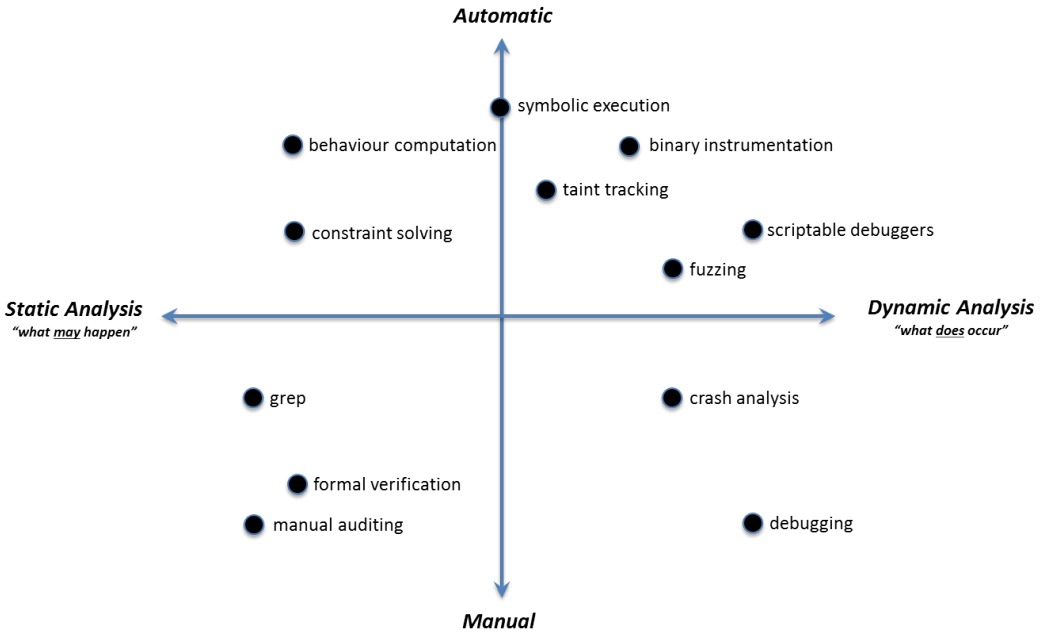

Figure 1: Program analysis for vulnerability research, adapted from Cohen [27]

but invalid behaviours—increasing complexity and resource requirements. Figure 1 organises a number of program analysis techniques into a graph highlighting the style and automation levels of individual methods. Static analysis, which is undertaken without executing the program, can provide good code coverage and reason about all possible executions but cannot analyse the executable environment such as the operating system and hardware. Dynamic analysis, on the other hand, is undertaken while the program is executing or by instrumenting the program to analyse behaviour. However, it can only reason about the observed execution paths and not all possible program paths.

2.2. Machine learning

In this section we provide a short overview of machine learning concepts, and highlight some of the relevant techniques that we encounter in the publications discussed later in this paper. Many different taxonomies exist for categorising ML techniques. For convenience, we use the book by Barber [13] as the reference source for this section unless specific references are given.

The field of machine learning concerns itself with the automated exploration of data, resulting in descriptive models that can be used for prediction. Generally, two main styles of learning are recognised: *supervised* learning builds its models from labelled data sources and focuses on the accuracy of prediction, while *unsupervised* learning concentrates on providing compact descriptions from unlabelled data. Aside these two main styles, several variations can be observed. *Anomaly detection* looks for outliers in the data that deviate from built models. *Online* learning is able to continually update models as new data becomes available. *Active* learning can build better models by requiring more data from the regions of the environment which the current model does not sufficiently describe. *Reinforcement* learning is able to interact with the environment in a trial and error fashion to create models that are optimised

according to some form of reward. Finally, *semi-supervised* learning utilises both labelled and unlabelled data, using one type of data to improve on models that can be created from the other type of data alone.

A large range of algorithms have been proposed and developed over the years within the above learning styles. Supervised learning mostly uses one of many types of *classifiers* that predict the group (class) a data point belongs to. This is a form of discrete learning as the output is limited to a set of values. When the result is required to be within a value range (i.e. it is a continuous variable), *regression* is the technique used. One of the simplest classification algorithms is the K-nearest neighbour (kNN) algorithm that determines the class label of a data point by looking at the labels of its K nearest neighbours and selecting the most frequent one, an example for instance-based learning where the decision about the class is made using examples in the data set rather than a model built from it. The Naïve Bayes classifier is a probabilistic algorithm that assumes conditional independence between the variables that describe the data to simplify the building of a generative model. The class label of a point is estimated by the probabilities of the data and their conditional probabilities given the different labels using Bayes rule. Other classification techniques fit linear models to the data and determine class membership based on the position of a data point respective to a decision boundary calculated from known examples. Logistic regression is a classification algorithm that uses a maximum likelihood function to approximate the probability of a data point belonging to a class. Linear Support Vector Machines (SVM) produce a hyperplane to separate the classes in such a way that the distance between the nearest point on each side of the plane is maximised.

Decision tree classifiers model sequential decision processes as special graphs, with each node serving as a feature test, so that different values of a given feature are laid out along separate branches. The leaves of the tree determine class membership. From any particular example data set there can be many different trees built, and often a combination of multiple trees from a single data set are used to build better models. Random forests are such an ensemble of decision trees aiming to deliver improved predictions over models created as a single tree. Graphs are also prominent in representing the various forms of neural networks (NN) [14] used as classifiers. Neural networks are organised in layers of nodes, containing an input layer and an output layer, with hidden layers between. Each node corresponds to a function that maps its input values into a single output value using weights assigned to the connecting edges of the node. Popular variants of NN classifiers include the multilayer perceptron (MLP) feedforward neural network, convolutional neural networks and long short-term memory recurrent neural networks. Neural networks are also central to deep learning, a machine learning paradigm that also concerns itself with the learning of the representations of the data.

Unsupervised learning is often associated with cluster analysis, the organisation of data into groups based on some definition of similarity. These can include statistical approaches based on data distribution, such as using expectation maximisation (EM) to build Gaussian Mixture Models (GMM). Algorithms exploiting data connectivity are examples of hierarchical clustering, either built bottom-up, i.e. starting with each data point as a cluster followed by merge operations, or top-down, starting with a single cluster and splitting according to some strategy. Centroid-based clustering identifies clusters by determining a selected number of cluster centres and assigning each data point to the closest one. Density-based clustering, on the other

hand, finds clusters of points that are close to each other based on thresholds of some distance measure, and can leave data unassigned as noise if they do not satisfy these requirements.

Associations [5] are rules inspired by market data analysis. They represent *if* ... *then* constructs that describe strong patterns in the data that occur with some minimum desired frequency. Finding frequent itemsets, or feature values that occur together more often than others, can reveal trends in the data. Sequential pattern mining is a learning activity that has the same goal but analyses data over time, exploiting the temporal ordering of data points to build models. Genetic algorithms are also rule discovery algorithms that mimic natural selection by applying crossover and mutation operators to an initial set of data and evaluating the fitness of subsequent generations until some termination conditions are met.

Learning is usually preceded by pre-processing activities and is followed by tasks such as model evaluation. Some of the pre-processing includes feature extraction, and the handling of noise and errors in the data. Feature selection and dimension reduction aim to identify features relevant to the learning task and to reduce complexity in order to improve the model generated by the learning algorithm. Both supervised and unsupervised learners can benefit from this process. Some important examples include Principal Component Analysis (PCA), Linear Discriminant Analysis (LDA), Non-negative Matrix Factorisation (NMF) and Singular Value Decomposition (SVD). Sampling to reduce data size and balancing input data can improve the efficiency and effectiveness of algorithms. Using ensemble methods such as bagging and boosting can improve predictive performance over single learning algorithms. For evaluation purposes, performance assessment methods are available to appraise the effectiveness of an algorithm. In binary classification, the relationship between the predicted conditions and actual conditions of data points can be described using several concepts. A true positive (TP), or hit, is a correctly predicted positive instance. A false positive (FP) is a negative instance incorrectly predicted as positive. Similar definitions exist for true negatives and false negatives. The true positive rate (TPR), or recall of an algorithm is the ratio of true positives over all positive instances. The precision is the ratio of TP over the sum of TP and FP, indicative of the number of mistakes made when predicting positive instances. The accuracy of the algorithm is the ratio of correctly identified instances (TP plus TN) over all data points. Many model evaluation methods exist, for example ROC analysis [31]. Combined, they can not only provide an assessment of an algorithm but determine other learning strategies such as different feature selection methods or parameter optimisations.

Machine learning has been applied in numerous fields of study. One that can be relevant to examining computer programs is Natural Language Processing (NLP), a field that concerns itself with tasks such as language modelling, parsing and speech recognition. Some notable techniques available for document classification include Latent Dirichlet Allocation (LDA) and Latent Semantic Indexing (LSI), both modelling text as a collection of topics.

3. Learning in Software Vulnerability Research

Machine learning can offer many benefits to a complex area of study such as software vulnerability research. It can be used to model the syntax and semantics of the code and infer code patterns to analyse large code bases, assist in code auditing and code understanding,

whilst achieving tolerable false positive rates. As SVR processes increase in complexity, so does the need for the level of automation available to SVR practitioners. As a result, novel methods of analysing software are being proposed for both discovery and prevention purposes. Some of these are specialised. Others use readily available technologies from other scientific disciplines, including statistics, machine learning and data mining. In the following sections, we discuss existing work utilising primarily machine learning, organising them according to a loose grouping based on the similarity of their content.

We first point to a small selection of recent papers that consider addressing the software vulnerability problem at a more general level. For example, Avgerinos et al. [11] acknowledge the presence of software bugs in widely used large software projects such as the Firefox browser and the Linux kernel, some known but others potentially still undiscovered. With so many bugs found in critical software, the authors pose the questions: "Which of these bugs should we try to fix first?" "Can we determine which ones are exploitable?" Jimenez et al. [41] may indicate a possible way ahead. They analyse past known vulnerabilities (in this case, for Android) and establish a categorisation that lists issues leading to software vulnerabilities, the characteristics of the locations in the code where vulnerabilities reside, the complexity of these locations and the complexity to fix the vulnerabilities. Creating public datasets such as *VDiscovery* [34] which collect the results of test cases and are made available to facilitate further research is another promising initiative, as is the idea of crowd-sourcing "bug hunting", a model for which is described by Zhao et al. [93].

3.1. Source code analysis

As highlighted earlier, reducing the amount of manual tasks on human practitioners of SVR is a primary objective of many proposed approaches. Examples of automation include *Parfait* [26], a framework for finding bugs in `C`/`C++` code. *Parfait* was designed with multiple layers per bug type in an ensemble of program analyses for speed and scalability. The philosophy behind the solution is to employ simpler analyses for predicting some types of bugs, then moving to more computationally expensive ones to achieve best coverage and precision. The same philosophy is shared by another platform, *Mélange* [74], also analysing `C` and `C++` code. *Mélange* performs data and control flow analysis and generates bug reports to explain the bugs found to help with the necessary fixes. Analyses are executed in stages, both locally and globally, with the latter used on-demand to validate the outcomes of local analyses. Another example is the *SZZ* algorithm [88], which was developed to automatically identify code commits that induce fixes, and can be used by researchers to validate software metrics or models for predicting faulty components, an important activity in bug prevention.

However, these approaches are examples where the move towards automation does not necessarily rely on machine learning technologies. The following categories detail cases where they do.

3.1.1. Coding practices

One of the early uses of machine learning for code analysis is *PR-Miner* [50], the application of data mining techniques to build a set of programming rules for source code. It generates frequent itemsets from large code bases such as Linux, PostgreSQL and the Apache HTTP

Server, to automatically produce implicit programming rules which can then be checked for violations using another algorithm. The results are ranked and provided to an analyst in order of assumed severity to confirm whether they constitute actual bugs. The process is fast and the authors argue that it is able to identify violations that are more complex (e.g. contain more than two rule components) than those found by comparable tools which use user-defined templates. *AutoISES* [77] is a similar tool that detects vulnerabilities by inferring security specifications from source code rather than requiring them to be manually provided. The inference of specifications is still guided by concepts relating to secure coding practices but rules are now extracted based on evidence observed in the code and violations are offered for manual verification. The Linux kernel and Xen were used as test cases and for 84 extracted rules, 8 new vulnerabilities were found.

Assisting developers to correctly use Application Programming Interface (API) methods has been a focus of some papers, often inspired by the lack of sufficient available documentation. The *UP-Miner* tool [83] employs several data mining methods, such as clustering and frequent closed sequence mining to create frequent API usage patterns. It also contains new metrics to optimise the succinctness and coverage of the resulting patterns, which are then presented to users as probabilistic graphs for examination and understanding. Tests on a large Microsoft code-base in collaboration with Microsoft developers confirmed the utility of the approach. Another interesting contribution is detailed by Nguyen et al. [59]. They investigate API preconditions, conditions that need to be satisfied by the parameters of an API method prior to its call. They developed a system that finds the client methods invoking APIs, computes a control dependence relation from each call site and then mines the conditions used to reach those call sites to finally infer the preconditions for each API. A large-scale evaluation of the Java Development Kit using SourceForge and Apache projects identified a number of preconditions missing from written specifications. Furthermore, the results can be used to identify coding errors where preconditions in source code are not met. The paper also has a very good collection of references on earlier API mining literature.

VCCFinder [62] is a tool that combines knowledge about vulnerabilities in code with metadata about commits made to repositories to identify potentially vulnerable software code commits. A series of features generated for each commit from both source types are matched against the features of known cases of "vulnerability-contributing commits" (VCCs), identified from commit data for CVEs, to determine if a new commit is a likely source of a vulnerability. A two-class SVM is built for this purpose. Tests on 66 GitHub projects showed a greatly reduced false positive rate (FPR) compared to existing tools while maintaining a similar true positive rate. Whilst successfully identifying a VCC can greatly reduce the amount of code to inspect for security flaws, significant expertise and manual effort to audit them will still be required by practitioners.

3.1.2. Clone detection

Duplicate code can not only make a software project more difficult to maintain due to code bloat, but also to resolve flaws when they are scattered across a large code base as the result of copy-paste programming. Detecting clones has therefore attracted a lot of attention in the literature, including from the vulnerability perspective. A recent survey by Roy et al. [70] provides concepts and a qualitative comparison and evaluation of clone detection techniques

and tools, along with a taxonomy.

Much of the research associated with clone detection focuses on determining code fragment similarities to locate copied code. Udagawa [81] presents a code-structure based approach that extracts lexical data from Java source code fragments using a parser and applies a similarity measure defined by the ratio of the number of fully matching sequences of tokens to the number of sequential partially matching statements. Lazar and Banias [47] use a sub-tree similarity measure in multiple file sets of C programs using abstract syntax trees in another example of a structure-based approach. These are often preferred to text or token-based approaches as they are robust to code and variable name changes, although they are often not well suited to large programs due to scalability issues.

Clone detection algorithms have also been used for bug-fixing purposes in conjunction with machine learning, for example by Steild and Göde [76]. Their idea is to extract features from clones and after training a classification model, identify whether similar clones have incomplete bug fixes. Clone detection is essentially token-based, that is, statements are tokenised rather than represented as trees, and this is reflected in the type of features that are extracted from code fragments: global context features are complemented with local lexical ones that allow for minor inconsistencies within clones. The authors investigated multiple classification techniques and found that decision trees were the most promising and offered an easy to understand representation for users. Test results show that even at a high ratio of false positive to true positives (approximately four in five finds were false), their approach represents a marked improvement when compared to manual analysis.

Taking the concept of clone detection further, the $\mathcal{C}3$ system [45] investigates code changes in source code repositories. The idea is to simplify the application of bug fixes by automatically locating similar code changes without the need for interaction with a user and/or existing code change patterns, to be passed on to other tools for application. Two similarity measures are proposed, one based on a traditional diff-based representation, and another on abstract syntax trees, which are used on extracted code changes to generate the similarity matrix. Clustering is then used to detect groups of similar changes (rather than identical ones as in the case of clones). Results on large code depositories show that they can be delivered in a time efficient manner with a success rate similar to that achieved by expert manual extraction.

3.1.3. Error detection

Code flaws, whether exploitable or not, can be difficult to identify, particularly in large software projects. In the case of source code, some of the more frequently used techniques to find bugs include using templates to guide the search for known vulnerabilities; examining the contents of source code files and comparing with known vulnerable ones; and analysing the structure of code to diagnose potential mistakes. Often, a combination of methods is used to reinforce results, with the theme of these activities being either the identification of code that is different from normal, or the recognition of code that is similar to known bad.

Employing a priori knowledge can be very effective when used to target specific vulnerability patterns. Neglected conditions are the topic of Chang et al. [25]. Their approach requires the user to specify constraints for learning conditional rules from code which are used to

discover violations indicating neglected conditions. Extracted rules are represented as graphs and a maximal frequent subgraph mining algorithm followed by a graph matching algorithm classifies rule violations. A user may be consulted to evaluate the usefulness of the extracted rules and adjust them before matching takes place. Various open source software projects were tested with this approach and revealed previously unknown violations. *Alattin* [79] is another proposal to identify neglected conditions, using a modified frequent pattern mining algorithm called *ImMiner*. It introduces the concept of alternative patterns as the disjunction of two rules affecting the same API calls in the program. When both individual patterns are frequent, the alternative pattern is called balanced; when only one is, the alternative pattern is called imbalanced, and can be used for both program comprehension and defect detection. A variation on the frequent itemset mining algorithm has been developed to search for balanced and imbalanced alternative patterns, and applied to detect neglected conditions around API calls. The approach tested well when compared to similar solutions and is available as an Eclipse plugin.

Patterns that describe normal behaviour rather than violations are also used for bug detection. Gruska et al. [35] parse a large corpus of software projects to extract frequent temporal properties representing the flow of values between functions calls. An anomaly detection algorithm is then used to detect violations of the learnt patterns using association rule lift measures to rank and filter them for user evaluation. In real-world tests, 25% of the top-ranked violations were found to be issues, either actual defects or weaknesses in code design.

Treating source code as a document collection and building models to describe them is another approach to develop an understanding of software projects. Lukins et al. [52] use Latent Dirichlet Allocation to generate topic models on string literals, comments and identifiers, then evaluate them with manually crafted bug description queries in several case studies. They find that their technique compares favourably to competing approaches and that it scales well when used on large source code bases. Hovsepyan et al. [39] also look at individual source code files and convert them into feature vectors by removing comments, string and numerical values, and tokenising the remaining code elements (keywords, variable and function names) as feature words for supervised learning. Labels are assigned to each file prior to running a support vector machine algorithm to train a model which is used to predict vulnerable feature vectors extracted from test files. The technique is able to identify most vulnerabilities tested against with a low false discovery rate, and is intended to be complementary to existing vulnerability discovery solutions based on software metrics. Pang et al. [60] expand on this work by including n-grams into the feature vectors generated. Sequences of up to five tokens are considered instead of just single words, and to avoid the resulting feature explosion, a statistical feature selection algorithm is used to provide a ranking. The top 20% of features are then used with an SVM algorithm. Tests on four Android Java projects achieved slightly better classification results than earlier attempts. Scandariato et al. [72] use single words, including comments and string values, in their tokenisation of source code files into feature vectors, but introduce discretisation to bin the various feature counts in order to improve the models generated by their machine learning algorithms. In addition to SVM, they test with decision trees, k-nearest neighbour, naïve Bayes and random forest, with the latter two algorithms performing best in their experiments. The paper also contains a review of previous comparable efforts, including those relying on software metrics to determine if a file contains a software vulnerability. The preference of these metrics over text mining solutions is the topic

of Tang et al. [78]. They argue that while many machine learning based solutions may show improved results, they are not sufficiently large enough to justify the added cost to apply these models in preference to software metrics based models.

Numerous researchers look at solutions that take advantage of the structural properties of programs. Machine learning has been used in Kremenek et al. [44] to investigate the combination of source code program analysis and probabilistic graph models to automatically infer program specifications directly from programs. Inference is undertaken by the creation of an annotation factor graph which can be used to rank possible errors by their probabilities before examining any inferred specifications. Tests on open source code bases showed high rates of true positives for the case of memory allocation/deallocation specifications. Peng et al. [61] explore the possibility of deep learning for program analysis. They argue that natural language token-based granularity can yield sparse data and instead encode abstract syntax tree nodes into vector representations that codify a node in the AST as a single neural layer. They are then used as input into a Convolutional Neural Network for deep supervised learning to classify programs. Quantitative evaluation of the vector representation is done using k-means clustering to show that similar nodes can be successfully grouped together, while a qualitative evaluation of classification tasks reveals a slightly superior outcome with deep learning compared to baseline classifiers such as logistic regression and support vector machines.

Utilising program structures specifically for vulnerability discovery has been the focus of Yamaguchi and his co-authors. Latent semantic analysis [90] of abstract syntax tree nodes is an early attempt to offer assistance to a security analyst during auditing of a source code base. After the extraction of API and syntax nodes, they are embedded into vector space and structural patterns are identified using LSA to generate topics. These can then be compared against the characteristics identified for known vulnerabilities. Tests on several open-source projects helped to uncover new zero-day vulnerabilities. Missing checks are the target of another Yamaguchi et al. paper detailing *Chucky* [92], an anomaly detector that statically taints source code and identifies anomalous or missing conditions linked to security-critical objects in the source code. After extracting the abstract syntax tree, the k-nearest neighbour algorithm is used to perform neighbourhood discovery of related functions. Lightweight tainting followed by the embedding of functions into vector space then allows comparison for missing checks by geometrically comparing the check with the known checks for similar function embeddings in the rest of the code base, identifying anomalies. Chucky was tested to diagnose known vulnerabilities with a very high detection rate, and it was also able to assist analysts to identify previously unknown vulnerabilities by producing a ranked list of anomalies from various open-source projects. The *Joern* project [89] introduces the concept of a code property graph (CPG), a combination of abstract syntax tree, control flow graph (CFG) and program dependency graph (PDG), three existing representations of code, each of which is able to capture some, but not all characteristics of software important for vulnerability research. The idea is that CPGs combine these characteristics to offer a single representation that can be used to model existing vulnerabilities more generally. Checking code bases then becomes an exercise in building and representing the code as CPGs, and issuing graph traversal queries against the resulting graphs to find matches for vulnerability patterns. The platform has been tested against the Linux kernel code base and was able to identify 18 previously unknown vulnerabilities. A limitation of the design is being partially addressed in a proposal to automate the inference of search patterns for taint-style vulnerabilities [91]. Post-dominator trees (PDT) are

used to augment code property graphs to capture situations where statements are executed before another, enabling the detection of functions calls in code that result in modifications of their arguments. These can then be used to generate graph traversal patterns to search for taint-style vulnerabilities. The approach has been implemented as a plug-in for Joern and was shown to produce substantial reductions in code inspections.

3.1.4. Bug fixes and patching

Bug fixing, whether reactive or preventative, remains a necessary activity within software development. *Vulture* [57] is a tool that automatically learns from the locations of past vulnerabilities to predict the future vulnerability of new components before they are fully implemented. This is accomplished by extracting import statements from source code components and applying frequent pattern mining to determine whether they relate to existing vulnerabilities. A new component can then be evaluated to determine whether it will be vulnerable based on its imports using a classification model built on the project's import matrix and the vulnerability vector attained from the previous step. The system, using an SVM classifier for prediction, was evaluated on the Mozilla project. Kim et al. [43] proposes a technique called "change" classification that operates on lines of code rather than complete modules, files or functions. This is achieved by examining the code history in software configuration management systems. Features are extracted for both bug fixes and regular commits from change metadata, software complexity metrics, log messages and code. An SVM classifier is chosen to build a model to predict if a new code change is buggy or clean, with results available immediately at the time of a new change commit. The predictive power of different groups of features are also analysed in the paper.

Knowledge of the vulnerable components in a software project does not, however, provide an answer to the question of how likely these vulnerabilities can be exploited. Automatically ranking vulnerabilities has been proposed as a potential solution by Bozorgi et al. [15], utilising public vulnerability databases such as CVE and OSVDB to classify and predict the vulnerabilities most likely to be exploited. A high number of features are extracted from text fields, time stamps, cross-references and other entries in existing vulnerability disclosure reports. A linear SVM is then trained on a random balanced sample using positive labels for vulnerabilities that have exploits and negative labels for those that do not. This methodology is then engaged to study offline and online exploit predictions, identifying features most relevant to predictions and estimating the time it would take for an exploit to appear for a vulnerability. The latter topic is also investigated in Wijayasekara et al. [86] which looks at the mining of bug reports in bug databases using text mining techniques to aid the discovery of hidden impact vulnerabilities. A hidden impact vulnerability is a vulnerability identified sometime after the related bug has been disclosed to the public usually via the release of a patch, which an attacker can use to discover a potentially high impact exploit. A feature vector is obtained by processing (tokenising, stemming etc.) the text in the bug report, and a classification is obtained by calculating the Bayesian detection rate, or the probability that a bug is a hidden impact vulnerability given that it detects a bug as a vulnerability. The Linux kernel and MySQL bug databases were used as data sources. An analysis of the ratio of hidden impact vulnerabilities to vulnerabilities has shown it to be relatively high and it was additionally observed to be on the increase in the last two years of the study.

Once bugs are identified and prioritised, they must be fixed. *Prophet* [51] is an example for an automated patch generation system that obtains patches from open-source software repositories and builds a model of correct code. This probabilistic model is learnt in an offline training phase from features of successful patches extracted from previous code revisions and is used to generate and prioritise candidate patches for new defects. The candidate patches are then validated and offered for manual inspection and insertion by a developer. Tests on a benchmark set of 69 real-world defects drawn from eight open-source projects show that Prophet compares favourably to existing patch generation systems. *GenProg* [84] uses a genetic programming approach on abstract syntax trees and weighted program paths to correct defects in code. Once an error is identified, for example, by the program failing a test case, variants of the original program are searched until a valid version is found. The technique uses the observation that a defect may be repaired by adopting existing code from another location in the program. By mutating the defective code using similar templates, a repair can be identified by the program successfully passing the previously failed test case. The *DeepSoft* framework [29] is an ambitious approach to model software and its development in order to predict and mitigate risks and to automatically generate code patches for bugs identified. It employs deep learning, specifically long short-term memory, to model source code and its evolution. It is proposed that this representation, in conjunction with natural language processing enables the automatic generation of code patches for resolving issues. Le et al. [48] investigate the validity of applying automatic repairs for identified cases of software error. They are interested in finding repairs in a reasonable time, and argue that sometimes manual rather than automatic intervention may be more cost effective according to some defined time budget. To this effect, they build a random forest classifier that uses multiple runs of GenProg to generate data, with an effectiveness indicator added as the classification label. The consequent model is used as an oracle to predict the effectiveness of future repair instances. Results indicate that around three out of four repairs are correctly identified for the more suitable repair type.

3.1.5. Mitigation and prevention

Ways to reduce the number of bugs that are introduced during the software development process have also often been explored in the literature. One possible course to lessen the probability of errors in code is to guide the automatic generation of correct code snippets. Code completion is discussed by many authors, including Hindle et al. [36], who use the n-gram concept from natural language processing to statistically model code token sequences, based on the assumption that code is like language, repetitive and predictable. They find that the entropy of source code is much lower than that of natural language. They develop a new Eclipse plug-in from their model as a proof-of-concept which is shown to outperform the capabilities of the built-in code completion engine. Allamanis and Sutton [7] take this research further by compiling and analysing much larger data sets, scaling up to "giga-token" models and introducing new data-driven metrics to measure code complexity. They demonstrate that these models are even better at code suggestion. *SLANG* [66] complements n-gram models with recurrent neural networks to address the problem of generating completions for "holes" (gaps, missing lines) in code with API method calls. Code completion is treated as the natural-language processing problem of predicting probabilities of sentences. The resulting tool is able synthesise complex solutions with multiple statements and arguments that typecheck correctly and include the desired outcome in the top 3 results in 90% of the cases. *DeepSyn* [65] is a

code completion system for JavaScript programs. It utilises a domain specific language over abstract syntax trees, which removes code specifics and facilitates learning on partial parse trees. A program in this language that best matches training data can then be generated which is shown to perform better at the code completion task than existing solutions.

Data mining techniques for intelligent code completion are discussed by Bruch et al. [17]. Three separate solutions are devised to better the built-in Eclipse code suggestion system. The first simply orders all available suggestions by frequency counts as opposed to the default Eclipse ordering. The second uses association rules to find correlations between code objects and suggests those that are closely associated with an observed object. The third solution is a variation on the k-nearest neighbours algorithm, called the best matching neighbours algorithm. It extracts binary feature vectors for each variable in the code base, encoding indicators about API calls that use them and calculates distances between the current and the example code base based on a modified use of the Hamming distance. Code completions are then recommended based on their frequencies in selected nearest neighbours. In testing, each proposal outperformed the default Eclipse suggestion system, with the best matching neighbours algorithm producing the leading results.

GraLan [58] is a graph-based approach to statistical language modelling which computes the appearance probabilities of usage graphs and uses them for code completion. After the language model is built from source code examples, usage subgraphs can be extracted from the neighbourhood of the currently edited code, and *GraLan* can be used to compute the probabilities of the children graphs given those usage subgraphs. These are collected and ranked as candidate API elements for suggestion. This approach is further extended into *ASTLan*, an AST-based language model to suggest syntactic templates rather than API elements at the current editing location. Both *GraLan* and *ASTLan* compare favourably to previous efforts in API code and syntactic template suggestions, with the paper also offering a broad overview of code completion research.

Machine learning techniques have also been used to help with proving the memory safety and functional correctness of programs. An example is *Cricket* [16], a verification tool extension that utilises logistic regression and two-layered neural networks to automate the annotation of programs with appropriate invariants. Initially, only shape properties are learnt and used to verify the correctness of heap programs. If this fails, valid shape invariants are strengthened in a second stage with data invariants to improve the obtained memory safety proof, again employing ML algorithms.

3.1.6. Attribution

A potential follow-up activity after code vulnerability analysis is the attribution of code to its author(s). Such information may identify relationships between software projects, can attest to the quality and maturity of the code, and assist in developing and applying suitable solutions for vulnerability prevention. Caliskan-Islam et al. [24] address the authorship attribution problem for `C++` source code using machine learning. Their approach considers three types of features for a total of up to 20,000 features: code stylometry features are layout features such as whitespaces that do not change the meaning of a program; lexical features are derived from program tokens, or strings with identified meaning such as the number of loops, if/then

statements and comments; and syntactic features are derived from ASTs like the term frequency inverse document frequency (TFIDF) of AST node types. Using the information gain criterion, the original set of features were substantially reduced and a random forest ensemble classifier was used for authorship attribution with highly accurate results. The authors suggest that syntactic features are resilient to code obfuscation attempts and find that advanced programmers have a more identifiable/unique coding style than novices.

3.2. Binary code analysis

Binaries are a transformation of source code, and as such, suffer from some loss of semantics inherent in high level programming languages. Therefore, some of the activities in vulnerability research, in particular those aiming to prevent vulnerabilities from being introduced into software, are either unavailable, not applicable or require a different approach or technique. When both source code and the corresponding binary can be analysed, researchers have an opportunity to work with a rich set of data. Binaries, however, are often all that are available and that can be investigated. Similar to source code, they can be transformed into different representations, which may make some of the earlier seen techniques appropriate for use with this medium. However, different computer architectures and different compiler optimisations on the same source code can introduce an additional layer of complexity in the analysis of a binary.

Automation remains a primary goal in SVR activities on binary code. Some examples include *GUEB* [32], a static tool detecting use-after-free (UaF) vulnerabilities on disassembled code. It utilises an abstract memory representation on which a value set analysis is performed, using forward traversal of the control-flow graph to facilitate UaF detection. *Sword* [23] is an automated fuzzing system that prioritises areas of a binary for fuzzing. It combines multiple vulnerability detection approaches to increase efficiency, namely symbolic execution which searches for and identifies desired execution paths of the target program, and taint analysis to check the execution paths and generate path-dependent information in order to guide the fuzzer performing the vulnerability analysis. *Code Phage* [75] is a system to automatically transfer correct code from donor applications into a recipient application to eliminate errors in the latter. Once an error-causing input is identified for a program, a database is searched for a donor application where the same input does not trigger the error. Various activities, including candidate check discovery, patch excision, insertion and validation follow this step, and are repeated if they do not produce a safe solution. The paper also includes an overview of earlier program repair efforts. The opposite of fixing, i.e. exploiting a program automatically based on the examination of patches issued for it is discussed by Brumley et al. [20]. The idea is to locate any new sanitisation checks introduced in a patch and find the inputs that they safeguard against. Attackers can leverage this information, assuming that these inputs could be potential exploits for the un-patched programs, which often do not receive updates in a timely manner.

Code similarity can also be leveraged for vulnerability identification. Tree Edit Distance Based-Equational Matching [64] is a method used to automatically identify binary code regions similar to another that contains a known bug. In a pre-processing phase, semantic information in the form of expression trees that summarise the results of computations performed in the basic block, is extracted. For a known bug, this is used as a signature to locate similar code

regions, using a block-centric metric based on tree edit distances to measure the similarity of binary code. The metric allows for small syntactic differences in code and has been used to identify unknown vulnerabilities in software such as forks of the popular SSH client `PuTTY`. A generalisation of this signature-based bug finding approach across multiple CPU architectures using a code similarity metric is proposed in Pewny et al. [63].

3.2.1. Data structures

A fundamental task in binary program analysis is the understanding of program characteristics that enable further investigation into potential issues hidden within the code. *Laika* [28] uses Bayesian unsupervised learning to identify data structures and their instantiations in a program image. It locates pointers to potential data objects and estimates their sizes using a conversion of machine words into vectors of block types as features. Similar sequences are then clustered together using a probabilistic similarity measure to determine classes of data structures. The data structures generated were tested in a virus detection scenario and proved highly effective. White and Lüttgen [85] use genetic programming to identify dynamic data structures from program execution traces by matching them against pre-defined templates for labelling. The templates capture the operations necessary to manipulate complex data structures such as linked lists and binary trees, and tests show that this approach is able to infer such data structures with a low false positive rate.

3.2.2. Program structures

Structures within binaries also extend to executable code. Rosenblum et al. [68] address the 'function start identification' problem, or the recognition of the entry points of functions (FEPs) in stripped binary code. They use a supervised classification algorithm based on the Conditional Random Fields (CRF) statistical modelling method. Every byte offset is assumed as a candidate FEP, with two models considered: one based on the classification of individual FEPs, the other on structural aspects such as functions that can call other functions. The first model uses short instruction patterns as the set of features with a logistic regression model formulated as a conditional random field used for classification. The second model uses a CRF to model the function call interactions using pairwise function/structural calls and overlap features. The approach compares favourably against results achieved using existing tools such as *Dyninst* [37] and the commercial disassembler *IDA Pro*. *ByteWeight* [12] improves this approach by employing a technique based on prefix trees to learn signatures for function start identification purposes and then recognising functions by matching binary fragments with the signatures. This is done through training a classification model on a reference corpus of source code and their corresponding binaries where the function addresses are known. The model takes the form of a weighted prefix tree which encodes function start byte sequences: an actual function start is detected if the corresponding sequence terminal node in the prefix tree has a weight value larger than a given threshold. Once a function start is found, further techniques can be used to extract the full functions from the binary.

Describing the general structure of binaries to enable detailed analysis is a goal explored in many publications. For example, *BAP* [19] is a multi-purpose analysis platform to perform program verification and analysis tasks on binaries. Like many similar solutions, it converts binary instructions into an intermediate language to enable subsequent analyses to be written

in a syntax-directed form. Tasks made available this way can include creating control flow and program dependence graphs, eliminating dead code, performing value set analysis and generating verification conditions according to given postconditions.

3.2.3. Dynamic analysis

VDiscover [34] is a system that applies a machine learning approach that uses scalable, lightweight static and dynamic features extracted from a binary to predict if a vulnerability test case is likely to contain a software vulnerability. Static features are extracted from disassembled code by detecting potential sub-sequences of library function calls obtained from a random walk of the program call graph. Dynamic features are captured from execution traces containing concrete function calls augmented with their arguments. To reduce the large number of feature values, argument values are subtyped and are preprocessed by considering each execution trace of function calls as a document, and using latent semantic analysis for dimensionality reduction. The machine learning classification algorithms tested in the system include logistic regression, multilayer perceptrons and random forests. Random forests gave the best results in a large experiment designed to identify programs with memory corruptions.

3.2.4. Symbolic execution

Symbolic execution explores the execution space of a program and generates test cases with high coverage. A limitation of the approach is that the number of possible program paths are often excessively large. Various schemes have been proposed to guide the exploration of program paths according to some selected strategy to overcome this issue. Li et al. [49] focuses on less travelled paths by examining the frequency distribution of length-n subpaths in order to improve test coverage and error detection. The idea is to use statistical analysis of the already covered subpaths by maintaining a count of the number of times the subpath has been explored before. Then, for a given execution state, the subpath with the lowest count is chosen to continue execution. The strategy has been implemented in the symbolic execution engine *KLEE* [21] and tested on GNU Coreutils programs. Cadar and Sen [22] provide a overview of similar proposals and discuss challenges related to symbolic execution. In particular, they focus on higher levels of automation for tasks such as pruning redundant paths, generating inputs that results in errors and even the creation of corresponding exploits. Avancini and Ceccato [10] use genetic algorithms and symbolic execution to generate test cases for the identification of cross-site scripting (XSS) vulnerabilities in web applications. They combine the two techniques by first executing a genetic algorithm and evolving a population of test cases for some generations before switching to symbolic execution. Symbolic constraints are collected, solved and reinserted as new input values and the execution returns to the genetic algorithm. This process is repeated until some pre-defined termination conditions are met. Test show a higher coverage achieved with this combined technique when contrasted against random test case generation and both methods used separately.

BitScope [18] is an example platform for the automatic path-based analysis of malicious binaries. Its architecture contains components to monitor the flow of information in and out of the malicious binary, to prioritise available paths for exploration and to perform combined concrete and symbolic (concolic) execution of the binary to build formulas to express the satisfiability of execution paths. Extractor modules then generate the control-flow graph of

discovered code, the inputs required by the binary to travel along different execution paths and dependency information between the inputs and outputs of the binary. This data can then be used by practitioners to further analyse the program.

Similarity function based automatic vulnerability discovery (SFAVD) [53] is a process that combines machine learning from source code and testing with symbolic execution in order to detect vulnerabilities. First, a set of functions similar to a known vulnerable one is generated using a multi-step process that includes creating an abstract syntax tree representation for each function, enumerating them into feature vectors and performing a similarity assessment against the vulnerable function. Then, function calls graphs are generated for the functions with the largest similarity and KLEE is used with a constraint solver to determine if they are vulnerable. Tests show that a significant reduction in execution time can be achieved when compared to analysing the program using just KLEE.

3.2.5. Malware

Although not strictly part of the various activities of software vulnerability research, malware analysis is one of the related areas where machine learning has received considerable attention and shown to be helpful. Malware analysis can also assist in developing vulnerability mitigation strategies and often employs techniques directly relevant to other SVR tasks, exposing opportunities to use the same or similar ML approaches in those areas. The analysis of malware often starts with the identification of pertinent features. Ahmed et al. [6] extracts statistical features from spatial and temporal information available from Windows API call arguments and sequences. They include means, variances and entropies of address pointers and size parameters and features obtained from discrete time Markov chain modelling of API sequences. A variety of machine learning algorithms are compared using these features, including an instance based learner, a decision tree classifier, Naïve Bayes, an inductive rule learner and a support vector machine, and they indicate that improved results can be achieved when using a combination of spatial and temporal features for classification. *OPEM* [71] also checks for malware with a range of algorithms, such as decision trees, Bayesian classifiers, support vector machines and k-nearest neighbour, but uses a different set of features. Static features are based on the frequencies of fixed-length sequences of operational codes found in malware. Dynamic features are extracted from execution traces as a binary vector representing the presence of specific system calls, operations, and raised exceptions within the executable. Feature reduction is applied to static features using information gain in order to produce a manageable feature set. Results show that the combined static and dynamic features produce better results than when features of only a single type are used. Anderson et al. [9] describe an example when only features of one type, in this case those from dynamically collected instruction traces are used. Instruction sequences are transformed into a Markov chain representation, forming a directed instruction trace graph (ITG), represented by a weighted adjacency matrix. A two-class support vector machine is used for classification and produces highly accurate results. Unfortunately, the graph kernels used to determine the similarity between ITGs can be expensive to compute for large graphs. Gascon et al. [33] also uses an SVM but it derives its features based on the structural embeddings of function call graphs by employing an explicit mapping inspired by a linear-time graph kernel, thus producing a static feature set. In tests, the malware detection rate achieved with this technique is reasonably high, with a low false positive rate.

Saxe [42] suggests an ensemble of similarity measures to defeat obfuscation in malware to better identify the class to which they belong. This is an important task as determining the correct group for a malware sample can assist with attribution and follow-up activities. The idea is that full obfuscation of the entire malware is difficult, and therefore using multiple techniques for analysis should yield reliable results even in cases when an individual similarity measure is ineffective. The four measures proposed are based on PE (portable executable) metadata similarity, dynamic similarity, instruction-gram similarity and file similarity. This approach is a simpler and a more scalable solution to the identification of malware groups than other available techniques such as clustering.

3.2.6. Attribution of binaries

Authorship analysis may also be performed on binaries. Rosenblum et al. [69] collect a corpus of programs with known authorship and use control flow graphs and instruction sequences to extract and determine relevant stylistic features from each program based on the mutual information between features and labels. These features are tested using a support vector machine classifier, and used to learn a metric that minimises the distance in the feature space between programs by the same author. The metric is then used with k-means clustering to group programs of unknown authors. The technique achieves reasonable performance (94% of the time the correct author is ranked within the top five), which leads the authors of the paper to conclude that programmer style can be preserved through the compilation process. Several improvements on this technique are proposed subsequently. Alrabaee et al. [8] argue that a more layered approach is necessary. First, a pre-processing layer filters out library code to eliminate functions unrelated to style. A code analysis layer then extracts binary code blocks that correspond to source code vocabularies to build author profiles for classification. Finally, a register flow analysis layer which uses register flow graphs that characterise how registers are manipulated, is utilised to define author style signatures and to identify program authors. Attribution accuracy is shown to have improved over the results achieved in [69]. However, neither of these approaches are specifically designed to handle programs with more than one author. Meng [54] introduces new block level features to attribute code written by teams. After analysing the machine instructions generated from attributed source code [55], the basic block is determined as a suitable candidate to attribute individual authors. Consequently, several new features are proposed at this level to complement existing ones from previous efforts. A linear SVM classifier on basic blocks achieved comparable results to earlier reports, suggesting that authorship identification is practical both at the basic block level and for multiple authors.

4. Summary

In this paper we report on some of the recent activities of the software vulnerability research community that involve the use of machine learning. Although not furnishing an exhaustive list, we aim at presenting a representative collection of publications that illustrate the use of ML in support of SVR. We find that apart from new ways of finding solutions tackling large code bases, the most prolific theme in SVR is that of helping practitioners such as code auditors, by simplifying or automating their processes wherever possible. Machine learning is one of the tools that is used increasingly to improve on traditional approaches where appropriate. It has

been applied with success to solve individual SVR tasks or through multiple steps to provide guidance with complex processes. In this role, ML should continue to offer benefits in the future. There still remain areas within SVR that have seen limited use of ML (for example, fuzzing [38]), and even those that currently utilise machine learning could be improved as new, more effective and efficient techniques are proposed. Identifying these trends and recognising their suitability to specific SVR problems is not an easy task, and requires close ties among SVR practitioners, and program analysis and machine learning experts. In particular, we believe that using machine learning with semantic input information rather than its application on syntactic features could produce more meaningful results for many SVR activities. Finally, we would like to recognise a couple of recent efforts which aim at creating greater awareness of and encourage collaboration for the purposes of software vulnerability research. The 2016 Cyber Grand Challenge [2] was a competition between researchers to evaluate software, test for vulnerabilities and automatically generate and apply security patches on networked computers. The *MUSE* project [56] is an ongoing program seeking experts from multiple disciplines to use big data analytics to improve software reliability by developing approaches for automatically constructing and repairing software programs.

Acknowledgements

The authors would like to thank Paul Montague and Junae Kim for their comments and reviews of earlier versions of this manuscript.

5. References

[1] Category:OWASP Top Ten Project. [Online] https://www.owasp.org/, 2016.

[2] Cyber Grand Challenge. http://archive.darpa.mil/cybergrandchallenge/, 2016.

[3] Category:Attack - OWASP. [Online] https://www.owasp.org/, 2017.

[4] CVE - Common Vulnerabilities and Exposures (CVE). [Online] https://cve.mitre.org/, 2017.

[5] R. Agrawal, T. Imieliński, and A. Swami. Mining Association Rules Between Sets of Items in Large Databases. SIGMOD'93, pages 207–216, New York, NY, USA, 1993. ACM.

[6] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq. Using spatio-temporal information in api calls with machine learning algorithms for malware detection. In *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence*, AISec'09, pages 55–62, New York, NY, USA, 2009. ACM.

[7] M. Allamanis and C. Sutton. Mining Source Code Repositories at Massive Scale Using Language Modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR'13, pages 207–216, Piscataway, NJ, USA, 2013. IEEE Press.

[8] S. Alrabaee, N. Saleem, S. Preda, L. Wang, and M. Debbabi. OBA2: An Onion approach to Binary code Authorship Attribution. *Digital Investigation*, 11, Supplement 1:S94–S103, May 2014.

[9] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane. Graph-based Malware Detection Using Dynamic Analysis. *Journal in Computer Virology*, 7(4):247–258, Nov. 2011.

[10] A. Avancini and M. Ceccato. Comparison and Integration of Genetic Algorithms and Dynamic Symbolic Execution for Security Testing of Cross-site Scripting Vulnerabilities. *Information and Software Technology*, 55(12):2209–2222, Dec. 2013.

[11] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic Exploit Generation. *Communications of the ACM*, 57(2):74–84, Feb. 2014.

[12] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 845–860, Berkeley, CA, USA, 2014. USENIX Association.

[13] D. Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.

[14] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, Oct. 2007.

[15] M. Bozorgi, L. Saul, S. Savage, and G. M. Voelker. Beyond Heuristics: Learning to Classify Vulnerabilities and Predict Exploits. In *Proceedings of the Sixteenth ACM Conference on Knowledge Discovery and Data Mining*, pages 105–113, Washington DC, USA, 2010. ACM.

[16] M. Brockschmidt, Y. Chen, B. Cook, P. Kohli, S. Krishna, D. Tarlow, and H. Zhu. Learning to Verify the Heap. Technical Report MSR-TR-2016-17, Microsoft, Seattle, WA, USA, Apr. 2016.

[17] M. Bruch, M. Monperrus, and M. Mezini. Learning from Examples to Improve Code Completion Systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE'09, pages 213–222, Amsterdam, The Netherlands, 2009. ACM.

[18] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. BitScope: Automatically dissecting malicious binaries. Technical report, In CMU-CS-07-133, 2007.

[19] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A Binary Analysis Platform. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, number 6806 in Lecture Notes in Computer Science, pages 463–469. Springer Berlin Heidelberg, July

2011.

[20] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP'08, pages 143–157, Washington, DC, USA, 2008. IEEE Computer Society.

[21] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, San Diego, California, USA, 2008. USENIX Association.

[22] C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM*, 56(2):82–90, Feb. 2013.

[23] J. Cai, S. Yang, J. Men, and J. He. Automatic software vulnerability detection based on guided deep fuzzing. In *2014 IEEE 5th International Conference on Software Engineering and Service Science*, pages 231–234, Beijing, China, June 2014. IEEE.

[24] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt. De-anonymizing Programmers via Code Stylometry. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 255–270, Berkeley, CA, USA, 2015. USENIX Association.

[25] R.-Y. Chang, A. Podgurski, and J. Yang. Discovering Neglected Conditions in Software by Mining Dependence Graphs. *IEEE Transactions on Software Engineering*, 34(5):579–596, Sept. 2008.

[26] C. Cifuentes and B. Scholz. Parfait: Designing a Scalable Bug Checker. In *Proceedings of the 2008 Workshop on Static Analysis*, SAW'08, pages 4–11, Tucson, Arizona, USA, 2008. ACM.

[27] J. Cohen. Contemporary Automatic Program Analysis. Black Hat USA 2014, Las Vegas, Nevada, 2014.

[28] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for Data Structures. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 255–266, San Diego, California, USA, Dec. 2008. USENIX.

[29] H. K. Dam, T. Tran, J. Grundy, and A. Ghose. DeepSoft: A Vision for a Deep Model of Software. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 944–947, New York, NY, USA, 2016. ACM.

[30] M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, Indianapolis, Ind., 1 edition, Nov. 2006.

[31] T. Fawcett. An Introduction to ROC Analysis. *Pattern Recognition Letters*, 27(8):861–874, June 2006.

[32] J. Feist, L. Mounier, and M.-L. Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, 10(3):211–217, 2014.

[33] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural Detection of Android Malware Using Embedded Call Graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec'13, pages 45–54, New York, NY, USA, 2013. ACM.

[34] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY'16, pages 85–96, New York, NY, USA, 2016. ACM.

[35] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 Projects: Lightweight Cross-project Anomaly Detection. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA'10, pages 119–130, New York, NY, USA, 2010. ACM.

[36] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, pages 837–847, Zurich, Switzerland, June 2012. IEEE.

[37] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pages 841–850, Knoxville, Tennessee, USA, May 1994. IEEE.

[38] M. Höschele and A. Zeller. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 720–725, New York, NY, USA, 2016. ACM.

[39] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden. Software Vulnerability Prediction using Text Analysis Techniques. In *International Workshop on Security Measurements and Metrics*, pages 7–10, Lund, Sweden, Sept. 2012. ACM.

[40] F. Inc. CVSS v3.0 Specification Document. [Online] https://www.first.org/cvss/specification-document, 2017.

[41] M. Jimenez, M. Papadakis, T. F. Bissyandé, and J. Klein. Profiling Android Vulnerabilities. In *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 222–229, Vienna, Austria, Aug. 2016. IEEE.

[42] Joshua Saxe. A scalable, ensemble approach for building and visualizing deep code-sharing networks over millions of malicious binaries. Black Hat USA 2014, Las Vegas, Nevada, 2014.

[43] S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, Mar. 2008.

[44] T. Kremenek, A. Y. Ng, and D. Engler. A Factor Graph Model for Software Bug Finding. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pages 2510–2516, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[45] P. Kreutzer, G. Dotzler, M. Ring, B. M. Eskofier, and M. Philippsen. Automatic Clustering of Code Changes. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR'16, pages 61–72, New York, NY, USA, 2016. ACM.

[46] Kymberlee Price and Jake Kouns. Epidemiology of Software Vulnerabilities: A Study of Attack Surface Spread. Black Hat USA 2014, Las Vegas, Nevada, 2014.

[47] F.-M. Lazar and O. Banias. Clone detection algorithm based on the Abstract Syntax Tree approach. In *2014 IEEE 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 73–78, Timisoara, Romania, May 2014. IEEE.

[48] X.-B. D. Le, T.-D. B. Le, and D. Lo. Should fixing these failures be delegated to automated program repair? In *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering*, pages 427–437, Washington DC, USA, Nov. 2015. IEEE.

[49] Y. Li, Z. Su, L. Wang, and X. Li. Steering Symbolic Execution to Less Traveled Paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA'13, pages 19–32, New York, NY, USA, 2013. ACM.

[50] Z. Li and Y. Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *Proceedings of the 10th European Software Engineering Conference*, ESEC/FSE-13, pages 306–315, Lisbon, Portugal, 2005. ACM.

[51] F. Long and M. Rinard. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'16, pages 298–312, New York, NY, USA, 2016. ACM.

[52] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent Dirichlet allocation. *Information and Software Technology*, 52(9):972–990, Sept. 2010.

[53] Q. Meng, S. Wen, B. Zhang, and C. Tang. Automatically discover vulnerability through similar functions. In *Proceedings of the 2016 Progress in Electromagnetic Research Symposium*, pages 3657–3661, Shanghai, China, Aug. 2016. IEEE.

[54] X. Meng. Fine-grained Binary Code Authorship Identification. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1097–1099, Seattle, WA, USA, 2016. ACM.

[55] X. Meng, B. P. Miller, W. R. Williams, and A. R. Bernat. Mining Software Repositories for Accurate Authorship. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, pages 250–259, Eindhoven, The Netherlands, Sept. 2013. IEEE.

[56] S. Neema. Mining and Understanding Software Enclaves (MUSE). [Online] http://www.darpa.mil/program/mining-and-understanding-software-enclaves, 2017.

[57] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting Vulnerable Software Components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS'07, pages 529–540, New York, NY, USA, 2007. ACM.

[58] A. T. Nguyen and T. N. Nguyen. Graph-based Statistical Language Model for Code. In *Proceedings of the 37th International Conference on Software Engineering*, volume Volume 1 of *ICSE'15*, pages 858–868, Piscataway, NJ, USA, 2015. IEEE Press.

[59] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan. Mining Preconditions of APIs in Large-scale Code Corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 166–177, New York, NY, USA, 2014. ACM.

[60] Y. Pang, X. Xue, and A. S. Namin. Predicting Vulnerable Software Components through N-Gram Analysis and Statistical Feature Selection. In *Proceedinggs of the 14th IEEE International Conference on Machine Learning and Applications*, pages 543–548, Miami, FL, USA, Dec. 2015. IEEE.

[61] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin. Building Program Vector Representations for Deep Learning. In S. Zhang, M. Wirsing, and Z. Zhang, editors, *Knowledge Science, Engineering and Management*, number 9403 in Lecture Notes in Computer Science, pages 547–553. Springer International Publishing, Oct. 2015.

[62] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS'15, pages 426–437, New York, NY, USA, 2015. ACM.

[63] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-Architecture Bug Search in Binary Executables. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP'15, pages 709–724, Washington, DC, USA, 2015. IEEE Computer Society.

[64] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow. Leveraging Semantic Signatures for Bug Search in Binary Programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC'14, pages 406–415, New York, NY, USA, 2014. ACM.

[65] V. Raychev, P. Bielik, M. Vechev, and A. Krause. Learning Programs from Noisy Data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'16, pages 761–774, New York, NY, USA, 2016. ACM.

[66] V. Raychev, M. Vechev, and E. Yahav. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'14, pages 419–428, New York, NY, USA, 2014. ACM.

[67] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[68] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt. Learning to Analyze Binary Computer Code. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, pages 798–804, Chicago, Illinois, 2008. AAAI Press.

[69] N. Rosenblum, X. Zhu, and B. P. Miller. Who Wrote This Code? Identifying the Authors of Program Binaries. In *Computer Security - ESORICS 2011*, pages 172–189. Springer, Berlin, Heidelberg, Sept. 2011.

[70] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, May 2009.

[71] I. Santos, J. Devesa, F. Brezo, J. Nieves, and P. G. Bringas. OPEM: A Static-Dynamic Approach for Machine-Learning-Based Malware Detection. In Á. Herrero, V. Snášel, A. Abraham, I. Zelinka, B. Baruque, H. Quintián, J. L. Calvo, J. Sedano, and E. Corchado, editors, *International Joint Conference CISIS'12–ICEUTE'12–SOCO'12 Special Sessions*, number 189 in Advances in Intelligent Systems and Computing, pages 271–280. Springer Berlin Heidelberg, 2013.

[72] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Predicting Vulnerable Software Components via Text Mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, Oct. 2014.

[73] H. Shahriar and M. Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Computing Surveys*, 44(3):11:1–11:46, June 2012.

[74] B. Shastry, F. Yamaguchi, K. Rieck, and J.-P. Seifert. Towards Vulnerability Discovery Using Staged Program Analysis. In J. Caballero, U. Zurutuza, and R. J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, number 9721 in Lecture Notes in Computer Science, pages 78–97. Springer International Publishing, July 2016.

[75] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'15, pages 43–54, New York, NY, USA, 2015. ACM.

[76] D. Steidl and N. Göde. Feature-based Detection of Bugs in Clones. In *Proceedings of the 7th International Workshop on Software Clones*, IWSC'13, pages 76–82, Piscataway, NJ, USA, 2013. IEEE Press.

[77] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically Inferring Security Specifications and Detecting Violations. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 379–394, Berkeley, CA, USA, 2008. USENIX Association.

[78] Y. Tang, F. Zhao, Y. Yang, H. Lu, Y. Zhou, and B. Xu. Predicting Vulnerable Components via Text Mining or Software Metrics? An Effort-Aware Perspective. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 27–36, Washington, DC, USA, Aug. 2015. IEEE.

[79] S. Thummalapenta and T. Xie. Alattin: Mining Alternative Patterns for Detecting Neglected Conditions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 283–294, Auckland, New Zealand, Nov. 2009. IEEE.

[80] Tyler Bletsch. *Code-Reuse Attacks: New Frontiers and Defenses*. Ph.D., North Carolina State University, Raleigh, North Carolina, 2011.

[81] Y. Udagawa. Source Code Retrieval Using Sequence Based Similarity. *International Journal of Data Mining & Knowledge Management Process*, 3(4):57–74, July 2013.

[82] J. Vanegue, S. Heelan, and R. Rolles. SMT Solvers for Software Security. In *Proceedings of the 6th USENIX Conference on Offensive Technologies*, WOOT'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[83] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining Succinct and High-coverage API Usage Patterns from Source Code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR'13, pages 319–328, Piscataway, NJ, USA, 2013. IEEE Press.

[84] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE'09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[85] D. H. White and G. Lüttgen. Identifying Dynamic Data Structures by Learning Evolving Patterns in Memory. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 354–369, Rome, Italy, 2013. Springer-Verlag.

[86] D. Wijayasekara, M. Manic, J. L. Wright, and M. McQueen. Mining Bug Databases for Unidentified Software Vulnerabilities. In *Proceedings of the 2012 5th International Conference on Human System Interactions*, HSI'12, pages 89–96, Washington, DC, USA, 2012. IEEE Computer Society.

[87] J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Static Intrusion Prevention. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems*, pages 68–

84, Karlstad, Sweden, Nov. 2002.

[88] C. Williams and J. Spacco. SZZ Revisited: Verifying when Changes Induce Fixes. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, DEFECTS'08, pages 32–36, New York, NY, USA, 2008. ACM.

[89] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014.

[90] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC'12, pages 359–368, New York, NY, USA, 2012. ACM.

[91] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812, San Jose, CA, USA, May 2015. IEEE.

[92] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. Chucky: exposing missing checks in source code for vulnerability discovery. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 499–510, Berlin, 2013. ACM Press.

[93] M. Zhao, A. Laszka, T. Maillard, and J. Grossklags. Crowdsourced Security Vulnerability Discovery: Modeling and Organizing Bug-Bounty Programs. Austin, TX, USA, Nov. 2016. AAAI Press.

DEFENCE SCIENCE AND TECHNOLOGY GROUP DOCUMENT CONTROL DATA			1. DLM/CAVEAT (OF DOCUMENT)
2. TITLE A Review of Machine Learning in Software Vulnerability Research			3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION) Document (U) Title (U) Abstract (U)
4. AUTHORS Tamas Abraham and Olivier de Vel			5. CORPORATE AUTHOR Defence Science and Technology Group PO Box 1500 Edinburgh, South Australia 5111, Australia
6a. DST GROUP NUMBER DST-Group–GD–0979	6b. AR NUMBER AR-017-005	6c. TYPE OF REPORT General Document	7. DOCUMENT DATE October, 2017
8. OBJECTIVE ID	9. TASK NUMBER	10. TASK SPONSOR	
11. MSTC Cyber Assurance and Operations		12. STC Cyber Defence Analytics	
13. DOWNGRADING/DELIMITING INSTRUCTIONS		14. RELEASE AUTHORITY Chief, Cyber and Electronic Warfare Division	
15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT *Approved for Public Release* OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111			
16. DELIBERATE ANNOUNCEMENT No Limitations			
17. CITATION IN OTHER DOCUMENTS No Limitations			
18. RESEARCH LIBRARY THESAURUS software vulnerability research, machine learning, source code analysis, binary code analysis, computer security, software security, program analysis			
19. ABSTRACT Searching for and identifying vulnerabilities in computer software has a long and rich history, be that for preventative or malicious purposes. In this paper, we investigate the use of Machine Learning (ML) techniques in Software Vulnerability Research (SVR), discussing previous and current efforts to illustrate how ML is utilised by academia and industry in this area. We find that the primary focus is not only on discovering new approaches, but on helping SVR practitioners by simplifying and automating their processes. Considering the variety of applications already in evidence, we believe ML will continue to provide assistance to SVR in the future as new areas of use are explored and improved algorithms to enhance existing functionality become available.			