Information Sciences 524 (2020) 46–58

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Code analysis for intelligent cyber systems: A data-driven approach

Rory Coulter[a], Qing-Long Han[a,*], Lei Pan[b], Jun Zhang[a], Yang Xiang[a]

[a] School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, VIC 3122, Australia
[b] Deakin University, School of Information Technology, Geelong, VIC 3220, Australia

ARTICLE INFO

Article history:
Received 9 June 2019
Revised 12 March 2020
Accepted 13 March 2020
Available online 14 March 2020

Keywords:
Cyber security
Code analysis
Machine learning
Malware detection
Vulnerability discovery

ABSTRACT

Cyber code analysis is fundamental to malware detection and vulnerability discovery for defending cyber attacks. Traditional approaches resorting to manually defined rules are gradually replaced by automated approaches empowered by machine learning. This revolution is accelerated by big code from open source projects which support machine learning models with outstanding performance. In the context of a data-driven paradigm, this paper reviews recent analytic research on cyber code of malicious and common software by using a set of common concepts of similarity, correlation and collective indication. Sharing security goals in recognizing anomalous code that may be malicious or vulnerable. The ability to do so is not determined in isolation, rather drawn for code correlation and context awareness. This paper demonstrates a new research methodology of data driven cyber security (DDCS) and its application in cyber code analysis. The framework of the DDCS methodology consists of three components, i.e., cyber security data processing, cyber security feature engineering, and cyber security modeling. Some challenging issues are suggested to direct the future research.

1. Introduction

Cyber security is a fundamental requirement in a new digital era, which greatly differs from traditional information security. In the context of information security, information is the target of the attack and the defense. However, cyber security considers not only the information but also people and objects surrounding the cyber space [24,71,87]. Some new malware and ransomware attacks have caused severe damage to infrastructure and financial losses [14,16,34,40,77]. Many advanced malware and ransomware attacks rely on security vulnerabilities existing in the software of the operating systems of computers and/or I/O devices [8,34]. Furthermore, the association of ransomware with crypto-currency complicates the trace-back and analysis [30].

Recently, as typical cyber security research, Android malware detection and software vulnerability discovery have both adopted the use of Machine Learning (ML) in aiding security challenges. Increases to data availability, processing improvements and technological advancements are such reasons [12,37]. But with this, traditional challenges have been replaced with new ones. Cyber security specialists are not ML specialists and vice versa. We observe that these cyber security studies apply the same data-driven research methodology, and share many elements besides being code based. The nature of each

* Corresponding author.
E-mail address: qhan@swin.edu.au (Q.-L. Han).

https://doi.org/10.1016/j.ins.2020.03.036
0020-0255/

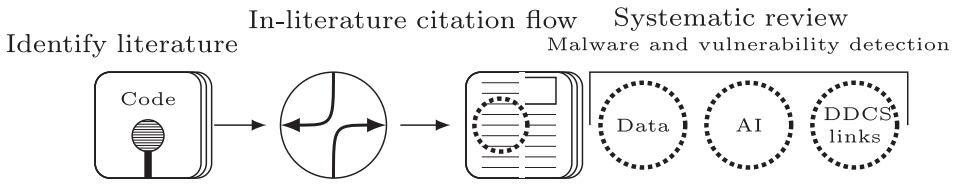

Fig. 1. Code perspectives of DDCS: It provides a systematic review of Android malware and vulnerability processes, where literature is first selected from these two fields. The flow of internal citations is measured for selecting influential literature. Data, AI and shared DDCS elements are then systematically surveyed.

methodology seeks to remove and identify individual code elements, which are either malicious in combination or vulnerable. The atomic power of individual code segments, individual or unison, provides more discriminative power than that of the code base.

Data-driven methods are spread across literature from visualization [7], determining flu trends [35] and across several cyber security fields [48]. The large abundance of data demands complex mechanisms for monitoring and protecting the appropriate use of the cyber space [47,77,79,90]. Therefore, approaches using traditional statistics and analysis [36] are gradually replaced by big data methods [4,26,28,53] in order to reveal new knowledge [41,74]. Data outcomes are now producible for cyber security such as Android malware and software vulnerability detection through the application of ML. Such a challenge area for delivering data outcomes is IoT cyber-physical systems where agency is now possible. Much of what allows the IoT to be beneficial in its mix of ML, sensors, and systems is in its ability to harness data and knowledge exported through data-driven outcomes. Excluding poorly labeled data and aging ML models, adversarial attacks and data poisoning are poised to cause disruption, as securing intrusions remains a challenging prospect for ML and IoT requirements [13].

Recent related surveys on either Android malware [20,21,55,73,75] or software vulnerability detection [25,68] focus on the application of various ML models. There is a lack of a unified data-driven framework for code analysis purposes. To bridge the gap, we present a new framework of data-driven cyber security (DDCS) with three aspects — *cyber security data processing, cyber security feature engineering*, and *cyber security modelling*. The major contributions of this survey are summarized as follows:

- To demonstrate the idea, concept, and methodology of DDCS for code analysis in cyber security;
- To provide a systematic review based on historical developments, methodologies, and solutions of malware detection and software vulnerability discovery; and
- To identify the limitations and challenges of current research from the DDCS perspective.

The paper is organized as follows: Section 2 introduces the DDCS methodology with some examples of code analysis for cyber security. Section 3 reviews the key milestones of recent research on Android malware detection. Section 4 concentrates on the review of key works in software vulnerability detection, and a summary of research boundary and challenges is provided in Section 5. Section 6 concludes this paper.

2. A new framework of DDCS for code analysis

This section presents a new framework of DDCS for code analysis. The new framework consists of three components — cyber security data processing, cyber security feature engineering, and cyber security modeling. Fig. 1 outlines the elements which contribute to this DDCS framework.

2.1. Cyber security data processing

In most cases of ML based code analysis, labeled data, clear processing and substitute knowledge to aid explainability is required. Labels are domain and problem specific. In Android malware analysis cases, there are typically two labels — malware and goodware (or clean-ware); in the software vulnerability detection case, there are usually two labels as well — vulnerable or not vulnerable. In some rare cases, malware samples can be assigned to the malware family-specific labels, and software code to vulnerability-specific labels. Proper processing allows for the unification of scattered elements, removal of noise and appropriate presentation for ML, while additional knowledge obtained and explored during the processing state helps link data and features to explaining the influence, or the ML result to users.

To label Android malware samples, popular choices are either manual inspection or signature matching. Manual inspection is accurate but certainly time-consuming. An automated scan by matching the signatures of known malware samples enhances the efficiency but often fails to detect the zero-day samples. Two of the most popular Android malware datasets with ML capabilities are *Drebin* [1] and *Marvin* [45]. To label software code with respect to vulnerabilities, manual inspection is necessary [42–44]. More specifically, code data is treated differently with respect to their source of origin [42]: For code existing within the NVD database, a piece of code is labelled as vulnerable if it contains at least one statement that is deleted or modified according to its patch; otherwise, the code piece is labelled as not vulnerable. Code form the SARD database contains code pieces labelled as good, bad, or mixed, where each piece in the mixed set needs to be manually

verified. Due to the large number of code pieces, there are often conflicting labels where the same code piece is labelled with both vulnerable and not vulnerable; samples with conflicting labels are often removed [42].

The process of tidying data not only helps present it correctly for ML use, but also allows additional information to be learnt, enabling the ability to help support explainable AI. Code exists in the form of human readable statements, interpreted by upon execution. These statements require proper transformation into machine readable formats, often through text processing matrices [2]. Therefore, the processing process has the ability to influence downstream results. Across the different processing techniques, its application plays a key role in not only understanding the security application, but providing clean and accurate data/code which can be interpreted by ML models and users. This practice was a key feature of the authors of *Drebin* [1], with thorough processing allowing threat information to be communicated to users. This was done by interpreting code statements to human understandable interpretations. This process therefore provides the basis for context for downstream items. Recent trends have also further aimed to reveal black-box models this practice as a whole over the entire ML process [95]. Explainable AI has recently sought to provide evidence and influence behind the decision making process.

2.2. Cyber security feature engineering

Effective features are essential to detect Android malware. Each Android App package contains a manifest file in plain text and a `dex` binary code file. Information including permissions, intents and API calls can all be expressed as strings. According to [1], there are eight categories of the strings — 1) hardware components, 2) requested permissions, 3) App components, 4) intents, 5) restricted API calls, 6) used permissions, 7) suspicious API calls, and 8) network addresses. The first four string sets can be obtained by using the *Android Asset Packaging Tool* from the `Manifest.xml` file which stores the installation and execution information; the last four string sets can be obtained by decompiling the `dex` binary file into Java source code with a decompiler such as *dex2jar* and *jd*. In particular, requested permissions are often different to the used permissions; the restricted API calls refer to those API calls requiring the administrative privilege. Furthermore, suspicious API calls include the following five groups [1] — API calls obtaining sensitive data regarding device and subscriber, API calls intervening the network communication, API calls sending and receiving SMS messages without user's consent, API calls capable of executing remote function calls, and API calls used for obfuscation. Finally, *Drebin* [1] uses a binary vector to represent each sample with respect to the above-mentioned features.

In addition to the above-mentioned static features, *ANDRUBIS* [46] extracts the run-time behaviors through monitoring and interacting with a Dalvik VM. Six categories of dynamic events are investigated: 1) File operations including the action type (read or write) and the file name; 2) network operations including communication protocols, the destination IP address and destination port; 3) mobile phone events including outgoing calls and sent SMS; 4) data leaks including personal identifiable information, network information and file names; 5) dynamically loaded code including run-time code, its file name and the dependent class name; and 6) dynamically registered broadcast receivers. The dynamic events are combined together with the static features to obtain a set of 490,000+ features [45]. Moreover, in the context of encrypted network traffic, using web app-specific operations as dynamic features is proven to be effective [11], for example, sending a new email through Gmail, refreshing Twitter's home page, accessing the Dropbox App, and so on.

The large number of Android malware features poses a serious challenge for human beings to comprehend. That is, most features extracted by different authors are named slightly differently so that it often causes ambiguity. A web-based system named *FeatureSmith* is proposed to align the Android malware features to the semantic meanings explained in the literature [94]. Common properties of Android malware samples are aligned to features, for example, many malware Apps performing an SMS fraud request the `SEND_SMS` permission [15,94]. By grouping the similar properties and the associated features, *FeatureSmith* consolidates a small feature space of 195 features. Furthermore, it annotates each feature with readable comments, for example, `getSimOperatorName` has the following behaviors — "return privacy-sensitive information, leak privacy-sensitive return value, leak to remote web server" [94].

Similar to Android malware analysis, software vulnerability detection heavily involves code analysis techniques; a vulnerability may be associated with API calls, values of the parameters, function names and so on. An exhaustive list of all possible elements will result in an extremely high dimension for further analysis. Hence, features must be properly confined without losing semantic relationships between program elements. To preserve data dependency and control dependency, program source code is transformed to an intermediate representation which can be subsequently converted to a vector as the input to neural networks [42]. For example, *VulDeePecker* [42] slices program source code into gadgets to isolate code blocks with respect to API calls and position information (line number); then it collects all tokens from the code gadgets and maps user-defined variable and function names. The process reduces the number of tokens from 6,166,401 to 10,480 [42].

An abstract syntax tree (AST) can be used to construct program representations [43,44]. An AST captures a program's structure information at a function level. A depth-first traversal of the AST generates a vector representation. According to [44], *word2vec* [52] is applied to convert the parsing results of the tool *CodeSensor* to 100-dimensions vectors. Different programs elements are distinct to each other, being operators, the function names and the variable types. The isolation of each type helps associate vulnerability with code information at the function level.

2.3. Cyber security modeling

Appropriate performance estimation is vital in determining whether a given model or system fulfils its needs. While accuracy alone is an insightful estimator, a cost is incurred as models become excessively tailored, or inadequate experience is applied. Those with the best state exposure are allowed to make the most rational decision [3]. In order to objectively discern the appropriateness of the process, several ML performance metrics are used throughout the literature.

A typical process of training ML models involves two divided datasets, namely training and testing: Training data primarily consists of samples collected in the past, and testing data contains more up-to-date samples. Due to availability issues of novel malware or vulnerability samples, training sets are usually much larger than testing sets. Furthermore, a validation dataset [32] with a mixture of historical and current samples is a pragmatic approach when high quality samples are too few, particularly with respect to malware [64]. The validation dataset can spare some high quality data in the testing set. If the data quality of the testing dataset is high, then performance metrics will be truly indicative.

In a simplified scenario of binary classification, malware samples or software vulnerabilities which belong to a class x, correctly classified as class x are recorded as true positive (TP). Those not belonging to x and not classified as x are known as true negative (TN). Loss occurs when samples are either incorrectly classified or missed. Those classed as x but determined y are known as false negative (FN). And those labeled as y, determined as x are known as false positive (FP). The most commonly used metrics are derived on the basis of a combination of TP, FP, FN, and TN, such as accuracy. Additional estimators measure existence or proportional representation, e.g., precision and recall.

The above metrics indicates the statistical quality of a trained ML model with respect to the testing dataset. Strong values of the performance metrics only mean that the model fits the data well. That is, the best model is often chosen by carefully balancing between a few independent metrics, such as between TPR and FPR, or between F-Measure and recall. For example, a vulnerability detection model can be chosen according to the largest FPR and the lowest FPR [42]. Furthermore, techniques like cross validation are often used to minimize the data bias [43,44,81]. Additionally, visual plots are often used to compare different models. An example is recall versus the number of predictions [43,44]; and an advanced example is a Receive Operating Characteristic (ROC) curve which examines TPR as y-axis and FPR as x-axis [1,31].

Unfortunately, real-world cyber security datasets are often imbalanced such that the most recent samples are significantly ewer than historical samples. The data imbalance issue affects the value of trained ML models. Selecting a similar numbers of samples from each malware family is used to offset the data imbalance biases [64]. Adjusting the weight values of records and adding the estimated samples to the training dataset are proposed in [51]. Another possible solution is ensemble learning [49]. Hence, many high quality publications justify the models not only through the performance metrics but also through the new knowledge discovered by using the model. For example, many Android malware papers have reported the findings of new malware families [1,92,93]. However, many existing classification models are based on simple algorithms like SVM which can be compromised by data-driven approaches [50]. Therefore, cautions should be taken while choosing the candidate models in the presence of imminent data-driven attacks.

3. Data driven android malware analysis

3.1. Android malware as data

Android users acquire Apps through various means, with the ability to install outside the Google market place. While the Google Play Store resides as the official market place, various other markets and websites exist for numerous reasons. For example, some provide region and language specific Apps, and others are clear attempts to proliferate malware. In conjunction with these locations, several malware repositories exist too, notably the *Contagio* Android malware repository [58]. The acquisition of applications can be acquired with relative ease. In determining the nature of a given application, several methods exist. Manual observation remains one such method, but given the sheer amount of applications to examine, manual methods become impractical, therefore *AVclass* was presented in 2016 [66] to leverage several techniques, where many commercial tools are used by researchers to determine the class and malware family labels. *VirusTotal* [76] is one of the most popular services to aid labeling. Other commercial tools exist, but not limited to *AntiVir, AVG, BitDefender, ClamAV, ESET, Kaspersky* and *McAfee*.

Over the lifespan of the Android mobile system, researchers have collected and labeled many samples. The lack of publicly available dataset of labeled Android malware existed for a considerably long period of time after the release of the Android system [6,19,38,67,92]. The *Android Malware Genome Project* is one of the first labeled datasets released in 2012 [92]. The *Android Malware Genome* dataset contained a total of 1260 Android malware samples, belonging to 49 individual families. The *Drebin* dataset was released in 2014 [1] — malicious samples increased to 5,560, totalling 131,611 including benign applications. However, it was noted that not only was the original *Drebin* data outdated, it too is not supported [78]. Thus, an updated dataset was released in response of 24,650 samples, collected between 2010 through 2016 [78]. Furthermore, the concept of data drift was shown in examining the *Drebin* dataset [33]. The *Drebin* and *Marvin* [45] datasets were compared, revealing that the *Drebin dataset* was inadequate in training for the evolving threat landscape.

3.2. Machine learning for detection

3.2.1. Supervised model

The use of APIs as an aid in classification has been popular among approaches. The mobile malware characteristics were anticipated via a framework for behavioral detection within the Symbian OS landscape [6]. Symbian is a precursor to Android, yet an initial stepping stone. A mixture of simulated and real world samples were analyzed specifically through the use of lower-level APIs and system calls for signature construction. SVM was applied in classifying signatures for viruses, worms and Trojans. The use of SVM was extended in [23] where call graphs were extracted and labeled, nodes neighbourhoods were hashed, feature spaces were embedded and classified via a linear-kernel SVM. A total of 147,950 samples (135,792 benign and 12,158 malicious) were collected and classified, and then classification results were compared through 10 commercial AV scanners. Maintaining supervised learning but transitioning to newer techniques, *Droiddetector* is an online deep learning (DL) detection engine [88]. *Droiddetector* was trained by using 20,000 Apps acquired from the official market and 1760 samples from the *Contagio* repository [92]. Sensitive APIs and permissions were collected statically as well as the dynamic analysis for behaviours. The primary static and dynamic analysis makes use of deep learning, yet also C4.5, SVM, Näive Bayes (NB), Logistic Regression (LR), and static or dynamic top features per class for a Deep Belief Network (DBN). In conjunction with classification, repackaging attacks are explored for ten antivirus products.

3.2.2. Unsupervised model

Decision Tree (DT), NB, Bayesian Networks (BN), *K*-Means, Histogram or LR classifiers were evaluated in determining behavior and classification through various feature selection techniques [67]. In *Andromaly* [67], twenty benign games and twenty benign tools were combined with self written malicious programs for classification surrounding information extraction, theft-of-service, unsolicited information, and DoS attacks. A filter approach was used for API feature selection, applying Chi-square (CS), Fisher score (FS) and information gain (IG). Among the experiments NB featured the most prominent, followed by DT and LR, while feature selection techniques of CS and IG were shown to be very similar. *DroidMat* [80] focused on an advanced feature analysis using the Contagio dataset [58] — 1738 samples were collected, 238 malware over 34 families and 1500 benign from the official market, with fifty from each market category. Feature selection was applied in combination of APIs, permissions and intent components. The *K*-Means algorithm was applied using Singular Value Decomposition (SVD) in obtaining an optimal K value as $K = 1$. The performance of Known-Nearest Neighbour (*K*-NN), NB and Expectation-Maximisation (EM) algorithms were compared in [80]. As a result, recall was best achieved with intent plus API, while the combination of permission, intent and API impacted precision the best.

Employing unsupervised methods still, yet focusing on more specific attacks, *RiskRanker* was presented for analyzing a specific application behaviour, raking risks for further inspection [27]. High risk were associated with exploitation of platform vulnerabilities; medium risk is associated with financial misuse or information leakage; and low risk is associated with low grade device information gathering. 104,874 applications were retrieved from the official market (49.78%) and from 14 alternative markets (50.22%) between September and October in 2011. From each application, APIs and permissions were extracted for analysis, first and second order analysis was conducted in identifying high and medium risk application per order, native code signature were compared for high first order risks, while medium risk control and data flow analysis was conducted. Second order required manual analysis for potential high risk applications as well as the encrypted native code execution and the potential unsafe Dalvik code execution. *RiskRanker* is primarily constructed for root-exploit, SMS/call misuse and leakage.

3.3. Malware campaign analytics

3.3.1. Behavior related analysis

The Android landscape has to be shaped by the introduction of analysis regarding the health of the market space and noteworthy tools. A market analysis concerning application security and a Dalvik decompiler tool named *ded* were presented in 2011 [18]. The top 1100 free Android Market applications were surveyed comprising over 21 million lines of code. The analysis was concerned with — 1) surveying previous discovered issues, 2) identification of general code security flaws, and 3) identification of security flaws within the Android framework. Analysis was employed through four approaches: control-flow, data-flow, structural, and semantic analysis. A total of 27 key findings were reached regarding identifiers, location information, telephony misuse, background audio/visual misuse, socket API use, investigation pre-installed applications, analytic and advertisement libraries, developer toolkits, information leakage, unprotected broadcasts, intent injection, delegation control, null checks, SD-card, and Java native interface (JNI) use.

The broader landscape of mobile malware (iOS, Symbian and Android) was surveyed from samples collected between January 2009 through June 2011 [22]. Forty-six samples collected during this period (4 iOS, 24 Symbian, and 18 Android) concerned investigation of their behavior, the ability of current defenses, and future threats [22]. Notable potential issues ranged from SMS misuse, privilege escalation and the effect the homebrew root exploit community has in gifting vulnerability avenues.

The overall health of the five existing official and unofficial markets (Google Play, eoeMarket, alcatelclub, gfan, and mmoovv) were evaluated in [93]. The analysis was conducted on 204,040 applications collected between May through June

2011. In analyzing a large quantity efficiently, permission-based behavioral footprinting and heuristic filtering (dynamic loading of remote Java binary code from unknown websites and local native code) were implemented as *DroidRanger*. It led to the discovery of 211 malicious applications, with heuristic filtering identifying two zero-day families among 40 samples. Empirical results showed that among free applications, low infections rates existed. It was also noted that application submission approval processes were inadequate [93], alongside with long delays for responding to user feedback.

The feasibility that malware samples could be monitored within a sandbox environment through a cloud service was argued in 2010 [5]. As an initial attempt, a list of primitive ideas and hypothetical scenarios was provided without evaluating real-world cases [5]. As a result, the duration of signature generation was still a lengthy process around 48 days. A loadable kernel module (LKM) was implemented for use within an Android emulator environment, with the ability to perform static, dynamic and polymorphic analysis.

Evasion techniques were evaluated against commercial and state-of-the-art antivirus tools of publishing, titled *DroidChameleon* [63]. Samples were chosen from the *Contagio* mini dump before obfuscation techniques were applied. Samples were inspected by 10 AV scanners (AVG, Symantec, Lookout, ESET, Dr. Web, Kaspersky, Trend micro, ESTSoft, Zoner, and Webroot) where products were susceptible to transformation attacks. At least 43% of signatures were not based on code-level elements so that static bytecode analysis is not required for 90% of signatures. It was shown that high-level bytecode is much more indicative than native code regarding obfuscated malware [63]. Hence, bytecode analysis is a promising candidate for further future investigation of Android malware.

The tools against evasion techniques particularly within a virtual machine (VM) environment were evaluated in 2014 [61]. The authors identified several evasion techniques used to obfuscate the malware payload embedded into malware samples without causing significant changes to functionality. Detection is concerned with static properties, dynamic sensors, and VM specific nuances. On the basis of the empirical studies, effective evasion can be easily accomplished through simple modifications such as trivial masking, bit flipping and so on. Furthermore, attackers fingerprinting an execution environment can be inferred with little programming efforts, so that only one tool provided limited information about evasion techniques. Four key countermeasures were recommended [61], including modification of emulators, sensor event simulation, accuracy of binary translation, and hardware-assisted virtualization artifacts. Hence, VM execution information needs to be better protected in order to reveal the malicious behavior of Android malware. Fingerprinting concepts have also enable security mechanisms in which underlying data-driven uses and systems trust are operated with higher belief [62].

3.3.2. Wild analysis

An anomaly detection framework was proposed in 2008 [38] through monitoring, detecting and analyzing power consumption. It heralded an introduction for mobile handset classification prior to mainstream Android development. The detection framework sought to collect consumption specifications for which a signature was created. Subsequently, similarities were assessed by a CS distance measure. Moreover, a concept WiFi malware program and a worm consuming Bluetooth signals were constructed [38].

Kirin was presented in 2009 [19] as a certification framework evoked during an application installation establishing security rules. *Kirin* aids to determine whether the application is safe or not. The release of *Kirin* sat at the cusp where proof-of-concept approaches were replaced with legitimate malicious samples. That is, 311 samples among the most popular samples were selected and downloaded from the official market for analysis. The security service possessed specification like policy which assessed the existence and conjunction of given features which may suggest malicious behaviour. *Kirin* rules were associated with hidden API use, voice/location eavesdropping and application replacement. An initial analysis led to the discovery of several features messages operated in an unprotected manner. For example, `SMS_RECEIVED` allowed forged messages and activity components surrounding, and `CALL_PHONE` allowed for unknown call activation. These bugs were subsequently fixed. Of the applications analyzed, ten categories possessed permissions considered dangerous, five categories potentially malicious, and the remaining five categories possessed dangerous but in-scope permissions.

TaintDroid was proposed in 2009 [17] to track privacy sensitive data flows among third-party applications. *TaintDroid* leverages Androids host VM based execution environment. The taint analysis was modified to integrate he Android platform and `file/message/method/variable` level propagation. 30 applications were selected from one of the previous works in [18] tracking data flows, excluding implicit. Results showed that sensitive information (number, International Mobile Subscriber Identity (IMSI), Integrated Circuit Card Identifier (ICC-ID) and geographic location) were secretly sent by two applications; six applications sent the device ID, and twelve sent location based information. Unfortunately, the leak of personal information without user's explicit consensus still exists in today's Apps.

3.4. Learned lessons

Several key lessons exist for the detection of Android malware, namely data, and machine learning maturity. Simply put, data spurs discovery. Datasets like The *Android Malware Genome Project* [92] and *Drebin* [1] have provided highly influential access to data. Both were very well received. Therefore, data is pivotal in maintaining efficient detection. With the coming of deeper ML methods, Android malware detection has employed a range of traditional algorithms, followed by effort in optimizing parameters and features. Interestingly, the use of various algorithms for a given objective still remains, without one clear choice. As observed in Table 1, a variety of algorithms have been used for security contexts of challenges, across many different threat landscapes.

Table 1
A summary of literature and their data, approach and security challenges for Android malware detection.

Data Sources	ML Methods	Refer-ence	Methodology and security challenges	Features and approaches
Self collected data	Supervised	[6]	SVM for detecting Worms, Virus & Trojan	API & system calls for behaviour detection
		[23]	Call graph and linear SVM for benign vs. malicious detection-BvM	Static approach
	Unsupervised	[67]	NB, DT, BN, LR, and Histograms in detecting information extraction, resource theft & DoS	API and static
		[27]	Control and data flow used with signatures for root-exploits, SMS/call misuse and information leakage	API, permission used for static method
	n/a	[38]	A^2 distance regarding battery/Bluetooth misuse	Static behaviour
		[19]	Detect hidden API and application replacement	API, permission and intent through static rule policy
Dataset	Unsupervised	[80]	SVD, EM, *K*-Means, *K*-NN, and NB for benign vs. malicious detection	API, permission and intent for static method
	Taint	[17]	benign vs. malicious detection	Dynamic method
Mixed	Mixed	[88]	DBN for benign vs. malicious detection	API, permission for a mix of static and dynamic methods

4. Data driven software vulnerability analysis

4.1. Bid code of software

Introduction of vulnerable code statements can occur any time during the development, updating or patching of software applications. Combating vulnerabilities remains difficult, as knowing which programming statements may be vulnerable remains unknown. It presents many challenges in determining how to detect susceptible code fragments, as manual auditing is past the means of programmers. Furthermore leaving the task to another program is unachievable [29]. Data can be found in numerous places; generally, open source programs are used given the availability of the source code. The Mozilla code base, LibTIFF, LibPNG, FFmpeg, Pidgin, VLC or RHEL kernels have been typical code bases used, where labeling of code fragments drawn from known examples. These may come from a variety of sources including the national vulnerability database (NVD) and common vulnerabilities and exposures (CVE), or Bugzilla.

Data requires labelling to fulfill the requirements of matching code to a given vulnerability. Many approaches take a known vulnerability, select an appropriate model and train based on its characteristics [60,83,84,86]. While this sees targeted and rewarded auditing, the cultivation of datasets remains narrowly focused. For each new vulnerability, this process must be repeated, if not continually. This fact leads to fit for purpose datasets, where little is applicable after characteristics are no longer of interest. However, two datasets have been released for the research community for use. Six open source projects were collected where there were manually labeled 475 vulnerable functions amongst 32,531 non-vulnerable [44]. A dataset tailored for deep learning was provided with the vulnerabilities mapped from NVD and CVE sources [42]. The data were sourced from NVD and SARD, with respect to buffer overflows and resource management errors. The dataset consists of 61,638 code snippets. Rather than matching known examples, the harnessing of representations of vulnerabilities may provide a promising wider net for unknown code segments [44].

4.2. Transition from statically mining to pattern recognition

4.2.1. Statistical mining of vulnerabilities

The Mozilla code base has received a wide range of attention given its large size, high level of contributions, and wide scale of code. The unrestricted association of vulnerabilities to imports or function calls used within components were investigated in 2017 [56]. Instead of the calling of imports or functions, they were shared among similar vulnerable components. Past vulnerabilities were mapped to the Mozilla base from the Mozilla Foundation Security Advisories (MFSA), refined to 576 includes and 2470 function calls. An SVM classifier was used to reduce the amount of code to inspect to 45%, among which 70% were determined vulnerable.

A statistical analysis of the JaveScript engine (JSE) was conducted in determining the difference among complexity code metrics in predicting vulnerable source code [70]. The Wilcoxon rank sum was used as the indicator for determining the difference between non-vulnerable and vulnerable, with later versions of the JSE (v1.5.0.8, v2.0, and v2.0.0.4) showing significant differences in the nesting complexity. Moreover, the likelihood ratio (LR) test was used for predicting faulty or vulnerable from all, and vulnerable from faulty. The LR method struggled with different versions of JSE. A statistical ML approach was used in complexity, coupling, and cohesion metrics (CCC) metrics [10]. When determining the discriminative value of each CCC metric, the prediction performance of C4.5, RF, LR, and NB was compared. 52 releases of Mozilla Fire-

fox were used, resulting in 718 vulnerable files determined from MFSA. Overall, C4.5 performed the best among the four algorithms. C4.5 was further tested on newer released versions of Firefox.

The Mozilla code base remained popular with complexity, code churn and developer usage metrics were further explored in 2011 [69]. LR and other ML classifiers were applied to the Firefox browser and to the RHEL4 kernel. According to [69], neutral files are almost always more simple than vulnerable files; less code churn exists in neutral files; and the best discriminative metric set seemed to be developer metrics. A reduction of 72% for Firefox and between 51%–72% for RHEL were achieved in code sample identification. Static analysis was adopted in 2010 [57], with the use of member dependency graph (MDG), static code elements, and component dependency graphs (CDG).

Concentrating of other popular open source code bases, function level detection [43] was investigated based on cross-project abstract syntax trees (ASTs). LibTIFF, LibPNG and FFmpeg code bases were obtained, for a total 5736 functions with 274 vulnerable functions. ASTs were extracted and compared with essential complexity metrics for representation learning in a recurrent neural network (RNN) with Long Short-Term Memory (LSTM) (bi-directional) cells [43]. The evaluation was compared to RF trained with complexity metrics. Learned representations exhibited greater predictive influence. Adopting the same approach, a cross-domain transfer representation learning was explored in 2018 [44]. The evaluation using RF was again made with 23 traditional code metrics. LibTIFF, LibPNG, FFmpeg, Pidgin, VLC Media Player, and Asterisk were used for Bi-LSTM. Deep learning methods are becoming a promising approach to software vulnerability prediction.

4.2.2. Vulnerability pattern recognition

Regarding software vulnerability discovery, the correlation between software usage and potential vulnerabilities was reported in [83]. This approach has been proven effective by empirical studies where FFmpeg (v0.6.0) and RHEL kernel (v2.6.32) were chosen as the targets; source code functions were tokenized, referenced by type and function as APIs; and tokenized information was processed with PCA to isolate dominant elements. Having condensed similar vectors, 6778 functions were reduced to 20 vulnerable functions. Among the 20 functions, there was one zero-day vulnerability. Further vulnerability extrapolation used abstract syntax trees (ASTs) [84]. LibTIFF, FFmpeg, Pidgin, and Asterisk projects were used and extracted into a code base. Using the queries based on the mapped CVEs, the improved approach reduced inspection to 8.7% on average [84] and several new vulnerabilities were discovered through ASTs and CVE mapped queries.

Taint-style analysis was successfully applied for missing code checks [86]. Firefox, RHEL, LibPNG, and Pidgin were used as the code base, API vectors again were used to identify neighboring elements in conducting taint analysis based for CVEs, so that anomaly scores were derived. Based on these anomaly scores, taint analysis aided in identifying 12 vulnerabilities. Furthermore, source code was modeled through CPGs in 2014 [82]. Analysis based on known CVEs resulted in the discovery of 18 unknown RHEL kernel vulnerabilities. An anomaly approach in conjunction with CPGs and taint aspects were joined [85]. A source code repository was built together with the existing CVEs to discern the code of interest. A complete-linkage clustering (CLC) was applied to control flow graphs (CFG). Taint style analysis is an effective method for vulnerability identification.

VUDDY is a vulnerability detector associated with a scalable clone [39]. The retrieved functions are fingerprinted with MD5 hashes, so that a key lookup distinguishes code clones and vulnerabilities for a hash lookup of Git code. An Apache HTTPD zero-day vulnerability was identified as `CVE-2012-0876` among their comparison of several other approaches (SourcererCC, ReDeBug, CCFinderX, DECKARD), as well as an improvement of scalability. Moreover, a deep learning method was used in the vulnerability detector *VulDeePecker* in 2018 [42]. It aimed to reduce FPs, compared with [39] and other pattern-based and similarity-based approaches. The aims of using deep learning methods were to relieve experts in feature selection of library/API function calls. Nineteen projects were selected and combined from the National Institute of Standards and Technology (NIST), SARD and the NVD. Adapting the data for the use of deep learning, the learning phase produced vulnerability patterns which were used within a Bi-LSTM network.

In 2015 [60], it was discovered the ease of which vulnerable code slips past assurances, and the constrained FPs that exist among tools. The released database consists of 66 individual GitHub projects together with mapping commits to 718 CVEs [60]. A Bag-of-Words (BoW) model was combined with a linear SVM for code metrics with GitHub meta features; conversely, the *Flawfinder* tool was applied to identify vulnerability contributing commits (VCC). FPs were reduced by 99%. Furthermore, for contributors with less than 1% of all given commits for a project, the rate of committing a vulnerability is five times higher. It was questioned whether text mining is applicable for a prediction model in 2014 [65], that is, can the model withstand for future predictions and is cross-project application achievable? Ten applications were selected from the *F-Droid* repository along with ten pre-installed applications. Java files were tokenized into a vector of terms, employing a BoW method, for terms and frequency with a class label.

4.3. Learned lessons

Several key lessons cant be learnt from software vulnerability analysis, namely the reliance of OSS, and the prospect of predicting vulnerabilities. OSS features heavily for as the main data source for vulnerability detection. Granted that code bases are open and available for use, they do not constitute the majority of applications and programs used, with closed source Windows programs still the majority share. Therefore, many vulnerabilities are left to persist, subject to the discovery of a vulnerability for which malicious actors are not inclined to alert developers of any weaknesses found. The algorithms used also are now heavily dominated by deeper techniques. Often supporting their use are algorithms which enable greater

Table 2
Software vulnerability detection summary of all discussed literature .

Data Sources	Methods	Refer-ence	Methodology and vulnerability sources	Features
Mozilla	Prediction	[56]	SVM trained from MFSA	n/a
		[70]	Wilcoxon rank sum trained from MFSA	Complexity
		[57]	MDG, CDG and WEKA suite trained from MFSA	Graph attributes
		[10]	J48, RF, LogitBoost and NB trained from MFSA	Complexity, coupling and cohesion
Linux	Pattern	[82]	Graph traversal of LIBSSH2_ALLOC	n/a
OSS	Prediction	[69]	LR, NB, J48, RF and BN trained from MFSA	Complexity, code churn and developer activity
		[65]	DT, kNN, NB, RF and SVM trained from Fortify results	n/a
		[43]	Bi-LSTM and RF trained from NVD and CVE	Complexity
		[44]	Bi-LSTM trained from NVD and CVE	Complexity
		[42]	Bi-LSTM trained from NVD, SARD and CVE	n/a
	Pattern	[83]	Cosine similarity drawn from CVE	
		[84]	LSA drawn from CVE	
		[85]	CLC and CPG drawn from CVE	
		[39]	Hash lookup of matching CVE	
	Anomaly	[86]	Distance measure of matching CVE	
GitHub	Prediction	[60]	SVM trained from CVE	n/a

explainability. Early approaches used a range of techniques, with tree-based based methods proving popular. With the use of models like Bi-LSTM, new approaches in predicting vulnerable code have opened up as seen in Table 2.

5. Research challenges and future work

5.1. Data, processing and intelligence

There is a real need for new data, as current datasets for Android malware research languish comparing to current challenges [59]. The situation of free and paid application further complicates acquiring large samples, as well as the sheer amount of application to consider. Initial research attempts were concerned for on-device practices, offline classification had become predominant, where many process steps are involved that may not be applicable for real-time classification. Feature engineering and optimization have flourished, too the investigation of various ML methods in obtaining as much out of the data. Multi-class, ensemble methods, attack based detection (ignorant of a family membership), and deep learning may prove a key future step for the development of malware detection.

Interpreting the strategy of vulnerability detection, or *prevention*, helps define the means in which vulnerabilities are sought, and why such research trends exist. Considering the purpose of prevention and detection, comparisons to traffic analysis reveal the fundamental nature of intelligent code analysis as vulnerability prevention. Cyber security traffic analysis relies on combining many weak indicators of compromise in determining the nature of traffic, this is analogous with intrusion detection, and therefore why the two are so closely related and shaped by particular methodologies [96]. Therefore, vulnerable code prevention is an overarching concept defining the methods for which vulnerabilities are firstly sought, or malicious code uses are found. The mantra of prevention seeks to determine and mitigate potential issues in advance, thus parallels are easily drawn of intelligent code analysis, determine vulnerabilities before the release of programs or applications. But unlike traffic analysis, many intelligent code analysis methods discussed do not rely on combining indicators, but rather instances of patters or prediction.

As no complex program is released in a perfect fashion, almost all programs are known to have vulnerabilities. According to Rice's theorem, determining code which may be vulnerable by another program is infeasible [29]; therefore manual investigation remains prevalent on how we use and interpret data. The addition of propriety software would yield beneficial given how popular some software is, but unlikely to happen. Therefore if ML models will be used for OSS and in-house testing, we are currently challenged on how we can apply ML. Approaches fall into one of two areas (prediction v.s. pattern) in reducing the amount of code to investigate. The utility of purpose has seen a transition away from reducing code amounts to actual vulnerability finding. That is, graph-based methods remain purposeful, tree-based algorithms too, with deep learning becoming an exciting solution.

There are some research trends on intelligent code analysis for malware detection and vulnerability discovery. Malware is typically approached as a binary problem. Multi-class classification may prove more beneficial than that of a one-fits-all approach. It is further strengthen by family representation is not uniform and vary in their attacks. Tailored, or more refined classification may yield improved results. Greater attention towards obfuscation and adversarial permutations. They persist with relative ease [9]. With the focus of vulnerability location, many efforts seek to match, find similar code fragments or identify statistical factors, rather than the problem location. Furthermore, finding ways to improve labelling techniques that currently rely on manual work, without introducing noise or artifacts in the data. In addition, the reliance on features either needs to be substituted or robust features determined. The ML methods have improved significantly, but the features used remain the same. And knowledge representation is emerging as a promising avenue to explore for removing feature

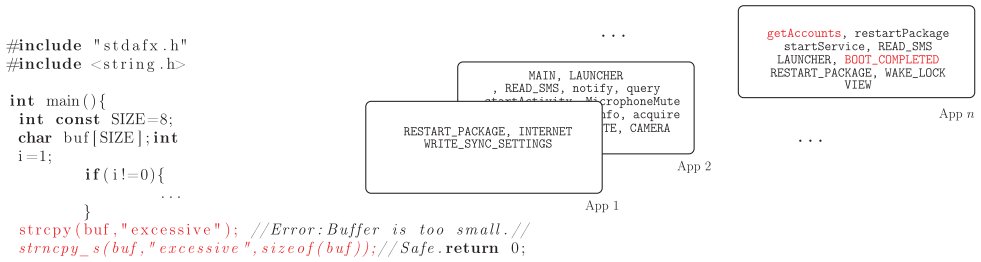

Fig. 2. Code based approaches to DDCS rely on only specific and atomic elements. ML methods look for small structures in the data opposed to the requirement of achieving classification by grouping them together.

reliance. Furthermore increased approaches to visualization, adopting newer techniques [7], and the use of deep learning over shallow ML.

5.2. Deep knowledge of code

A formative data structure for Android malware and vulnerability discovery can be observed in the representation of separation and atomic fragments. Obtaining discriminative use of the data does not require the entire makeup of each sample, instead select elements.

Malware approaches consider matching similar behavioral clones by features [6], vector spaces [1], and code fragments of interest [5,23,54,61,91]. Matching solely on too specific features can be undone [63,72]. Software vulnerability detection is even more refined than malware detection, as the code segment of interest may be one or a few lines long. Those matching or filtering code of interest from the literature is presented in [39,56,57,60,70,82–86]. Fig. 2 presents the data structure appropriate for code based data driven areas. The location of specific, or given elements in combination is sought, rather than collaborative and collective means.

Retention of knowledge is measurable in two areas. Firstly the ability to train and test classifiers between periods of time, while still achieving acceptable performance, and also observing a considerable data drift window. Should the drift of data or the ability to train and test with disrupted time periods be significant, little knowledge can be reapplied. Approaches to time delay and drift were explored for [54], while [33] further defined the drifting of samples. What may be observed is that important knowledge is mostly captured at the cost of performance downgrades. Much of what makes an older malicious application malicious is still present and bound by programming statements/APIs and such in newer applications.

Vulnerability discovery showed that retention was useful for future release testing but vulnerability specific [10,65,69]. Applying knowledge of code representations was investigated across different projects [44,81] where representation were extrapolated. It persists as considerable knowledge retention given the appliance over the different project and not remaining vulnerability specific. The idea of drift in vulnerability detection is redundant. The search for a given vulnerability will persist as long as it remains before correction, as opposed to the evolution of trends of methods. This persistence might be the result of continual integration of application features, along with a lack of constant testing and feedback given project size. As an example, through updates and release cycles, application functionality is maintained to help support wide user bases, however the concept of *Extreme Programming* [97] could prove beneficial in reducing the amount of necessary code. Engagement with users provides functionality dialogue and explicit use examples for development; this process may help reduce large portions of unnecessary code, providing tailored functionality which best meets users bases, thus eliminating portions of code bases often used which may contain vulnerabilities.

5.3. Ground truth for learning

The ability to retain knowledge is beneficial in applying it to help label data. Contrasting results are seen between Android malware and vulnerability discovery. Early generations of malware detection approaches relied on manual observation [19,38], where future literature was able to provide automatic means [23,27,80], to tools like *VirusTotal* being used [1,54,89]. Opposing is the ability to label vulnerable code segments. The results of [10,39,43,56,57,60,69,70,82–86] sought manual efforts, but [65] used a software tool. Though a disconnect persists, this is inherent of the domain thus far. Efforts are focused for a given exploit, where each occasion can be considered unique. A method to do so may not be transferable, and code references change.

Contributions are determined in the detection of anomalies and zero-day discoveries. The discovery of zero-day exploits and vulnerabilities we contributed from some of the literature contained within. Android zero-days were found in [27,91,93], and vulnerabilities detected in [42,82–84,86]. While not a definitive guide, we provide a basic process guide for security specialists in applying ML to a security challenge, as shown in Fig. 3. The balance, quality and label correctness of data influence ML results significantly. Validation and verification should be conducted to overcome some of the limitations. Furthermore, the security hypothesis needs to best match the problem and the associated data.

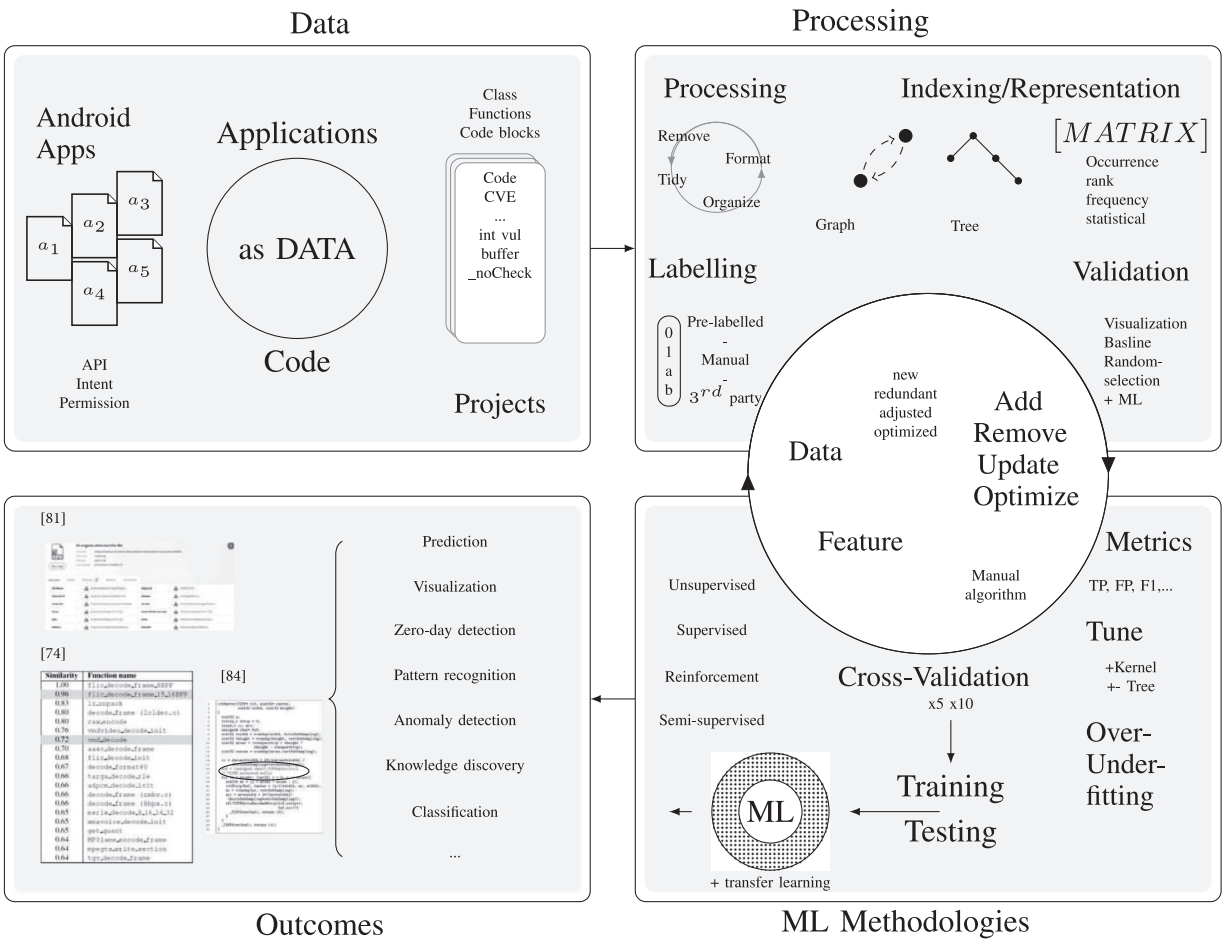

Fig. 3. Shared Data driven processes: Many domains that are data driven share similarities, particularly security. Data assets exist in many forms, from applications to known vulnerable code for which processing is conducted to best represent the data for classification, learning or analysis. Given the availability of new data, or the removal or adjustment of existing, a cycle exists between organizing the data and the application of machine learning. This process allows for tailoring and achieving data outcomes.

6. Conclusion

We present a new view of DDCS and surveyed the specific DDCS research on cyber code analysis. The detailed survey of recent key literature about malware analysis and software vulnerability analysis demonstrates the key ideas, concepts and methodology of DDCS. During the survey, the reviewed papers are compared from different technical point of view to show the research trends and opportunities. Finally, we summarize the research challenges and future work to encourage new DDCS research and new technology.

References

[1] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, C. Siemens, Drebin: Effective and Explainable Detection of Android Malware in Your Pocket, in: Proc. NDSS 14, 2014.
[2] R. Coulter, L. Pan, J. Zhang, Y. Xiang, A Visualization-based Analysis on Classifying Android Malware, in: International Conference on Machine Learning for Cyber Security, Springer, 2019, pp. 304–319.
[3] Y. Bengio, A. Courville, P. Vincent, Representation learning: a review and new perspectives, IEEE Trans. Pattern Anal. Mach. Intell. 35 (2013) 1798–1828.
[4] C.M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.
[5] T. Bläsing, L. Batyuk, A.D. Schmidt, S.A. Camtepe, S. Albayrak, An Android Application Sandbox System for Suspicious Software Detection, in: Proc. 5th International Conference on Malicious and Unwanted Software, 2010, pp. 55–62.
[6] A. Bose, X. Hu, K.G. Shin, T. Park, Behavioral Detection of Malware on Mobile Handsets, in: Proc. MobiSys 08, 2008, pp. 225–238.
[7] M. Bostock, V. Ogievetsky, J. Heer, d^3: Data-driven documents, IEEE Trans. Vis. Comput. Graph. 17 (2011) 2301–2309.
[8] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W.C. Lau, M. Sun, R. Yang, K. Zhang, IoTFuzzer: Discovering Memory Corruptions in Iot through App-based Fuzzing, in: Proc. NDSS 18, 2018.
[9] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, K. Ren, Android HIV: A study of repackaging malware for evading machine-learning detection, IEEE Transactions on Information Forensics and Security 15 (2020) 987–1001.
[10] I. Chowdhury, M. Zulkernine, Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities, J. Syst. Archit. 57 (2011) 294–313.

[11] M. Conti, L.V. Mancini, R. Spolaor, N.V. Verde, Analyzing android encrypted network traffic to identify user actions, IEEE Trans. Inf. Forensics Secur. 11 (2016) 114–125.
[12] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, A. Barenghi, S. Zanero, F. Maggi, ShieldFS: a self-healing, ransomware-aware filesystem, Proc. ACCSA 16, ACM, 2016, pp. 336–347.
[13] R. Coulter, L. Pan, Intelligent agents defending for an iot world: areview, Comput. Secur. 73 (2018) 439–458.
[14] D. Ding, Q.-L. Han, X. Wang, X. Ge, A survey on model-based distributed control and filtering for industrial cyber-physical systems, IEEE Trans. Ind. Inf. 15 (2019) 2483–3499.
[15] F. Dong, H. Wang, L. Li, Y. Guo, T.F. Bissyandé, T. Liu, G. Xu, J. Klein, FraudDroid: Automated ad fraud detection for Android Apps, Proc. 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, 2018, pp. 257–268.
[16] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, J.A. Halderman, The Matter of Heartbleed, in: Proc. IMC 14, ACM, New York, NY, USA, 2014, pp. 475–488.
[17] W. Enck, P. Gilbert, B.G. Chun, L.P. Cox, J. Jung, P. McDaniel, A.N. Sheth, Taintdroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones, in: Proc. 9th USENIX Conference on Operating Systems Design and Implementation, 2010, pp. 1–6.
[18] W. Enck, D. Octeau, P. McDaniel, S. Chaudhuri, A Study of Android Application Security, in: Proc. USENIX Security 2011, USENIX Association, Berkeley, CA, USA, 2011, pp. 21:1–21:16.
[19] W. Enck, M. Ongtang, P. McDaniel, On Lightweight Mobile Phone Application Certification, in: Proc. CCS 09, ACM, New York, NY, USA, 2009, pp. 235–245.
[20] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M.S. Gaur, M. Conti, M. Rajarajan, Android security: a survey of issues, malware penetration, and defenses, IEEE Commun. Surv. Tutor. 17 (2015) 998–1022.
[21] A. Feizollah, N.B. Anuar, R. Salleh, A.W.A. Wahab, A review on feature selection in mobile malware detection, Digital Invest. 13 (2015) 22–37.
[22] A.P. Felt, M. Finifter, E. Chin, S. Hanna, D. Wagner, A Survey of Mobile Malware in the Wild, in: Proc. 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, 2011, pp. 3–14.
[23] H. Gascon, F. Yamaguchi, D. Arp, K. Rieck, Structural Detection of Android Malware Using Embedded Call Graphs, in: Proc. AISec 13, ACM, New York, NY, USA, 2013, pp. 45–54.
[24] X. Ge, F. Yang, Q.L. Han, Distributed networked control systems: a brief overview, Inf. Sci. 380 (2017) 117–131.
[25] S.M. Ghaffarian, H.R. Shahriari, Software vulnerability analysis and discovery using machine-learning and data-mining techniques: a survey, ACM Comput. Surv. 50 (2017) 56:1–56:36.
[26] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT press, 2016.
[27] M. Grace, Y. Zhou, Q. Zhang, S. Zou, X. Jiang, Riskranker: Scalable and Accurate Zero-day Android Malware Detection, in: Proc. MobiSys 12, ACM, New York, NY, USA, 2012, pp. 281–294.
[28] J. Han, J. Pei, M. Kamber, Data Mining: Concepts and Techniques, Elsevier, 2011.
[29] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation (3rd Edition), Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
[30] D.Y. Huang, M.M. Aliapoulios, V.G. Li, L. Invernizzi, E. Bursztein, K. McRoberts, J. Levin, K. Levchenko, A.C. Snoeren, D. McCoy, Tracking ransomware end-to-end, in: Proc. S&P, IEEE, 2018. 18618–631.
[31] J. Huang, C.X. Ling, Using AUC and accuracy in evaluating learning algorithms, IEEE Trans Knowl Data Eng 17 (2005) 299–310.
[32] G. James, D. Witten, T. Hastie, R. Tibshirani, An introduction to statistical learning, Springer, 2013. 112
[33] R. Jordaney, K. Sharad, S.K. Dash, Z. Wang, D. Papini, I. Nouretdinov, L. Cavallaro, Transcend: Detecting Concept Drift in Malware Classification Models, in: Proc. USENIX Security 17), USENIX Association, Vancouver, BC, 2017, pp. 625–642.
[34] S. Karnouskos, Stuxnet Worm Impact on Industrial Cyber-physical System Security, in: Proc. IECON 11, 2011, pp. 4490–4494.
[35] A. Karpatne, G. Atluri, J.H. Faghmous, M. Steinbach, A. Banerjee, A. Ganguly, S. Shekhar, N. Samatova, V. Kumar, Theory-guided data science: a new paradigm for scientific discovery from data, IEEE Trans. Knowl. Data Eng. 29 (2017) 2318–2331.
[36] G. Keppel, Design and Analysis: A Researcher's Handbook, Prentice-Hall, Inc, 1991.
[37] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, E. Kirda, Unveil: A Large-scale, Automated Approach to Detecting Ransomware, in: Proc. 25th USENIX Security 16), 2016, pp. 757–772.
[38] H. Kim, J. Smith, K.G. Shin, Detecting Energy-greedy Anomalies and Mobile Malware Variants, in: Proc. MobiSys 08, ACM, New York, NY, USA, 2008, pp. 239–252.
[39] H. Kim, S. Kim, S. Woo, H. Lee, H. Oh, VUDDY: a scalable approach for vulnerable code clone discovery, in: Proc. S&P 17, IEEE, 2017, pp. 595–614.
[40] R.M. Lee, M.J. Assante, T. Conway, Analysis of the cyber attack on the ukrainian power grid, Electricity Information Sharing and Analysis Center (E-ISAC) (2016) 1–29.
[41] L. Li, T.F. Bissyande, J. Klein, Rebooting research on detecting repackaged android apps: literature review and benchmark, IEEE Trans. Software Eng. (2019).
[42] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, X. Wang, Z. Deng, Y. Zhong, Vuldeepecker: A Deep Learning-based System for Vulnerability Detection, in: Proc. NDSS 18, 2018, pp. 1–15.
[43] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, Poster: Vulnerability Discovery with Function Representation Learning from Unlabeled Projects, in: Proc. CCS 17), ACM, New York, NY, USA, 2017, pp. 2539–2541.
[44] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, P. Montague, Cross-project transfer representation learning for vulnerable function discovery, IEEE Trans. Ind. Inf. 14 (2018) 3289–3297.
[45] M. Lindorfer, M. Neugschwandtner, C. Platzer, Marvin: Efficient and Comprehensive Mobile App Classification through Static and Dynamic Analysis, in: Proc. ACSAC 15, 2015, pp. 422–433.
[46] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V.V.D. Veen, C. Platzer, Andrubis–1,000,000 apps later: A view on current android malware behaviors, in: Proc. BADGERS 14, IEEE, 2014. 3–17
[47] L. Liu, O. De Vel, Q.-L. Han, J. Zhang, Y. Xiang, Detecting and preventing cyber insider threats: asurvey, IEEE Commun. Surv. Tutor. 20 (2018) 1397–1417.
[48] Q. Liu, P. Li, W. Zhao, W. Cai, S. Yu, V.C.M. Leung, A survey on security threats and defensive techniques of machine learning: a data driven view, IEEE Access 6 (2018) 12103–12117.
[49] S. Liu, Y. Wang, J. Zhang, C. Chen, Y. Xiang, Addressing the class imbalance problem in twitter spam detection using ensemble learning, Comput. Secur. 69 (2017) 35–49.
[50] S. Liu, J. Zhang, Y. Wang, W. Zhou, W. Xiang, O. De Vel, A data-driven attack against support vectors of svm, Proc. ASIACCS 18, ACM, 2018, pp. 723–734.
[51] S. Liu, J. Zhang, Y. Xiang, W. Zhou, Fuzzy-based information decomposition for incomplete and imbalanced data learning, IEEE Trans. Fuzzy Syst. 25 (2017) 1476–1490.
[52] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, arXiv preprint arXiv:1301.3781 (2013).
[53] T.M. Mitchell, Machine Learning, 1, McGraw-Hill, Inc., New York, NY, USA, 1997.
[54] A. Narayanan, M. Chandramohan, L. Chen, Y. Liu, Context-aware, adaptive, and scalable android malware detection through online learning, IEEE Trans. Emerg. Top.Comput. Intell. 1 (2017) 157–175.
[55] F.A. Narudin, A. Feizollah, N.B. Anuar, A. Gani, Evaluation of machine learning classifiers for mobile malware detection, Soft Comput. 20 (2016) 343–357.
[56] S. Neuhaus, T. Zimmermann, C. Holler, A. Zeller, Predicting vulnerable software components, in: Proc. CCS 07, 2007. 529–540
[57] V.H. Nguyen, L.M.S. Tran, Predicting vulnerable software components with dependency graphs, in: Proc. 6th International Workshop on Security Measurements and Metrics, ACM, 3:1–3:8.

[58] M. Parkour, Contagio mobile data repository, 2019, Accessed: 2019-05-28 (http://contagiominidump.blogspot.com/).
[59] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, L. Cavallaro, Tesseract: eliminating experimental bias in malware classification across space and time, arXiv Preprint arXiv:1807.07838 (2018).
[60] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, Y. Acar, VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits, in: Proc. CCS 15, ACM, 2015. Proc.CCS 15, 426–437
[61] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, S. Ioannidis, Rage against the virtual machine: Hindering dynamic analysis of android malware, in: Proc. EuroSec 14, ACM, New York, NY, USA, 2014. 5:1–5:6
[62] P. Kieseberg, J. Schantl, P. Frühwirt, E. Weippl, A. Holzinger, Witnesses for the doctor in the loop, Springer International Publishing, 2015. 369–378, Brain Informatics and Health, pages Cham
[63] V. Rastogi, Y. Chen, X. Jiang, Droidchameleon: Evaluating android anti-malware against transformation attacks, in: Proc. ASIACCS 13, ACM, New York, NY, USA, 2013. 329–334
[64] C. Rossow, C.J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, M.V. Steen, Prudent Practices for Designing Malware Experiments: Status Quo and Outlook, in: Proc. S&P 12, IEEE, 2012, pp. 65–79.
[65] R. Scandariato, J. Walden, A. Hovsepyan, W. Joosen, Predicting vulnerable software components via text mining, IEEE Trans. Softw. Eng. 40 (2014) 993–1006.
[66] M. Sebastián, R. Rivera, P. Otzias, J. Aballero, AVClass: A Tool for Massive Malware Labeling, in: F. Monrose, M. Dacier, G. Blanc, J. Garcia-Alfaro (Eds.), Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses, Springer International Publishing, 2016, pp. 230–253.
[67] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, Y. Weiss, Andromaly: a behavioral malware detection framework for android devices, J. Intell. Inf. Syst. 38 (2012) 161–190.
[68] H. Shahriar, M. Zulkernine, Mitigating program security vulnerabilities: approaches and challenges, ACM Comput. Surv. 44 (2012) 11:1–11:46.
[69] Y. Shin, A. Meneely, L. Williams, J.A. Osborne, Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities, IEEE Trans. Softw. Eng. 37 (2011) 772–787.
[70] Y. Shin, L. Williams, An empirical model to predict security vulnerabilities using code complexity metrics, in: Proc. 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ACM, 2008. 315–317
[71] R. von Solms, J. van Niekerk, From information security to cyber security, Computers & Security 38 (2013) 97–102. Cybercrime in the Digital Economy http://www.sciencedirect.com/science/article/pii/S0167404813000801.
[72] G. Suarez-Tangil, J.E. Tapiador, P. Peris-Lopez, J. Blasco, Dendroid: a text mining approach to analyzing and classifying code structures in android malware families, Expert Syst Appl 41 (2014) 1104–1117.
[73] Sufatrio, D.J.J. Tan, T.W. Chua, V.L.L. Thing, Securing android: a survey, taxonomy, and challenges, ACM Comput. Surv. 47 (2015) 58:1–58:45.
[74] N. Sun, J. Zhang, P. Rimba, S. Gao, L.Y. Zhang, Y. Xiang, Data-driven cybersecurity incident prediction and discovery: a survey, IEEE Commun. Surv. Tutor. 21 (2019) 1744–1772.
[75] K. Tam, A. Feizollah, N.B. Anuar, R. Salleh, L. Cavallaro, The evolution of android malware and android analysis techniques, ACM Comput. Surv. 49 (2017) 76:1–76:41.
[76] VirusTotal, 2019, Accessed: 2019-05-28 https://www.virustotal.com.
[77] T. Vollmer, M. Manic, Cyber-physical system security with deceptive virtual hosts for industrial control networks, IEEE Trans. Ind. Inf. 10 (2014) 1337–1347.
[78] F. Wei, U. Li, A. Roy, I. Ou, W. Zhou, Deep Ground Truth Analysis of Current Android Malware, in: M. Polychronakis, M. Meier (Eds.), Proc. Detection of Intrusions and Malware, and Vulnerability Assessment, Springer International Publishing, 2017, pp. 252–276.
[79] G. Wei, J. Shao, Y. Xiang, P. Zhu, R. Lu, Obtain confidentiality or/and authenticity in big data by ID-based generalized signcryption, Inf. Sci. (Ny) 318 (2015) 111–122.
[80] D.J. Wu, C.H. Mao, T.E. Wei, H.M. Lee, K.P. Wu, Droidmat: Android Malware Detection through Manifest and API Calls Tracing, in: Proc. 7th Asia Joint Conference on Information Security, 2012, pp. 62–69.
[81] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, D. Song, Neural Network-based Graph Embedding for Cross-platform Binary Code Similarity Detection, in: Proc. CCS 17, ACM, 2017, pp. 363–376.
[82] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and Discovering Vulnerabilities with Code Property Graphs, in: Proc. S&P 14, 2014, pp. 590–604.
[83] F. Yamaguchi, F. Lindner, K. Rieck, Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning, in: Proc. WOOT 11, 2011. 13:1–13:10
[84] F. Yamaguchi, M. Lottmann, K. Rieck, Generalized vulnerability extrapolation using abstract syntax trees, in: Proc. ACSAC 12, ACM, 2012. 359–368
[85] F. Yamaguchi, A. Maier, H. Gascon, K. Rieck, Automatic inference of search patterns for taint-style vulnerabilities, in: Proc. S&P 15, IEEE, 2015. 797–812
[86] F. Yamaguchi, C. Wressnegger, H. Gascon, K. Rieck, Chucky: Exposing missing checks in source code for vulnerability discovery, in: Proc. CCS 13, ACM, 2013. 499–510
[87] X. Yu, Y. Xue, Smart grids: a cyber-physical systems perspective, Proc. of the IEEE 104 (2016) 1058–1070.
[88] Z. Yuan, Y. Lu, Y. Xue, Droiddetector: android malware characterization and detection using deep learning, Tsinghua Sci. Technol. 21 (2016) 114–123.
[89] M. Zhang, Y. Duan, H. Yin, Z. Zhao, Semantics-aware android malware classification using weighted contextual API dependency graphs, in: Proc. CCS 14, ACM, New York, NY, USA, 2014. 1105–1116
[90] B. Zhao, W. Kou, H. Li, L. Dang, J. Zhang, Effective watermarking scheme in the encrypted domain for buyer-seller watermarking protocol, Inf. Sci. (Ny) 180 (2010) 4672–4684.
[91] M. Zheng, M. Sun, J.C.S. Lui, Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware, in: Proc. 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 2013. 163–171
[92] Y. Zhou, X. Jiang, Dissecting android malware: Characterization and evolution, in: Proc. S&P 12, 2012. 95–109
[93] Y. Zhou, Z. Wang, W. Zhou, X. Jiang, Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets, in: Proc. NDSS 12, 2012.
[94] X. Zhu, T. Dumitraș, Featuresmith: Automatically engineering features for malware detection by mining the security literature, in: Proc. CCS 2016, 2016, pp. 767–778. ACM
[95] A. Holzinger, P. Kieseberg, E. Weippl, A.M. Tjoa, Current advances, trends and challenges of machine learning and knowledge extraction: From machine learning to explainable AI, Springer International Publishing, 2018. Machine Learning and Knowledge Extraction, pages 1–8, Cham
[96] R. Coulter, Q.-L. Han, L. Pan, J. Zhang, Y. Xiang, Data-driven cyber security in perspective–intelligent traffic analysis, 2019, IEEE Transactions on Cybernetics, 10.1109/TCYB.2019.2940940
[97] A. Holzinger, M. Errath, G. Searle, B. Thurnher, W. Slany, From extreme programming and usability engineering to extreme usability in software engineering education (xp+ue /spl rarr/ xu), Proc. 29th Annual International Computer Software and Applications Conference (COMPSAC'05) 2 (2005) 169–172. 1.