DEVELOPMENT STRATEGY AND MANAGEMENT OF AI-BASED VULNERABILITY DETECTION APPLICATIONS IN ENTERPRISE SOFTWARE ENVIRONMENT

Stefan Behfar
Geneva School of Business Administration, stefan-kambiz.behfar@hesge.ch

DEVELOPMENT STRATEGY AND MANAGEMENT OF AI-BASED VULNERABILITY DETECTION APPLICATIONS IN ENTERPRISE SOFTWARE ENVIRONMENT

Research in Progress

Stefan Kambiz Behfar
Department of Information Systems, Geneva School of Business Administration (HES-SO Genève), Geneva, Switzerland, stefan-kambiz.behfar@hesge.ch

Abstract

Industries are now struggling with high level of security-risk vulnerabilities in their software environment which mainly originate from open-source dependencies. Industries' percentage of open source in codebases is about 54% whereas ones with high security risks is about 30% (Synopsys 2018). While there are existing solutions for application security analysis, these typically only detect a limited subset of possible errors based on pre-defined rules. With the availability of open-source vulnerability resources, it is now possible to use data-driven techniques to discover vulnerabilities. Although there are a few AI-based solutions available, but there are some associated challenges: 1) use of artificial intelligence for application security (AppSec) towards vulnerability detection has been very limited and definitely not industry oriented, 2) the strategy to develop, use and manage such AppSec products in enterprises have not been investigated; therefore cybersecurity firms do not use even limited existing solutions. In this study, we aim to address these challenges with some strategies to develop such AppSec, their use management and economic values in enterprise environment.

Keywords: software vulnerability, vulnerability detection, development strategy, AppSec management.

1 Introduction

Small and medium-sized businesses are uniquely susceptible to cybersecurity threats because they often lack the resources of larger enterprises to invest in more sophisticated and comprehensive solutions. If the devastating effects of a cyber incident resulted in the theft of the target company's information, it would be manageable. However, a closer examination will tell you that if a company's information is stolen, so is the information of their customers, vendors, and employees. For every high profile, sophisticated attack there are dozens of smaller ones that we just don't hear about. In fact, 43 percent of online attacks are now aimed at small businesses, a favorite target of high-tech villains (Cerny, 2021).

In a digital world, security is a must. Cyberattacks like WannaCry have wreaked terrible effects on unprepared citizens, businesses, and organizations, putting their life and operations in jeopardy (Zhu et al. 2011). Although protecting our personal data on the internet is quite crucial, the exponential growth in the number of connected devices increased the complexity of cyber infrastructure, resulting in an increase in the number of vulnerable devices (Philip et al. 2014). Phishing, password attacks, downloads via hyperlinks, virus attacks, attacks originated from software source code vulnerabilities are all examples of possible cyberattacks (Paliwal 2016). Now, the internet has become an integral aspect of our everyday human life, and as a matter of fact, artificial intelligence security solutions have proved to maximize the efficiency and minimized the need for additional security solutions by leveraging the AI predictive and defensive capabilities (Thuraisingham 2020, Behfar and Hosseinpour 2022).

Most organizations manage hundreds to thousands of software components, ranging from mobile apps to cloudbasedsystems to legacy systems running on-premises. That software is typically a mixture of commercial off-the-shelf packages and custom-built codebases, both of which are increasingly made up of open source components. 99% of the codebases the Black-Duck Audit Services team audited in 2019 contained open source. If your organization builds or simply uses software, you can assume that software will contain open-source. Whether you are a member of an IT, development, operations, or security team, if you don't have policies in place for identifying and patching known issues with the open-source components thatyou're using, you're not really doing your job. The open-source community usually issues small updates at a much faster pace than the average commercial software vendor. When these updates contain security updates, companies need to have a strategy to adopt them rapidly. But because open source updates need to be "pulled" by users, an alarming number of companies consuming open source components don't apply the patches they need, opening their business to the risk of attack and applications to potential exploits. This makes it even more significant to find ways in order to detect and even predict code vulnerabilities.

Nowadays, enterprises search in Common Vulnerabilities and Exposures (CVE) databases for vulnerability relevant information; however open source vulnerabilities are very dispersed among very different sources such as OWASP, Node Security Project (NSP), RetireJS, OSSindex, Bulndler-audit, Hakiri, Snyk, Gemnasium, Sonatype 's Nexus, Protecode, among others. There are also vulnerability databases such as National vulnerability database (NVD), Github advisory database, Synopsys/Blackduck Hub, HCL AppScan, more. In fact, software dependencies constitute largest security vulnerabilities, while it was previously assumed that most risks come from web applications. There are studies which compare commercial and open source software and claim that one is more secure than the other, however what is important is that to secure a code it requires thorough inspection, dynamic security scanning and penetration testing and others, otherwise the code is not secure. Prior studies in vulnerability detection have focused on the following main topics, as shown figure below, while we focus on software metrics and features detected via machine learning, colored green.

As shown in *Figure 1*, vulnerability detection could be performed via static or dynamic analysis scanning software metrics. For a long time, the most commonly used features were found outside the source code itself, in the form of software and developer metric. Those include size of the code, cyclomatic complexity, code churn, developer activity, coupling, number of dependencies or legacy metrics (Morrison et al. 2015). Such metrics have been universally used as features for building fault prediction model. (Wartschinski 2019).

In fact, there are numerous ways to use artificial intelligence, depending on the language being used, the dataset's size and origin, the process used to create labels, the level of detail used in the analysis, the type of machine learning model being used (such as deep learning, convolutional neural networks, recurrent neural networks, support vector machines, random forests, or others), and finally, whether the model can be used to predict outcomes across multiple projects, see Russell et al. (2018) and Li et al. (2018). Further, Pang et al. (2015) identified entire Java classes as susceptible or not by using labels from an online database and a combination of feature selection.

Although there are already some existing application security (AppSec) products, investigating code vulnerabilities either via software metrics, similarity or pattern analysis, such as JFrog Advanced Security which claims to provide software composition analysis powered by JFrog Xray, container contextual analysis, IaC security, secrets detection, and more, there are however many challenges:

- Not only they provide inadequate accuracy, but also are constrained with both their training model and single repository. Therefore, we aim to address all the challenges of producing an AI-based solution to detect vulnerabilities based on open-source software dependencies.
- Although there are some AppSec products in the market, such as Blackduck, Acunetix, Deepcode, etc. however most companies do not often use them, because the strategy to manage such AppSec products have not been clearly defined.

- Trust in products has not been built, which is the main factor to push entreprises to use AppSec in their entire supply chain, although some companies have rendered DevSecOps solution attempting to unify developers and security teams to protect their complete software development pipeline.

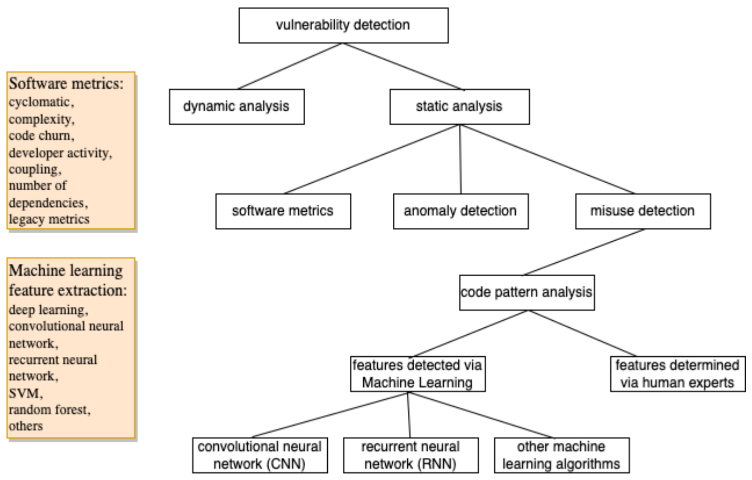

vulnerability detection
dynamic analysis
static analysis
software metrics
anomaly detection
misuse detection
code pattern analysis
features detected via Machine Learning
features determined via human experts
convolutional neural network (CNN)
recurrent neural network (RNN)
other machine learning algorithms
Software metrics: cyclomatic, complexity, code churn, developer activity, coupling, number of dependencies, legacy metrics
Machine learning feature extraction: deep learning, convolutional neural network, recurrent neural network, SVM, random forest, others

Figure 1. code vulnerability detection methodologies

2 Model

2.1 Research questions

In defining the strategies to develop and manage use of an AppSec to detect vulnerabilities, one needs first to find the challenges that the cybersecurity entreprises are now facing. One major challenge is that the scanning tools detect huge volume of vulnerabilities and entreprises have no facility to leverage from. Therefore, it would be necessary to find factors which could lead to vulnerabilities. These factors help in narrowing down search for vulnerabilities. The goal is not only to develop an AI-based vulnerability detection tool, but also to find the influential factors on project vulnerabilities, where:

RQ1. Does network structure impact on project vulnerability in a dependency network?

Since the vulnerabilities are expected to be spread from open-source packages into entreprise software environment, one needs to find out which packages are more more vulnerable to attacks (Behfar 2023). Seeing that there are some factors influencing package vulnerability such as number of contributors, project repository size, and package age. Although we already know that the greater the number of project contributors, the higher the probability of vulnerability discovery in a project, one could explore a) package vulnerability, with focus on the behaviors which are less predictable such as number of open issues and number of forks, b) whether and how versioning system influences on vulnerability, and whether newer package versions are less vulnerable compared to the older versions (Behfar 2023).

RQ2. Do number of open issues and number of forks have influence on package vulnerability ?

RQ3. Are newer package versions less vulnerable than older package versions ?

There are various number of vulnerabilities and cannot all be easily mapped to OSS projects. In fact, we can only say whether a package version has vulnerability or not based on known vulnerabilities. There are multiple factors influencing OSS package vulnerability. Consider that packages A and B depend on different versions of C, but only one of the versions of C depends on D. Therefore, any version dependency modeling which assumes that packages A and B are dependent on package D is not true. It can be explored via a version dependency network that how A and B depend on which version of C, and how C depends on which version of D, see Figure 2. Aggregated network do not simply answer to this question, and over-count all dependency measures, see Kikas et al. (2017). As more up-to-date versions for dependent components are released, dependency requirements also change to the latest versions. In fact, it would be interesting to see whether by defining some vulnerability metric such as vulnerability density (#actual vulnerabilities), one can predict detection of unpublished vulnerabilities in later package versions based on prior detected vulnerabilities.

RQ4. Can one determine the probability of vulnerability detection in latest package versions based on prior detected vulnerabilities ?

Some artificial intelligence based solutions to detect vulnerabilities have been explored in the literature (as referenced in the introduction), but has been inefficient due to:

a. lack of enough vulnerability sources, as they mainly use National Vulnerability Database (NVD) with limited published vulnerabilities, which affects accuracy of AI-based classification models.

b. these typically only detect a limited subset of possible errors based on pre-defined rules, and do not provide risk assessment, score or mitigation plan.

c. the built solutions produce many alerts and false alarms, and there is no sorting possibilities.

therefore,

RQ5. Can one create an application security solution with high accuracy and precision ?

We think that making a right strategy to building an innovative application security (AppSec) product is a solution to afore-mentioned research questions, however there are also some associated challenges:

- development of such AI-based application security product should be more industry oriented, and solves the existing entreprises' challenges. Therefore, such strategy should be well defined.
- the strategy to use, manage and scope for such AppSec in entreprises should be well defined; now cybersecurity firms do not use even limited existing solutions.
- the economic value of such AppSec product such as fullfillment costs and increase efficiency has to be better explained, which encourages the entreprises to use such products.

2.2 Dependency network

Not only we are interested in detecting vulnerabilities in entreprise software environment spread from open-source packages with vulnerabilities but also interested in knowing how rate of vulnerabilities in a software grows in time; is the network structure, dependent project version number, open issues, more play a role? Therefore, one could investigate a possible correlation between network structure and vulnerabilities found in software environment. We expect that having high package indegree implies well-known open-source project which leads to a higher number of contributors, open issues, and increase probability of being attacked. It is widely known that project health is associated with number of developers working on the project, which in fact shows how much the project has attracted attention from the community. This also makes sense that when number of developers increases over time, number of open issues should decrease over time. When both number of developers and number of open issues decrease over time, this translates to the fact that the project has decreasing popularity with negative trend of its health. This leads to higher vulnerability or potentially being attacked. We define packages as a code which is reusable in other packages or applications (we use packages as reusable codes whereas applications are not reusable; both are counted as projects). A dependency network is composed of nodes and links where nodes are denoted by packages and links are denoted by dependencies, see *Figure 2* for a sample dependency network. If package A depends on package B; it means that A is a dependent on B. A project has direct dependency when it directly includes the package, otherwise the project has transitive dependency. We identify various dependencies including direct and indirect ones.

1. A dependency which cannot be resolved until runtime is so called a runtime dependency. Runtime dependencies are important as an application cannot be installed or operate before first installing all its runtime dependencies. On the other hand, development dependencies are needed when one wants to make modifications.

2. Direct dependency refers to any package or application input directly exported by a library, API, or another component, whereas transitive dependencies are ones which are referenced by the component. So, in a sense, transitive dependency implies that if A depends on B and B depends on C, then A depends on both B and C.

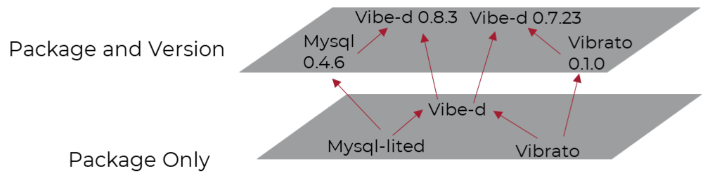


Figure 2. Illustration of a sample dependency network model

2.3 Vulnerability detection in entreprise software codes

In fact, code inspection is a key process for detecting vulnerabilities, and a necessary prerequisite to fixing them. Even code that is secure at some point in time often does not stay like that for very long. Any code change in a project can possibly alter the attack surface or introduce a security vulnerability. Therefore, constant scrupulous code reviews are necessary to catch any such flaw. But inspecting code manually is a tedious, time-consuming process that requires substantial knowledge in the area of security, mostly because modern software systems are becoming increasingly complex and interconnected, and problems can lie in very innocuous looking segments of code. There are already some studies investigating code vulnerabilities either via software metrics, similarity or pattern analysis, shown in *Figure 1*. Not only they provide inadequate accuracy, but also are constrained with both their training model and single software repository. Considering the current shortcomings, one can produce an innovative AI-based solution benefitting from both areas of software metrics and code pattern analysis in order to detect source code vulnerabilities based on the trained machine learning algorithms and code dependencies, and deliver a risk mitigation plan. The activities towards this goal include:

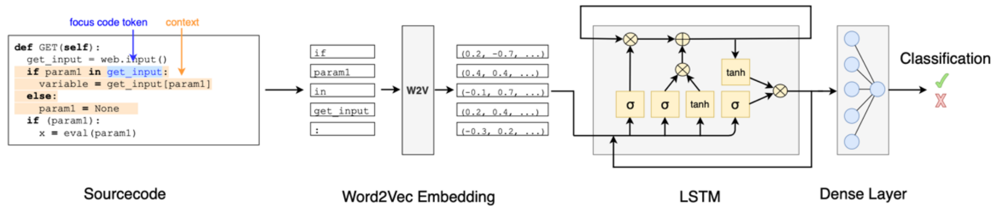
1. Design a common format dependency checker utility for
   - various programming languages and their package managers (e.g. PIP, NPM, Composer)
   - various package management systems (for example RPM, APT)
   - recognition of implications environmentals (e.g. development vs. production dependencies)
   - transitive dependencies (for example, dlopen(), try/catch)

2. Develop the utility and validate outcomes against
   - native package manager installation results
   - OWASP Dependency-Check
   - GitHub's Insights Graphs
   - existing software distribution package managers

3. In vulnerable code pattern analysis, vulnerable code segments are analyzed with data-mining and machine-learning techniques to extract their typical features. Those features represent patterns, which can then be applied onto new code segments to find vulnerabilities. Most of the works in this area gather a large dataset, processes it to extract feature vectors, then uses machine-learning algorithms.

4. Learn features without initial assumptions; therefore, it is independent of manually designed features that are the main limitation of the current static analysis tools.

5. Apply machine learning models for vulnerability detection and prediction, tokenize using lexical/word vector model, and use convolutional feature extraction with n filters to capture all token embedded dependencies, train the neural network with labeled vulnerabilities, and cross entropy as a loss function. For a sample of the pipeline, how it is performed, see *Figure 3*.



```
def GET(self):
  get_input = web.input()
  if param1 in get_input:
    variable = get_input[param1]
  else:
    param1 = None
  if (param1):
    x = eval(param1)
```

Figure 3. Illustration of vulnerability detection pipeline (Wartschinski 2019).

3 Analysis

3.1 Where do vulnerabilities' data and repositories come from ?

We have not indeed found a paper in the literature or an industrial application which explores code vulnerability prediction based on OSS dependency network with unconstrained source repositories, in that a large dataset of source code is collected from GitHub and the data is labeled according to information from the commit context. Since the data consists of security-related fixes, the sections of code that were changed or deleted in such a commit can be labeled as vulnerable, and the version after the fix, as well as all the data surrounding the changed part, is labeled as non-vulnerable. Of course, there are cases in which a fix does not actually solve a problem, or there are several vulnerabilities at the same time, or even a new vulnerability is introduced. That is why we also consider other vulnerability repositories in our vulnerability data collection, so that we can use them for labeling as well. Therefore, several different types of vulnerabilities are taken into consideration from NVD database among others. These data are used for labeling project components with their vulnerabilities. Vulnerabilities are not just detected on the file level, but in specific positions within the code, which is presumably more useful for developers. These are obtained by the information from github commits. The goal is to learn features without initial assumptions. Therefore, it is independent of manually designed features that are the main limitation of the static analysis tools. In general, the vulnerability sources include:

- vulnerability databases and repositories (ones applicable)
  - NVD, Github Advisory database, Synopsis/Blackduck Hub, HCL AppScan
  - OWASP, Node Security Project (NSP), RetireJS, OSSindex, Bulndler-audit, Hakiri, Snyk, Gemnasium, Sonatype 's Nexus, Protecode, among others
  - open-source package managers
- github commits by various developers
  - vulnerabilities are detected via differences of github commits

In order to collect data from github, one can find a large amount of commits that fix a security problem. Since the goal is to cover a range of different vulnerabilities, many examples for each of those vulnerability types are required. Since the act of fixing a flaw indicates the presence of the flaw in the first place and serves as the foundation for later data labeling, commits are actually the major subject of interest. Only a limited amount of calls can be made using the Github search API, and each request can only return up to 1000 results. Filters cannot be used in the search API, so it is not feasible to restrict results to only one programming language, in contrast to the standard search that users can use. As a result, this filtering must be done manually by further reducing the results once they have been received and identifying the few relevant and valuable ones among them.

In the next step, downloads of the diff files are made. A diff is only a text file that contains a commit's changes. It includes three lines of code before and after the altered lines as well as some meta data (such as the filename and the changed line's number). A commit on Github can simultaneously change numerous separate files. It is possible to download the diff for a commit URL with one straightforward HTTP request. This is a far simpler method than to clone the entire repository and pick only specific files from a specific period in a project's history, which at this point doesn't appear to be practical due to the size of the dataset and computational and time constraints. A large set of code diffs that can be utilized to reconstruct the pertinent lines of code before and after the patch are resulted from the previous phase. The diff from Github just includes the lines that have changed in addition to the three lines before and three lines after each change. The goal is to label the first version "vulnerable" and the second version "not vulnerable," which really yields fairly pleasing outcomes. The classifier that gains knowledge from the training set may accurately classify examples in the validation set and determine whether they fall under the fixed or vulnerable category (Wartschinski 2019).

3.2 Results and Discussion

The goals of the software vulnerability project are to both enhance vulnerability detection and risk assessment capabilities in software developmenüt projects and deployed software, and increase the efficacy. The application security (AppSec) solution is expected to deliver an unequaled capability to predict software security issues and assess risks that would otherwise require an extensive discovery and review process and more likely than not yield incomplete results. While typical software development project may include up to hundreds of third-party open-source components, it can be immeasurably difficult for developers to accurately and completely recognize and mitigate security issues and risks, and cumbersome for security researchers to find all candidate vectors.

Audit Projects discover, review and analyze infrastructure for potential vulnerabilities and demonstrate paths to exploitation of vulnerabilities. These activities include asset discovery, attack surface and vector discovery and ultimately usually successful attempts to exploit vectors. As a matter of fact, when code needs to be analyzed to find vulnerabilities, currently no analysis tools available are considered particularly useful. The frequency of false positives from them stifle the Audit Project's efforts, and the types of exploit vectors that end up being used are not covered. An innovative AppSec allows to assess vulnerabilities and risk in more complex code usages, throughout its dependency chains. The AppSec solution is expected to integrate in very divergent profiles and scenarios, and lead to different usages:

- Audit & Pentesting, Security Research, Threat Hunting
- Engagements in offensive and defensive security projects across various sectors including financial, energy, transportation.
- Offensive Security Auditor projects (Red Teaming) consists, to a significant degree, of manual evaluation of software security issues
- Code & Implementation Audit capabilities

The actionable benefits to AppSec are simply:

- Supplement certified offensive security specialists with the tool they require to improve their accuracy and coverage
- Supplement automated security vulnerability predictive and prescriptive with the necessary able-bodied staff to bring to Proof-of-Concept exploitation, and resolution
- Enable predictive & prescriptive capabilities on software analysis throughout a dependency chain
- Provide a risk assessment for or against the use of certain software, patterns or otherwise

The strategies to manage AppSec in entreprise software environment includes goals: 1) application and software security development, 2) reducing the uncertainties and volatility of variances in dependency chains, and 3) deploying AppSec as an integral part of in-house product development workflows:

1. AppSec is able to detect and predict software vulnerabilities through the collection of software and determining its variable form dependency chains. The dependency tracking utility is a critical component to success as it enables AppSec and further objectives. It provides the following functionalities:

- From an application's source code, determine, where applicable, the relevant dependency manager technologies, for example:
    - For NodeJS, this would be package(-lock).json for npm
    - For PHP, this would be composer.json for composer
    - For Python, this could be requirements(-dev).txt or setup.py, or undefined
    - For C, C++ and other languages, this could be CMake, FNU configure and build system, or undefined
- Determine the availability of pipeline staging for said dependency manager, for example:
    - For NodeJS and PHP, this could be development or production
    - For Python, this could be any number of text files that list dependencies
- Determine build- and run-time options and flags, and their impact on dependency chains, as:
    - Some software may have options such as "build with OpenSSL", or "Build with GnuTLS", or "build without any SSL", etc
- Mine the source code to find
    - Transitive dependencies, such as dlopen(), or conditional import
    - Undocumented dependencies
- Determine the dependency chain applied in linux distribution package manager solutions, where packaged software distribution is available.
- Export and store the outcomes to formats that are suitable for AppSec

2. In the context of AppSec, the dependency tracking utility vastly reduces the uncertainties and volatility of variances in dependency chains that AppSec will examine. Together with Offensive Security specialists, one can implement on-demand security testing for any application:

- From source code management repository/repositories, through the upload of a source archive.
- For existing (web-based) targets, the objective is to clearly position AppSec and assess the nature of security issues found through AppSec, and compare them with code audit and penetration testing project efforts and outcomes.

3. AppSec could be deploied as an integral part of in-house product development workflows, showcasing AppSec against different software development projects, with thousands of external, third party components and dependencies, including various high-profile Open Source projects such as MISP (PHP), Laravel (PHP), MediaSoup (NodeJS), Django (Python), Vue (JS), and many more.

4 Conclusion

In this study, we attempted to explore not only strategies to development of AI-based vulnerability detection applications in entreprise software environment, but also use management of such AppSec products. The goal of AppSec is to reduce fullfillment costs and increase efficiency. Automated discovery of security issues in applications will vastly reduce the effort needed on Audit and Penetration Testing, provided especially for custom, in-house or commissioned software development projects, and at least a doubling of exploit paths. With the inclusion of AppSec in such projects, should find the fullfillment costs reduced significantly before any exploitation path is found and successfully executed.

We had major contributions in 1) development strategy of such AI-based AppSec products with functionalities against different dependency manager technologies, 2) deploying AppSec as an integral part of in-house product development workflows, 3) describing all actionable benefits to AppSec in enterprises including predictive & prescriptive capabilities, risk assessment against the use of certain software, patterns or otherwise.

References

Behfar, S.K. (2023). *Exogenous and Endogenous Factors Leading to OSS Vulnerability: Study on Version Dependency Network.* Lecture Notes in Networks and Systems, Springer, pp 539–557.

Behfar, S.K. and Hosseinpour, M. (2022). *Cascading impact of cyberattacks on multi-layer social networks.* In Proceedings of 14th Mediterranean Conference on Information Systems (MCIS), Italy.

Cerny D. (2021). *The Cascading Effects of a Cyber Incident.* URL: https://www.sharp-sbs.com/simply-smarter-blog/the-cascading-effects-of-a-cyber-incident

Greenfield, V.A. and Pauli L. (2013). *A framework to assess the harm of crim*es. Br J Crimi., vol. 53, pp. 864–885.

Grieco G., Grinblat G.L., Uzal L., Rawat S., Feist J., and Mounier L. (2016). *Toward large-scale vulnerability discovery using machine learning.* In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, pages 85–96.

Kazienko P., Musial K., Kukla E., Kajdanowicz T., Bródka P. (2011a). *Multidimensional Social Network: Model and Analysis.* ICCCI 2011, LNCS, Springer, pp. 378-387.

Kazienko P., Kukla E., Musial K., Kajdanowicz T., Bródka P., Gaworecki J. (2011b). *A Generic Model for Multidimensional Temporal Social Network.* ICeND2011, CCIS, Springer, pp. 1-14.

Kikas, R., Gousios, G., Dumas, M., and Pfahl, D. (2017). *Structure and Evolution of Package Dependency Networks.* In proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories.

Kshetri N. (2006). *The simple economics of cybercrimes.* IEEE Secur Priv, 4, pp. 33–39.

Lee K. M., Min B., Goh K-I. (2015). *Towards real-world complexity: an introduction to multiplex networks.* European Physical Journal B, 88, 2.

Li Z., Zou D., Xu S., Ou X., Jin H., Wang S., Deng Z., and Zhong Y. (2018). *Vuldeepecker: A deep learning-based system for vulnerability detection.* arXiv preprint arXiv:1801.01681.

Morrison, P., Herzig, K., Murphy, B., and Williams, L. (2015). *Challenges with applying vulnerability prediction models.* In Proceedings of the Symposium and Bootcamp on the Science of Security 2015.

Neuhaus S., Zimmermann T., Holler C., and Zeller A. (2007). *Predicting vulnerable software components.* In ACM Conference on computer and communications security, pages 529–540, 2007.

Paliwal P. (2016). *Cyber Crime.* Nations Congress on the Prevention of Crime and Treatment of Offenders.

Philip C.L., Chen Q., and Zhang C.Y. (2014). *Data-intensive applications challenges techniques and technologies: A survey on big data.* Information Sciences, vol. 275, pp. 314-347.

Russell R., Kim L., Hamilton L., Lazovich T., Harer J., Ozdemir O., Ellingwood P., and McConley M. (2018). *Automated vulnerability detection in source code using deep representation learning.* 17th IEEE International Conference on Machine Learning and Applications (ICMLA), pages 757–762.

Shin Y., Meneely A., Williams L., and Osborne J.A. (2010). *Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities.* IEEE Transactions on Software Engineering, 37(6):772–787.

Simopoulos, R. (2021). *The SolarWinds Breach and the Cascading Effects of a Cyber Incident.* URL: https://commercialintegrator.com/networks/information_technology/solarwinds-breach-cyber-incident/

Synopsys report (2018). URL : https://www.synopsys.com/

Thuraisingham B.M. (2020). *Can AI be for Good in the Midst of Security Attacks and Privacy Violations?*. Proceedings ACM CODASPY.

Wartschinski L. (2019). *Detecting Software Vulnerabilities with Deep Learning.* Humboldt-Universität zu Berlin Mathematisch-Naturwissenschaftliche Fakultät Institut für Informatik.

Yamaguchi F., Lindner F. and Rieck K. (2011). *Vulnerability ex- trapolation: assisted discovery of vulnerabilities using machine learning.* In Proceedings of the 5th USENIX conference on Offensive technologies.

Zhu B., Joseph A., and Sastry S. (2011). *A taxonomy of cyber-attacks on SCADA systems.* International conference on internet of things and 4th international conference on cyber physical and social computing, pp. 380-388.