

Vdetekt: Synchronous Vulnerability Detection Plugin for Android Studio, a Step Towards a Secure Development Environment

Adekemi O Amoo

Department of Computer Science & Engineering,
Obafemi Awolowo University, Ile-Ife, Nigeria
aamoo@oauife.edu.ng

Temitayo O Oyegoke

Department of Computer Science &
Engineering, Obafemi Awolowo University,
Ile-Ife, Nigeria
tooyegoke@oauife.edu.ng

Mary T Onifade

Department of Computer Science &
Engineering, Obafemi Awolowo University,
Ile-Ife, Nigeria
tmonifade@oauife.edu.ng

Damilare E Bakare

MosaiQ Labs, London, United Kingdom
d.bakare@mosaiqlabs.com

Bukunmi E Ola

Department of Computer Science &
Engineering, Obafemi Awolowo University,
Ile-Ife, Nigeria
ebukunmiola@gmail.com

Abstract—As mobile app development becomes increasingly reliant on open source and third-party libraries, the risk of introducing security vulnerabilities during development becomes a major concern, most especially in the Android Studio environment. While automated tools can help reduce these vulnerabilities, they often lack real-time detection capabilities, are not user-friendly, and fail to integrate into the developer's workflows seamlessly. This paper presents a synchronous vulnerability detection system for Android Studio that enables real-time code analysis and security checks within the development environment. The plugin was implemented using Kotlin, IntelliJ Plugin Devkit, and Android Studio. The empirical evaluation shows a 100% success rate in detecting SQL injection, a 90% success rate for identifying hardcoded secrets, and a 72% detection rate for insecure data storage. This study contributes a practical and lightweight plugin that significantly enhances the overall security for Kotlin-based Android app development, reinforcing the need for early-stage vulnerability detection in the mobile application development lifecycle.

Keywords—Android App Development, Android Security, Kotlin, Vulnerability Detection, Secure Mobile Development, Android Studio.

I. INTRODUCTION

Vulnerabilities in source code are loopholes that malicious actors can exploit to compromise the application's security. These vulnerabilities are often targeted through scripts or commands that overturn the software's intended behaviour, leading to significant security breaches

This concern is particularly pressing in the context of mobile applications, which have become increasingly integral to modern life, offering functionalities like entertainment, communication, and productivity [11]. As their usage grows, so does the volume of sensitive data they handle, making them a key target for attackers [2]. To protect against common security threats such as insecure data storage, weak hosting controls, and insufficient

cryptography, organizations and users must proactively address vulnerabilities and ensure robust mobile app security.

Despite the existence of established security best practices like secure coding, data encryption, authentication, and authorization enforcement, and secure API design, many developers still overlook or delay the need to implement these security measures until a later phase of the development process. According to [8], many developers, often with only a few years of experience, prioritize feature delivery over security. This practice, known as the "Shift Right" approach, introduces measures only during the testing or deployment phase, typically through basic scanning and penetration testing. Unfortunately, this late-stage testing can miss complex security issues, thereby highlighting a gap in early-stage security implementation in the software development life cycle.

[5] underscore the risk associated with developer reliance on source code from platforms like Stack Overflow or using unsafe third-party dependencies. In their analysis of 1.3 million Android applications, 15.4% were found to contain security-related code snippets sourced from Stack Overflow, and 97.9% included at least one insecure snippet. Such vulnerabilities, introduced through copied code or unsafe third-party libraries, can lead to data leaks, unauthorized access, financial losses, and reputation harm. Addressing these risks requires a "Shift Left" security strategy that integrates vulnerability detection earlier in the software development life cycle (SDLC), thereby increasing developer awareness and improving code integrity from the outset.

In practice, vulnerabilities can also be introduced unintentionally due to developer fatigue, reduced concentration, or time constraints [5]. Moreover, the use of unaudited open-source libraries or community-contributed solutions increases the surface area for supply chain attacks and runtime exploits.

To mitigate this, several attempts have been made, with Defendroid by [12] being the most recent. They were able to detect common vulnerabilities identified by CWE via a blockchain federated neural network with XAI. Also, [14] worked on PrivDroid, which can prevent privilege escalation from the Android source code. However, these tools are limited by their language specificity, primarily supporting Java, and typically operate only during build time or when explicitly invoked, rather than providing continuous feedback. Hence, this study was carried out to accommodate the analysis of Kotlin source codes and afford a synchronous detection of these vulnerabilities.

Given the rapidly evolving nature of security threats in mobile apps, and also the wide adoption of Agile development, it is expedient to manage source code vulnerability detection synchronously, such that developers can ship less risk-prone products, especially Android apps developed with Kotlin.

This paper, therefore, investigates and addresses the security of Kotlin-based Android applications by analyzing common vulnerability patterns in Android source code, proposing a detection model tailored for Android Studio, and implementing this model using the Program Structure Interface (PSI) to enable real-time vulnerability detection during the development process.

II. RELATED WORKS

Security vulnerabilities have evolved with the rise of personal computers and online applications, leading to more complex threats like SQL injection and cross-site scripting (XSS). These vulnerabilities continue to pose significant security risks today.

The [10] highlights the most critical security risks for mobile applications, known as “The OWASP Mobile Top 10”. This list includes:

- i. Improper credential usage: Refers to handling login details or passwords in a way that can be easily exploited.
- ii. Inadequate supply chain security: Risks from third-party components or services that are integrated into an application. If these components are insecure, they can introduce vulnerabilities into the app.
- iii. Insecure authentication: Weak authentication practices that allow unauthorized access to data and features.
- iv. Insecure Communication: The lack of secure transmission channels, like HTTPS, can lead to data interception.
- v. Poor App Security Configurations: This involves improper settings or permissions that can make the app susceptible to attacks.
- vi. Insecure Data Storage: This addresses the risk of sensitive data being compromised due to inadequate protection measures, such as unencrypted files.
- vii. Insufficient Cryptography: Involves the use of outdated or weak encryption methods that expose data to unauthorized access.

Additional risks include Improper Platform Usage, which involves insecure use of platform features and APIs can lead to security lapses if best practices are not followed. Insecure Code, which results from poor coding practices that could expose sensitive information or make the app vulnerable to reverse engineering.

Code Injection involves attackers inserting malicious code into the app, potentially leading to data theft or other compromises. Reverse Engineering refers to the decompilation of apps to discover vulnerabilities or proprietary information.

Extraneous Functionality includes unnecessary features or code that can be exploited if left in the production version of the app. Key security practices include sandboxing, which is used to keep apps isolated, preventing them from accessing other apps' data, while tools like iOS Keychain and Android Keystore protect stored information.

Strong authentication methods like MFA, secure session handling, and ensuring users only access authorized data. Sensitive information, such as personal details or payment data, should be encrypted and securely stored. To make it harder for attackers to reverse-engineer apps, techniques like code obfuscation and anti-reverse engineering are used. Secure coding avoids hardcoded credentials and follows best practices to prevent vulnerabilities.

Encryption protects data both during transmission and on the device, and regular updates keep the app secure against new threats. Regular updates and security patches are essential to fix known vulnerabilities, and penetration testing helps identify and fix security weaknesses by simulating real attacks.

[7] proposed Defendroid, a real-time Android vulnerability detection framework leveraging a blockchain-powered federated neural network with Explainable AI (XAI). Their model, trained in a federated environment, demonstrated high detection performance with an F1-score of 0.96, outperforming previous tools such as MobSF and Qark. Despite its strength, Defendroid does not have support for the Kotlin programming language, which has since 2017 been named the official language for developing Android apps. Also, the developer must manually trigger scans of the code to receive feedback.

The authors in [14] developed PrivDroid, a static analysis tool designed to detect privilege escalation vulnerabilities in Android source code. The tool was tested on over 200 real-world applications and proved effective in identifying potential escalation risks. However, similar to Defendroid, PrivDroid operates post-build, meaning developers only receive security insights after code compilation, a limitation this study aims to overcome.

III. METHODOLOGY

The system was developed following a structured, step-by-step approach guided by a well-defined requirement. The Unified Modeling Language (UML) tool, specifically the

use case diagrams, activity diagrams, and a conceptual framework, was used for the system design.

For implementation, the system was implemented using the IntelliJ Plugin Development Kit, the Kotlin programming language, and the IntelliJ IDEA environment, ensuring seamless integration with Android Studio and full compatibility with Kotlin-based Android projects.

A. Use Case Diagram

The use case diagram models the system's functional requirements by outlining key interactions between actors, namely Android developers, engineering managers, and the system itself, and the corresponding use cases as shown in Fig. 1 below.

B. Activity Diagram

The activity diagram in Fig. 2 shows the interaction between a developer, the IDE, web platforms, and potential malicious actors during the software development process. When a developer encounters an issue during development, they usually search online if they don't know the solution, and may copy code from a web platform, where malicious actors can also post malicious code.

Once the developer pastes in the IDE, the system scans synchronously for vulnerabilities. If a vulnerability is detected, the plugin highlights the code and displays information about the issue. The developer can then decide to address or ignore the vulnerability. This process ensures real-time security feedback, particularly for externally sourced code.

C. System Framework

Fig. 3 below shows how the system operates within the IDE to provide real-time vulnerability detection. When a developer writes or pastes code into the IDE, the plugin scans the code synchronously for security issues. Detected vulnerabilities are immediately flagged, and relevant information is displayed. The system also supports additional test implementations and script executions to validate issues further. All identified vulnerabilities are reported instantly, allowing developers to take immediate corrective action without disrupting their workflow

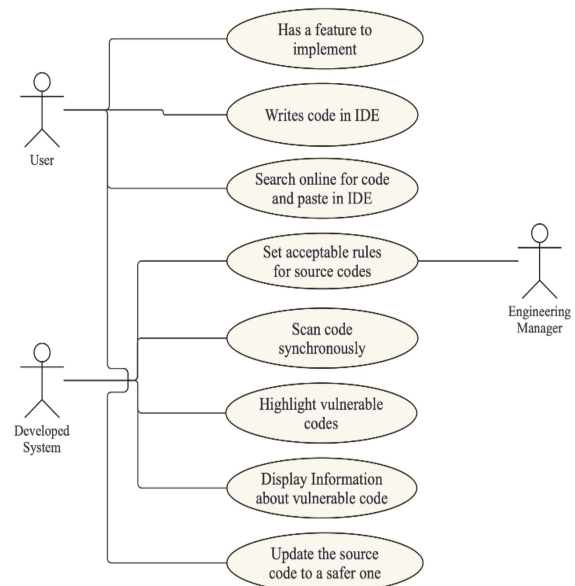


Fig. 1. Use case diagram for synchronous vulnerability detection system.

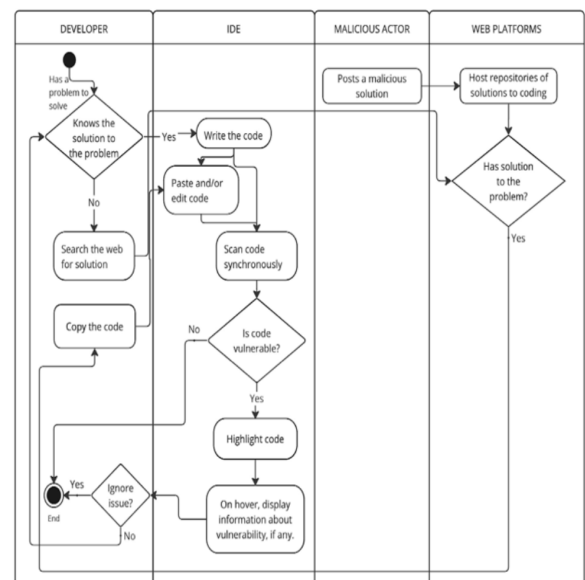


Fig. 2: Activity diagram for synchronous vulnerability detection system.

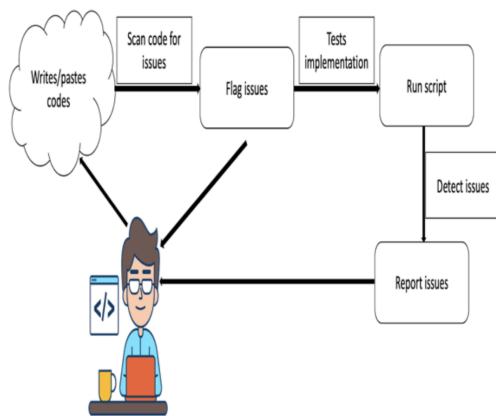


Fig. 3: System framework developed for synchronous vulnerability detection

IV. SYSTEM IMPLEMENTATION

This system is a plugin specifically designed for Android Studio. It leverages the IntelliJ Plugin Development Kit (Dev Kit), the standard tool for developing plugins within JetBrains-based IDEs. The plugin implementation uses the Kotlin programming language and the Program Structure Interface (PSIElement) module to monitor changes in the source code and analyze its structure.

Key Algorithms

The following are the key algorithms employed in the development process.

Algorithm for detecting secrets

```

function isPossibleApiKeyOrSecretKey(input):
  Step 1: Define known regex patterns
  for API/secret keys
  Step 2: Check if input matches any
  known patterns
  for pattern in patterns:
    if regexMatch(input,
    pattern):
      return true
  Step 3: Perform heuristic checks if no
  patterns match
  length = len(input)
  isAlphanumeric = allCharactersInSet(input, "A-Za-z0-9_-.")
  if length >= 15 and length <= 64 and
  isAlphanumeric:
    return true
  Step 4: Return false if checks fail
  return false
  
```

An algorithm for detecting unsanitized SQL

```

function isSQLQuery(input):
  Step 1: Check if input starts with any
  sql query start command [Select,
  Update, Drop, Alter, Delete,
  insert, Create.]
  Step 2: for all assignment operator
  (=) in input:
  
```

```

    if the next character ignoring
    spaces is not a value placeholder(?):
      return true
    else: return false
  
```

```

  Step 3: Return false if checks fail
  return false
  
```

Algorithm for flagging vulnerabilities

```

function annotate(input):
  
```

```

  Step 1: Check if input meets specified
  vulnerabilities
  
```

```

  Step 2: highlight the affected portion
  of code
  
```

```

  Step 3: Add a message to the holder
  
```

```

  Step 4: Add an underline to the
  affected portion
  Step 5: create
  annotation
  
```

```

  Step 6: Do nothing if false.
  
```

A. How it works

When the plugin detects a vulnerability in the source code, it visually highlights the issue based on its severity level. Critical vulnerabilities are marked in red, immediately drawing the developer's attention, while less severe issues are highlighted in yellow. This intuitive color-coding allows developers to quickly assess the urgency of the problem at a glance.

In addition to the visual indicators, the plugin attaches a contextual message to the highlighted code. When the message is hovered over, a tooltip appears with the detailed information about the vulnerability, guiding the developer on how to resolve it. This combination of real-time alerts and feedback simplifies the process of identifying and fixing security issues during development.

Fig. 4 shows a use case where the plugin detects an SQL injection vulnerability, highlighting the affected code and displaying an informative message on hover.

B. How to install Vdetekt

To use the plugin, users must first have Android Studio installed. For manual installation, navigate through the following steps: Android Studio → Settings → Plugins →  → Install Plugin from Disk, and then select the downloaded .zip file and proceed with the installation. A restart of the IDE will be required to complete the setup.

Alternatively, the plugin can be installed directly from the JetBrains Marketplace by searching for VDetekt under the Plugin tab.

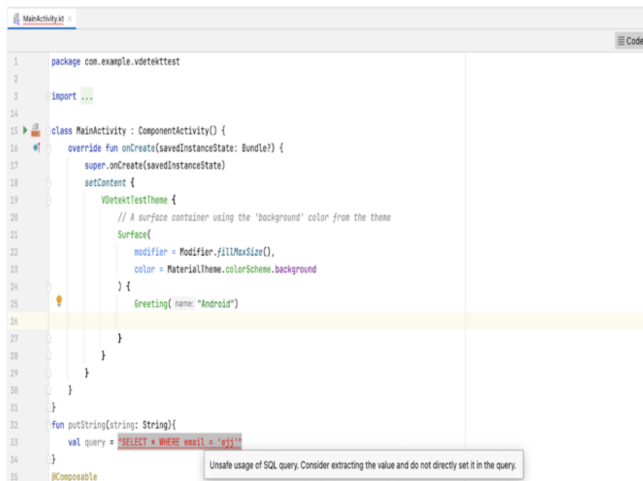


Fig. 4. Unsafe SQL query detection

V. RESULT AND DISCUSSION

To evaluate the proposed system, the plugin was distributed to a group of eleven Android developers, and a survey was conducted to gather feedback on its effectiveness in real-world development environments. The results, as shown in Figs. 5, 6, and 7 indicate that the plugin successfully detected and flagged SQL injection vulnerabilities with 100% accuracy across all tested cases. It also demonstrated strong performance in identifying hardcoded secrets associated with insecure data communication, achieving a 90% detection rate.

However, the plugin's performance in detecting insecure data storage was comparatively lower, with a 72% success rate. In this case, while several vulnerable instances were correctly flagged, some were missed, suggesting the need for further refinement of the detection algorithm in this category.

These results highlight the plugin's effectiveness in identifying common security vulnerabilities during development and its potential to enhance secure coding practices, particularly for Kotlin-based Android applications.

The developers were also asked to assess the relevance of the plugin to their workflow. An overwhelming 90% of respondents rated the plugin as "Very relevant", indicating that it significantly enhanced their ability to write secure code. The remaining 10% found it "relevant", reflecting a universally positive reception. This distribution is shown in Figure 8. Overall, the feedback emphasized the plugin's practical value and strong performance in helping developers identify and address vulnerabilities during the development process.

Was your code highlighted when you set parameter directly in an sql query string?

11 responses

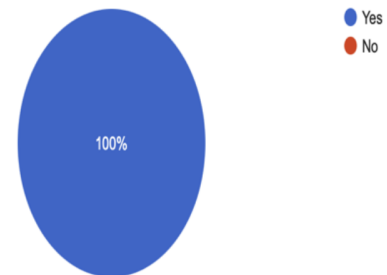


Fig. 5: SQL Injection check response

Was your code highlighted when you declared a secret or API key?

11 responses

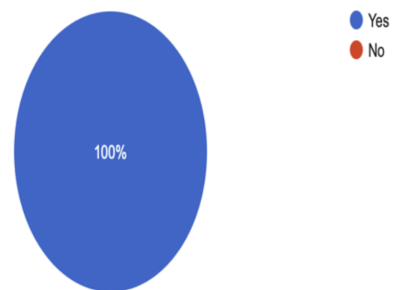


Fig. 6: Hardcoded Secrets Response

When you try to use putString in sharedPreferences, Did the plugin flag a non-encrypted string?

11 responses

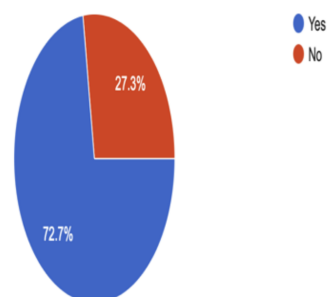


Fig. 7: Insecure data storage detection response

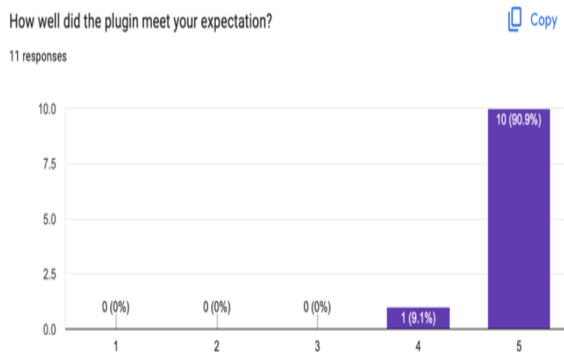


Fig. 8: Developer's perception

VI. LIMITATION

While the proposed plugin demonstrates strong performance in detecting common vulnerabilities in Kotlin-based Android applications, several limitations were identified during testing and developer feedback:

1. **Kotlin-only support:** The plugin is tailored for Kotlin, which limits its applicability in projects written in Java or other languages. This reduces its adoption in legacy or multi-language Android projects.
2. **IDE Dependency:** The plugin is designed specifically for Android Studio, which may not support developers using an alternative IDE or text editor.

VII. CONCLUSION AND FUTURE WORK

This study presents the design and implementation of VDetekt, a real-time vulnerability detection plugin tailored for Kotlin-based Android applications. It contributes a significant advancement to Android app development security by offering developers immediate, in-editor feedback on potential security issues during the development process. This is in contrast to many other tools that are either difficult to use or do not work well with Kotlin. The evaluation results show strong detection performance and high developer satisfaction, with 90% of users rating the plugin as "very relevant" to their development workflow. The research shows that real-time detection is a potent strategy for boosting app security, even though there is space for improvement in identifying specific kinds of vulnerabilities. This system helps developers build more secure apps and increases awareness of potential security risks at every stage of the development process.

To enhance the plugin's robustness and broaden its adoption, several areas have been identified for future development:

- i. **Java Support Integration:** Extending support to Java-based Android applications would make the

plugin more versatile and accessible to a broader developer base.

- ii. **Machine Learning Integration:** Integrating machine learning models for adaptive pattern recognition could improve the detection scope of vulnerabilities, especially in large and dynamic codebases.
- iii. **Custom Rule Definition:** Enabling developers to define and manage custom detection rules would increase flexibility and allow the plugin to adapt to any security policies

REFERENCES

- [1]. Amaefule, A. and Ogwueleka, F. (2020) Criteria for choosing the right software development lifecycle method for the success of a software project. *Journal of Innovation in Computing*, Vol. 1, Issue 1, pg. 17.
- [2]. Balapour, A., Nikkhah, H.R., & Sabherwal, R. (2020) Mobile application security: Role of perceived privacy as the predictor of security perceptions. *International Journal of Information Management* Volume 52, 102063.
- [3]. Bovenzi, G., Persico, V., Pescapé, A., Piscitelli, A., Spadari, V. (2022) Hierarchical Classification of Android Malware Traffic. In *Proceedings of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Wuhan, China, 9–11 December 2022; pp. 1354–1359.
- [4]. Cifuentes, C., Gauthier, F., Hassanshahi, B., Krishnan, P., McCall, D. (2023) The role of program analysis in security vulnerability detection: Then and now. *ScienceDirect Computers & Security*, Volume 135.
- [5]. Fischer, F., Bottinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., Fahl, S.(2017) Stack Overflow Considered Harmful? The Impact of Copy & Paste on Android Application Security. *IEEE Symposium on Security and Privacy*, pp 121.
- [6]. Garg, S., and Baliyan, N. (2020) Android security assessment: A review, taxonomy, and research gap study. *ScienceDirect Computers & Security*, Volume 100.
- [7]. Gowdhaman, V. and Dhanapal, R. (2021) An intrusion detection system for wireless sensor networks using deep neural networks. *Soft Comput.* 2021, 26, 13059–13067.
- [8]. "HackerRank Developer Skills Report",
- [9]. <https://www.hackerrank.com/research/developer-skills/2024>, [Accessed 08-01-2024].
- [10]. Janaka, M.D., Kalutarage, H., Petrovski, A., Piras, L., and Al-Kadri, M.O. (2024) Defendroid: Real-time Android code vulnerability detection via blockchain federated neural network with XAI. *Journal of Information Security and Applications*, 82(4), p.103741.
- [11]. "What are the different types of malware?",

- [12]. <https://www.kaspersky.com/resource-center/threats/types-of-malware>, [Accessed 08-01-2024].
- [13]. “Top 10 2024: Final Release Updates”, <https://owasp.org/www-project-mobile-top-10/> [Accessed: 01-05-2024].
- [14]. Reychav, I., Beeri, R., Balapour, A., Raban, D. R., Sabherwal, R., Azuri, J. (2019) How reliable are self-assessments using mobile technology in healthcare? The effects of technology identity and self-efficacy. *Computers in Human Behavior*, 91, 52–61.
- [15]. Senanayake, J., Kalutarage, H., Al-kadri, M.O., Petrovski, A., & Piras, L. (2023) Android Source Code Vulnerability Detection: A Systematic Literature Review. *ACM Computing Surveys*, Volume 55, Issue 9, pp 187:9, 12,28-29.
- [16]. Senanayake, J., Kalutarage, H., Petrovski, A., Piras, L., Al-Kadri, M.O. (2024) Defendroid: Real-time Android code vulnerability detection via blockchain federated neural network with XAI. *Journal of Information Security and Applications*, 82, p.103741, pg. 11.
- [17]. Tebib, M., Amin, E., André, P., Graa, M., Aktouf, O. (2023) PrivDroid: Android Security Code Smells Tool for Privilege Escalation Prevention. *IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on CyberScience and Technology*
- [18]. Congress(DASC/PiCom/CBDCom/CyberSciTech), Abu Dhabi, United Arab Emirates, pp. 1024-1031.
- [19]. Thakkar, A. and Lohiya, R. (2021) A Review on Machine Learning and Deep Learning Perspectives of IDS for IoT: Recent Updates, Security Issues, and Challenges. Springer: Dordrecht, The Netherlands, 2021; Volume 28, pp. 3211–3243.