International Journal of Advanced Smart Convergence Vol.13 No.2 48-60 (2024)
http://dx.doi.org/10.7236/IJASC.2024.13.2.48

IJASC 24-2-7

Application Consideration of Machine Learning Techniques in Satellite Systems

Jin-keun Hong

Professor, Division of Advanced IT, Baekseok University, Korea
E-mail jkhong@bu.ac.kr

Abstract

With the exponential growth of satellite data utilization, machine learning has become pivotal in enhancing innovation and cybersecurity in satellite systems. This paper investigates the role of machine learning techniques in identifying and mitigating vulnerabilities and code smells within satellite software. We explore satellite system architecture and survey applications like vulnerability analysis, source code refactoring, and security flaw detection, emphasizing feature extraction methodologies such as Abstract Syntax Trees (AST) and Control Flow Graphs (CFG). We present practical examples of feature extraction and training models using machine learning techniques like Random Forests, Support Vector Machines, and Gradient Boosting. Additionally, we review open-access satellite datasets and address prevalent code smells through systematic refactoring solutions. By integrating continuous code review and refactoring into satellite software development, this research aims to improve maintainability, scalability, and cybersecurity, providing novel insights for the advancement of satellite software development and security.

The value of this paper lies in its focus on addressing the identification of vulnerabilities and resolution of code smells in satellite software. In terms of the authors' contributions, we detail methods for applying machine learning to identify potential vulnerabilities and code smells in satellite software. Furthermore, the study presents techniques for feature extraction and model training, utilizing Abstract Syntax Trees (AST) and Control Flow Graphs (CFG) to extract relevant features for machine learning training. Regarding the results, we discuss the analysis of vulnerabilities, the identification of code smells, maintenance, and security enhancement through practical examples. This underscores the significant improvement in the maintainability and scalability of satellite software through continuous code review and refactoring.

Keywords: *satellite, AST, CFG, smell, machine learning*

1. Introduction

With the rapid advancements in satellite technology, the utilization of satellite data across various application domains has seen exponential growth globally. Satellite data, in particular, has brought about significant changes in the field of machine learning, accelerating innovation in satellite systems and data

Manuscript Received: April. 15, 2024 / Revised: April. 21, 2024 / Accepted: April. 26, 2024
Corresponding Author: jkhong@bu.ac.kr
Tel: +82-41-550-2445, Fax: +82-41-550-9027
Professor, Division of Advanced IT, Baekseok University, Korea)

processing.

Machine learning techniques present new analytical methodologies and automated solutions within the complex environment of satellite systems, and they are extensively employed across domains ranging from cybersecurity to software quality enhancement.

The architecture of a satellite system can be broadly classified into the ground segment, space segment, and user segment:

Ground Segment: Comprises ground control centers, communication networks, data processing and storage systems, and user support devices. This segment is responsible for satellite operation and control and for processing and analyzing data collected from the satellites to provide it to end-users.

Space Segment: Refers to the satellites in orbit and includes both the satellite bus and the payload.

Satellite Bus: Manages the operation of the satellite itself, including power supply, attitude control, thermal regulation, and propulsion.

Payload: Carries out mission-specific functions and includes observation sensors, communication devices, and data processing systems.

User Segment: Refers to devices or organizations that directly utilize satellite data and play a role in delivering satellite information and services to end-users.

Machine learning techniques have become indispensable tools in the satellite domain, applied in areas such as vulnerability analysis of satellite components, testing, source code refactoring, code smell detection, program understanding and synthesis, source code summarization, security flaw detection, feature-based learning, design pattern detection, and software quality analysis. In particular, cybersecurity in satellite systems has emerged as a significant issue, highlighted in U.S. cybersecurity strategies and policy research. Satellite systems play a pivotal role in national security, communications, navigation, and meteorological observation, and cybersecurity threats to these systems present serious challenges both nationally and globally. The security threats to satellite systems are complex and varied, with attack methods constantly evolving.

This paper addresses the importance of cybersecurity in satellite systems, drawing on U.S. satellite cybersecurity strategies and policies as a foundation. Section 2 analyzes the machine learning techniques in satellite systems, while Section 3 introduces cases of code smells and refactoring in satellite system source code. Section 4 concludes the paper by suggesting future research directions, including security enhancement through the application of machine learning techniques in satellite systems, datasets, code smells, and refactoring.

Through this research, we aim to provide novel insights into how machine learning techniques can be utilized to solve security challenges in satellite systems and contribute to the advancement of satellite software development and security fields.

2. Machine Learning Techniques in the Satellite Systems

2.1 Composition of Satellite System Software

The software architecture of a satellite system can be broadly divided into the following modules [1][2][3]:

Satellite Operations Software: Software responsible for the operation of the satellite bus, including attitude control, orbit maintenance, and power management. Typically developed as a real-time embedded system.

Mission Software: Software that executes the functions of the payload, including sensor data collection, processing, compression, transmission, and communication device control.

Ground Control Software: Software that manages satellite operations from ground control centers, handling command transmission, data reception, status monitoring, and orbit tracking.

Data Processing Software: Software that processes and analyzes data received from the satellite, employing various data processing algorithms, including machine learning techniques.

Security Software: Software designed to strengthen the cybersecurity of satellite systems, incorporating intrusion detection, authentication, encryption, and vulnerability analysis for satellite components and communication networks.

The application of machine learning techniques in the satellite domain spans a variety of areas, including vulnerability analysis of satellite components, testing, source code refactoring, code smell detection, program understanding and synthesis, source code summarization, security flaw detection, feature-based learning, design pattern detection, and software quality analysis. These technologies have established themselves as critical tools for enhancing the efficiency and security of satellite systems.

In particular, cybersecurity in satellite systems has emerged as a significant issue, as highlighted in U.S. cybersecurity strategy and policy research. Satellite systems play a pivotal role in national security, communications, navigation, and meteorological observation, and cybersecurity threats to these systems present serious challenges both nationally and globally. The security threats to satellite systems are complex and diverse, with attack methods continuously evolving

2.2 Feature Training Techniques for Vulnerability in Satellite System

In satellite systems, feature training techniques aim to identify vulnerabilities and malicious code patterns by understanding the structure and semantics of the source code. The initial step involves feature extraction using representations such as Abstract Syntax Trees (AST) or Control Flow Graphs (CFG). Subsequently, a vectorization process transforms the extracted features, which are then merged and fed into machine learning training models for prediction and evaluation.

2.2.1 Feature Extraction

Techniques used to identify vulnerabilities or code patterns include AST analysis, CFG analysis, and Natural Language Processing (NLP)-based embedding.

AST Analysis: in AST analysis, the source code is first converted into an AST to identify structural features. The AST is then transformed into a vector, which serves as an input to the machine learning model.

CFG Analysis: in CFG analysis, the source code's control flow is represented as a graph to understand the execution flow of the code. The CFG is subsequently converted into a vector to extract features.

NLP-Based Embedding: in NLP-based embedding, the source code is treated as text, and techniques like Word2Vec, FastText, or BERT are employed for embedding. Word embeddings help comprehend the semantics of the code and facilitate feature extraction.

1) Feature Extraction from Satellite System Code Using AST

For feature extraction using AST, a class named ASTFeatureExtractor() is required, which converts AST node types into strings to extract relevant characteristics from the source code. Here is a description of the feature extraction process:

```
class ASTFeatureExtractor(ast.NodeVisitor):
    def __init__(self):
```

In addition, the class requires the definition of a visit method. This method extracts node types as strings using node.__class__.__name__ and adds them to the features list. It also calls generic_visit to traverse each node.

```
    def visit(self, node):
```

Furthermore, the class requires the definition of the extract method. This method parses the source code into an AST and uses the visit method to extract node types, which are stored in the features list. The extracted node types are then returned as a space-separated string.

```
    def extract(self, source_code):
```

2) Feature Extraction from Satellite System Code Using CFG

The core of feature extraction using CFG involves constructing a control flow graph from the source code and requires the definition of a CFGFeatureExtractor class.

```
class CFGFeatureExtractor(ast.NodeVisitor):
    """
    Build a CFG from source code and extract features.
    """
```

Within the class, an __init__ method is defined to initialize the graph and the counter.

```
def __init__(self):
```

Within the class, an add_node method is defined to add nodes to the graph and return a unique ID

```
def add_node(self, label):
```

Within the class, an add_edge method is defined to add edges to the graph.

```
    def add_edge(self, from_node, to_node):
```

To provide a comprehensive implementation, it is included the visit_FunctionDef method, which adds function definition nodes to the control flow graph.

```
def visit_FunctionDef(self, node):
```

To complete the CFGFeatureExtractor class, we'll also define the visit_If, visit_For, and visit_While methods to handle conditional statements and loop structures.

```
def visit_If(self, node):
        def visit_For(self, node):
        def visit_While(self, node):
```

To handle assignment statements, it is defined the visit_Assign method.

```
        def visit_Assign(self, node):
```

To include return statements, the visit_Return method is defined as well.

```
        def visit_Return(self, node):
```

Here's the CFGFeatureExtractor class with visit_Call methods defined.

```
        def visit_Call(self, node):
```

To facilitate the complete parsing and graph construction process, it is defined a build method that will parse the source code into an AST and then call the appropriate visit methods to construct the control flow graph. Here's the CFGFeatureExtractor class with the build method.

```
        def build (self, source_code):
```

2.2.2 Vectorization Step for Extracted Features

To transform features from the AST and CFG into vectors, the TfidfVectorizer is used. Term Frequency - Inverse Document Frequency(TF-IDF) combines two measures:
TF : Measures how frequently a term appears in a document.
IDF : Measures how rarely a term appears across all documents.

By combining the indices of TF and IDF, weights are calculated and vectorized using TfidfVectorizer.

```
vectorizer = TfidfVectorizer()
```

The result is returned in the form of a sparse matrix, where most elements are '0'. This format minimizes memory usage by storing only non-zero elements and their indices.

```
# Vectorization of AST and CFG features
X_ast = vectorizer.fit_transform(ast_features)
X_cfg = vectorizer.fit_transform(cfg_features)
```

A sparse matrix efficiently stores only the non-zero elements and their indices, eliminating memory waste by not storing zero elements. The TF-IDF sparse representation can be illustrated as follows:

```
TF-IDF Matrix (Sparse Representation):
   (0, 6)          0.4697910006742352
   (0, 7)          0.4697910006742352
```

2.2.3 Merging Extracted Features

To integrate features from both AST and CFG, the sparse matrices need to be combined. Using NumPy's hstack function, the AST and CFG features are merged. After converting the sparse matrices to dense arrays, the combination is achieved as follows:

```
X_combined = np.hstack([X_ast.toarray(), X_cfg.toarray()])y = [0, 1]
```

The merged feature matrix is then used as input for the machine learning model.

2.2.4 Prediction and Evaluation in Machine Learning Model Training

1) Training and Prediction Code:

The labels are defined as 0 for normal code and 1 for vulnerable code:

```
y = [0, 1]
```

The data is split into training and testing sets using train_test_split:

```
X_train_, X_test_, y_train_, y_test_ = train_test_split(X_combined, y, test_size=0.5,
random_state=xx)
```

A random forest classifier is then used to train and fit the model:

```
model = RandomForestClassifier(n_estimators=100, random_state=xx)
model.fit(X_train_, y_train_)
```

Based on the trained model, predictions are made using the predict function:

```
y_pred_ = model.predict(X_test_)
```

2) Training Code for Machine Learning Models

The training codes used for machine learning model training can be found in the following list of models [4]:

```
models = {
    "Logistic Regression": LogisticRegression(random_state=xx),
    "Support Vector Machine": SVC (random_state=xx),
    "Random Forest": RandomForestClassifier(n_estimators=100, random_state=xx),
    "Gradient Boosting": GradientBoostingClassifier(n_estimators=100,
  random_state=xx),
    "K-Nearest Neighbors": KNeighborsClassifier(n_neighbors=xx),
    "Multilayer Perceptron": MLPClassifier(hidden_layer_sizes=(100,), max_iter=500,
random_state=xx),
    "Decision Tree": DecisionTreeClassifier(random_state=xx),
    "Naive Bayes": MultinomialNB()
```

Logistic Regression: Logistic regression is used in satellite software for binary and multi-class classification tasks like anomaly detection and fault prediction. By using the sigmoid function, it estimates the probability of different error types.

Support Vector Machine (SVM): SVM models help classify data in satellite software, such as identifying communication anomalies. They seek the best decision boundary (hyperplane) between normal and anomalous data and can use linear, polynomial, RBF, and sigmoid kernels for both linear and non-linear problems [5].

Random Forest: Random forest models, which aggregate predictions from multiple decision trees, are used in satellite software for fault detection and data classification. They are robust against overfitting and provide insights into feature importance.

Gradient Boosting: Gradient boosting combines weak learners to create strong classification models for detecting signal interference and predicting errors. Each learner improves upon the previous one, making it suitable for handling imbalanced data like rare faults.

K-Nearest Neighbors (KNN): KNN is useful for classifying data anomalies and segmenting satellite imagery. It predicts new data classes based on the nearest neighbors' classes, using distance metrics to find the k closest neighbors.

Multilayer Perceptron (MLP): MLP, a supervised learning model based on artificial neural networks, is effective for analyzing satellite sensor data and classifying signals. It uses hidden layers and backpropagation to learn complex, non-linear patterns.

Decision Tree: Decision trees help diagnose faults and categorize components in satellite software by generating decision rules using a tree structure. They are easy to interpret but can overfit without careful pruning.

Naive Bayes: Naive Bayes models are used in satellite software for classifying communication signals. They assume conditional independence among features and include variants like multinomial, Gaussian, and Bernoulli models.

2.2.5 Prediction and Evaluation of Models

In the classification evaluation phase of the trained models, metrics such as the confusion matrix, accuracy, and recall are used.

```
for name, model in models.items():
    model.fit(X_train, y_train_)
    y_pred_ = model.predict(X_test_)
    accuracy = accuracy_score(y_test_, y_pred_)
```

In the classification evaluation stage of training models, the metric indicators used include the confusion matrix, Area Under Curve (AUC), accuracy, F- measures, precision, and Recall [5].

2.3 Open data sets based on Satellites System or making using Satellite Camera

In recent years, the surge in availability of high-resolution satellite imagery has catalyzed significant advancements in remote sensing research. From monitoring wildfire dynamics to securing satellite communication, researchers are harnessing the power of these datasets to address pressing global challenges. This article presents two notable datasets that exemplify this trend: a comprehensive wildfire imagery dataset for segmentation and a robust dataset focused on identifying unique message headers from the Iridium satellite constellation.

For, Landsat-8, Sentinel-1, and Sentinel-2 Satellite Imagery Dataset, Daniele Rege Cambrin and his colleagues have curated an image dataset sourced from the California Department of Forestry and Fire Protection. Their research aims to semantically segment wildfire-affected areas, distinguishing between burned and unburned regions using binary classification. The Sentinel-2 dataset leverages computer vision techniques and satellite imagery to address the open problem of binary segmentation of burned regions. This dataset is composed of pre- and post-wildfire imagery from California wildfires dating back to 2015, obtained via Sentinel-2 L2A [6]. The dataset is publicly available at Hugging Face, and the source input data were collected from the Copernicus Open Access Hub using Sentinel-2 L2A.

In another study, researchers present the SICKLE dataset, a multi-resolution time series dataset collected from Landsat-8, Sentinel-1, and Sentinel-2 satellites. This dataset encompasses time-series imagery taken from January 2018 to March 2021 [7].

For Iridium Message Header Dataset, Joshua Smailes and his team focus on the Iridium satellite constellation to collect a dataset of 1,705,202 messages [8]. They aim to train a fingerprint model, which combines a Siamese neural network with an autoencoder, for effective encoding and learning of message headers while preserving information identification.

The researchers analyze temporal stability by considering the time gap between training and testing datasets, introducing transmitters that reflect this temporal change. The dataset and SatIQ machine learning code are publicly available (GitHub, Zenodo Record 1, Zenodo Record 2). In their dataset, they emphasize high sample rate data capture and the consistency of message headers across messages.

Since each satellite shares identical transmitter hardware, distinguishing satellites requires identifying the subtle differences present from the time of manufacturing. By training machine learning models to recognize these variations, the team aims to differentiate between satellites. They store the IQ samples of message headers alongside the demodulated message bytes and decoded message contents. The decoded messages are used to label the data and generate a corresponding dataset of message headers.

To collect training data, the researchers gathered one million messages over 23 days, with 872 messages per transmitter. They pre-process the data to remove channel noise and scale the waveforms. For training, validation, and testing, they apply a 90:5:5 split ratio, utilizing 50,000 messages for validation and testing each. Performance evaluation metrics include the Equal Error Rate (EER), which reflects the rate where False Positive Rate (FPR) equals False Negative Rate (FNR), and the AUC of the Receiver Operating Characteristic (ROC) curve. The study also examines the correlation between the distance between messages in the embedding space and the temporal gap between messages.

For GNSS Satellite-Based ITS V2AIX Dataset, the V2AIX dataset encompasses a diverse set of real-world V2X messages, including the Cooperative Awareness Message (CAM) standardized by ETSI. The dataset comprises 230,000 V2X messages collected from 1,800 vehicles and roadside units involved in public road traffic, all adhering to the ETSI ITS message set. The dataset meticulously logs the geographical coordinates (longitude and latitude) measured via GNSS, with vehicle onboard units recording location data through both GNSS and INS systems [9].

These datasets, with their unique characteristics and high-quality annotations, provide an invaluable resource for the research community. Whether delineating wildfire boundaries or fingerprinting satellite message headers, these datasets empower researchers to develop innovative solutions that advance our understanding of the environment and enhance global security. By making these datasets openly accessible, the authors encourage further exploration and collaboration in these critical fields.

3. Smell and refactoring of source code in satellite system

In satellite systems, source code often incorporates open-source components. However, not all open-source software in practical use is free of vulnerabilities or code smells. Thus, addressing code smell issues in satellite systems is imperative. The types of code smells that need to be considered include the following:

Code smells indicate poor design decisions reflected in the software's source code. They make software maintenance challenging and compromise its robustness.

Examples of code smells include God Class, Long Method, Feature Envy, Spaghetti Code, Functional Decomposition, Data Class, Swiss Army Knife, Duplicated Code, Lazy Class, Long Parameter List, Message Chain, Anti-Singleton, Class Data Should Be Private, Complex Class, Refused Parent Bequest, Speculative Generality, Delegator, Middle Man, Switch Statement, and Large Class [1][4] [10-21].

Table 1 identifies prevalent code smells in satellite systems and provides targeted solutions:
God Class: Split monolithic classes into smaller, specialized modules, like DataCapturer and DataProcessor.
Long Method: Refactor lengthy methods into modular sub-methods, enhancing readability.
Feature Envy: Move methods to the classes where the data is primarily located.
Spaghetti Code: Refactor tangled code into clearly defined, modular components.
Functional Decomposition: Decompose complex functions into meaningful sub-methods.
Data Class: Add data manipulation methods directly within data classes.
Swiss Army Knife: Divide multipurpose classes into specialized components.
Duplicated Code: Consolidate duplicate code into reusable utility methods.
Lazy Class: Merge redundant classes or remove them entirely.
Long Parameter List: Use Data Transfer Objects (DTOs) to encapsulate parameter lists.
Message Chain: Simplify long chains of calls by adding intermediate methods.
Anti-Singleton: Properly implement the Singleton pattern or use a regular class.
Class Data Should be Private: Make class data private and use getters/setters.
Complex Class: Break down complex classes into manageable components.
Refused Parent Bequest: Remove inheritance and create independent classes.
Speculative Generality: Eliminate unused abstractions and simplify structures.
Delegator: Eliminate unnecessary delegation and perform tasks directly.
Middle Man: Remove middleman classes and directly interact with core components.
Switch Statement: Replace switch statements with polymorphism.
Large Class: Split large classes into smaller, modular components.

Sim et al. investigate the vulnerabilities and defense models in satellite-based communication systems, focusing on countermeasures against attacks. They consider the key characteristics of satellites [22].

Also, Table 1 shows about solution cases of before and after smell for satellite system.

Table 1. Cases of before and after smell for satellite system (Telemetry data processing).

Items	Smell	Solution
God class	The TelemetryProcessor class contains all functions related to telemetry data management.	Split the class into specialized components, such as TelemetryReader, TelemetryValidator, and TelemetryAnalyzer.
Long method	The process_telemetry() method	Decompose process_telemetry() into

	is too long, making it difficult to understand and maintain.	smaller, self-contained sub-methods, such as read_data(), filter_data(), and validate_data().
Feature envy	The analyze_data() method relies heavily on the SignalProcessor class.	Move analyze_data() to the SignalProcessor class and add utility methods to simplify the telemetry analysis process.
Spaghetti code	The TelemetryProcessor class has tangled code, making it hard to follow.	Modularize the class by refactoring into individual classes like TelemetryReader, TelemetryValidator, and TelemetryAnalyzer.
Functional decomposition	Functions in TelemetryProcessor lack logical decomposition, resulting in bloated methods.	Break down methods into sub-methods like filter_data(), normalize_data(), and validate_data().
Data class	The TelemetryData class only stores raw telemetry data without any associated methods.	Add methods to manipulate telemetry data directly within the TelemetryData class.
Swiss Army Knife	The TelemetryHandler class handles diverse, unrelated functions such as data collection, processing, and transmission.	Separate into specialized classes like TelemetryCollector, TelemetryProcessor, and TelemetryTransmitter.
Duplicated code	Code for data validation is duplicated across multiple parts of the telemetry system.	Extract validation logic into a reusable utility class.
Lazy class	The SignalAnalyzer class has minimal functionality.	Merge it with SignalProcessor or remove entirely.
Long parameter	The transmit_data() method in TelemetryTransmitter has too many parameters..	Group parameters into a single cohesive object, such as TransmissionConfig.
Message chain	The telemetry processing system involves excessively long message chains to pass data through multiple intermediary objects.	Introduce a communication manager class that abstracts message details, reducing the chain complexity.
Anti-singleton	Singleton implementation for telemetry storage is either incorrect or unnecessary, resulting in tightly coupled dependencies.	Implement the Singleton pattern properly, or convert to a regular class if Singleton is not required.
Class data should be private	Publicly accessible class data exposes telemetry data, signal parameters, and configuration settings directly.	Encapsulate class data and provide getters/setters.
Complex class	The class complexity is too high, exemplified by a monolithic SatelliteControl class handling thermal regulation, power management, and attitude control.	Split the class into smaller, specialized classes.
Refused parent bequest	Subclass inherits from a parent class but doesn’t utilize its properties or methods.	Abandon inheritance and create an independent class if the base class isn't fully applicable.
Speculative generality	Unused generalized structures, such as redundant interfaces or abstract classes, are implemented unnecessarily.	Remove unused abstract structures.

Delegator	Excessive delegation without adding value, leading to unnecessary complexity.	Refactor to directly perform data processing or signal filtering.
Middle man	An intermediary class provides minimal value and introduces redundant layers in data transmission.	Remove the middle class and interact directly with the relevant class.
Switch statement	Switch statements are excessively used for telemetry processing types.	Replace switch statements with polymorphism.
Large class	The class size is too large, encompassing data capture, processing, and transmission.	Split the class into smaller, modular components based on functionality.

In satellite systems, maintaining clean, efficient, and secure code is crucial for ensuring system reliability and performance. By proactively identifying and mitigating code smells through refactoring, developers can significantly improve the maintainability, robustness, and longevity of satellite system software. Thus, continuous code review and refactoring should be integral parts of the software development lifecycle in satellite systems. Addressing these code smells not only enhances the maintainability, readability, and efficiency of satellite system code but also ensures that the software remains robust and scalable for future developments. Refactoring is essential to building a reliable and sustainable satellite software ecosystem.

4. Conclusions

Addressing code smells in satellite source code through refactoring is essential for improving maintainability, readability, and efficiency, ultimately ensuring a more robust and scalable satellite software ecosystem. In conclusion, this research underscores the imperative of integrating continuous code review and refactoring into the software development lifecycle of satellite systems. Proactively addressing code smells not only improves the software's efficiency but also strengthens its resilience against future cybersecurity threats. By fostering collaboration and open access to high-quality datasets, the research community can drive significant advancements in satellite software development and security. Ultimately, leveraging machine learning techniques for cybersecurity challenges in satellite systems will pave the way for a more secure, innovative, and sustainable future in satellite technology.

Acknowledgement

This work was supported by funding at Baekseok University in 2024.

References

[1] T. Liu, C. Sun, and Y. Zhang, “Load Balancing Routing Algorithm of Low-Orbit Communication Satellite Network Traffic Based on Machine Learning,” *Wireless Communications and Mobile Computing*, pp. 1-14, 2021. DOI: https://doi.org/10.1155/2021/3234390.

[2] B. H. Choi, Y. -J. Song, and J. -H. Won, “Design of a KPS Civil Signal Candidate Simulator Using a Fully-Reconfigurable GNSS Signal Generator,” in *Proc of IPNT Conference*, pp. 23-27, Nov 2-4, 2022. *http://ipnt.or.kr/2022proc/14.*

[3] NASA Advanced Supercomputing (NAS) Division, Open Source Software, [Internet] cited 2024 May 08, Available from: *https://www.nas.nasa.gov/software/software.html.*

[4] Alazba and H. Aljamaan, "Code smell detection using feature selection and stacking ensemble," *Empirical investigation, Information and Software Technology*, Vol. 138, pp. 106648, 2021. DOI: https://doi.org/10.1016/j.infsof.2021.106648.

[5] V. Markovic, Z. Jakovljevic, and Z. Miljkovic, "Feature sensitive three-dimensional point cloud simplification using support vector regression," *Tehnički vjesnik*, Vol. 26, pp. 985-994, 2019. DOI: https://doi.org/10.17559/TV-20180328175336.

[6] D. R. Cambrin, L. Colomba, and P. Graza, "CaBuAir: California Burned Areas dataset for delineation," *IEEE Geoscience and Remote Sensing Magazine*, Vol. 11, pp. 106-113, 2024. DOI: https://doi.org/10.48550/arXiv.2401.11519.

[7] Sani, D., Mahato, S., Saini, S., Agarwal, H. K., and C. C. Devshali, " SICKLE: A Multi-Sensor Satellite Imagery Dataset Annotated with Multiple Key Cropping Parameters," WACV [Internet] cited 2024 May 08, Available from: *https://sites.google.com/iiitd.ac.in/sickle/home*

[8] J. Smailes, S. Kohler, S. Birnbach, M. Strohmeier, and I. Martinovic, "Watch This Space: Securing Satellite Communication through Resilient Transmitter Fingerprinting," in *Proc of CCS'23*, pp.1-14, Nov 26-30, 2023. [Internet], cited 2024 May 08, Available from: *https://arxiv.org/pdf/2305.06947*.

[9] Kueppers, G., Busch, J. P., Reiher, L., & Eckstein, L. 2024, V2AIX: A Multi-Modal Real-World Dataset of ETSI ITS V2X Messages in Public Road Traffic, [Internet], cited 2024 May 08, Available from*: https://arxiv.org/html/2403.10221v1.*

[10] F. A. Fontana, M. V. Mantyla, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, Vol. 21, pp. 1143-1191, 2016. DOI: https://doi.org/10.1007/s10664-015-9378-4.

[11] A. Kaur, S. Jain, and S. Goel, "Sp-j48: a novel optimization and machine-learning-based approach for solving complex problems, special application in software engineering for detecting code smell," *Neural Computing Application*, Vol. 32, pp. 7009-7027, 2020. DOI: https://doi.org/10.1007/s00521-019-04175-z.

[12] F. A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based System*, Vol.128, pp. 43-58, 2017. DOI: https://doi.org/10.1016/j.knosys.2017.04.014

[13] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, Q, "Machine Learning Techniques for Code Smell Detection, A Systematic Literature Review and Meta-Analysis," *Information & Software Technology*, Vol. 18, pp. 115-138, 2019. DOI: DOI: https://doi.org/10.1016/j.infsof.2018.12.009.

[14] A. Jesudoss and S. Maneesha, "Identification of code smell using machine learning," in *Proc of International Conference Intelligence Computing Control Syst. (ICCS)*, pp.54-58, 2019. DOI: https://doi.org/10.1109/ICCS45141.2019.9065317.

[15] T. Lin, X. Fu, F. Chen, and L. Li, "A novel approach for code smell detection based on deep leaning," *EAI International Conference on Applied Cryptography in Computer and Communications*, Springer, pp. 171-174, 2021. DOI: DOI: https://doi.org/10.1007/978-3-030-80851-8_16.

[16] D. Cruz, A. Santana, and E. Figueiredo, "Detecting bad smell with machine learning algorithms: an empirical study," in *Proc of the 3rd International Conference on Technical Debt.*, pp.31-40, 2022. https://doi.org/10.1145/3387906.3388618.

[17] S. Wang, Y. Zhang, and J. Sun, "Detection of bad smell in code based on bp neural network,*" Computer Engineering*, Vol. 46, pp. 216-222, 2021. DOI: https://doi.org/10.1117/12.2561197.

[18] F. Pecorelli, Di Palomba, D. Di Nucci, D., and A. De Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *Proc of IEEE/ACM 27th International Conference*

on Program Comprehension (ICPC), pp. 25-26 May 2019. DOI: https://doi.org/10.1109/ICPC.2019.00023.

[19] B. H. Choi, Y. -J. Song, and J. -H. Won, "Design of a KPS Civil Signal Candidate Simulator Using a Fully-Reconfigurable GNSS Signal Generator," in *Proc of IPNT Conference*, pp. 23-27, Nov 2-4, 2022. *http://ipnt.or.kr/2022proc/14.*

[20] N. Maneerat and P. Muenchaisri, "Bad-smell prediction from software design model using machine learning techniques," in *Proc of Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pp. 331-336, May 2011. DOI: https://doi.org/10.1109/JCSSE.2011.5930143.

[21] J. Wang, J. Chen, and J. Gao, "Ecc multi-label code smell detection method based on ranking loss," *Journal of Computer Research and Development*, Vol. 58, pp. 178-188, 2021. DOI: https://doi.org/10.7544/issn1000-1239.2021.20190836.

[22] I. Sim, J. Jeong, S. Yun, Y. Lim, and J. Seo, "A Survey for Vulnerability Attack and Defense Method of Satellite-Link Based Communication System," *International Journal of Internet, Broadcasting and Communication*, Vol. 15, No. 4, pp. 128-133, 2023. http://doig.org/10.7236/IJIBC.2023.15.4.128.