A Novel Approach to Identify Security Controls in Source Code

Ahmet Okutan, Ali Shokri, Viktoria Koscinski, Mohamad Fazelinia, Mehdi Mirakhorli

Abstract—*Secure by Design* has become the mainstream development approach ensuring that software systems are not vulnerable to cyberattacks. Architectural security controls need to be carefully monitored over the software development life cycle to avoid critical design flaws. Unfortunately, functional requirements usually get in the way of the security features, and the development team may not correctly address critical security requirements. Identifying tactic-related code pieces in a software project enables an efficient review of the security controls' implementation as well as a resilient software architecture. This paper enumerates a comprehensive list of commonly used security controls and creates a dataset for each one of them by pulling related and unrelated code snippets from the open API of the `StackOverflow` question and answer platform. It uses the state-of-the-art NLP technique Bidirectional Encoder Representations from Transformers (BERT) and the Tactic Detector from our prior work to show that code pieces that implement security controls could be identified with high confidence. The results show that our model trained on tactic-related and unrelated code snippets derived from `StackOverflow` is able to identify tactic-related code pieces with F-Measure values above 0.9.

Index Terms—Security tactics, NLP, BERT

1 Introduction

Secure by Design has become the mainstream development approach to ensure the security and privacy of software systems. During the software development life cycle, security requirements are identified early and addressed from the ground up with a robust architectural design. A secure design often relies on well-known *security controls* [1], defined as reusable techniques to achieve specific quality concerns. These controls provide solutions to enforce *authentication*, *authorization*, *confidentiality*, *data integrity*, *privacy*, *accountability*, *availability*, *safety* and *non-repudiation* requirements, even when the system is under attack [2].

Architectural security controls need to be identified in the early stages of the design process and then implemented alongside functional features [3]–[6]. However, in practice, this is not always the case, as architectural solutions often evolve during the development process [7], [8]. For example, the analysis of the source code of the *Wavelet Enterprise Management Portal (EMP)* [1], a large open-source enterprise resource planning and customer relationship management system, shows that its initial release included a module to edit sensitive personal information including social security number and credit limit. However, no *Audit Trail* control was provided until the system was refactored in the version 6 release. This happened due to developers' primary focus on implementing the functional requirements, which lead to ignoring architecturally significant security requirements either intentionally or accidentally and not incorporating appropriate security controls to satisfy quality concerns in a timely manner.

In practice, many software projects fail to fully document security-related architectural design decisions, providing only high-level lists of decisions, if any. Existing approaches generally document rationales at the design level, and fail to provide explicit traceability down to the code level in which the security controls have been implemented [9]. Therefore, the state of the practice solutions provide only limited support for keeping developers informed of underlying security design decisions during the code maintenance process.

To address this problem, we present an automated and *just-in-time* reverse engineering solution to detect architectural security controls implemented in the Java language. This solution identifies security controls incorporated by designers and pinpoints modules and source files that implement them. This will help developers to understand underlying security design decisions as they sustain a software product [10]. We use the Bidirectional Encoder Representations from Transformers (BERT) [11] and the Tactic Detector from our earlier work [10] to identify security control-related code snippets and map them to a security control catalog to help developers have a better understanding of the security design decisions implemented in their code.

Experiment results indicate that code snippets that implement security tactics can be identified with F-Measure values up to 0.98. The main contributions of this paper are:

- This paper defines three high-level security control categories, *i.e.*, *Detect*, *Prevent*, and *React* and compiles a comprehensive list of commonly used security controls to fill each category.
- It presents and evaluates a novel methodology and algorithms to identify security control-related code snippets in a given software system. A state-of-the-art classification method is used to identify code

• *All authors are with the Department of Software Engineering, Rochester Institute of Technology, Rochester, NY, 14620. (email: ahmet.okutan@gmail.com; as8308@rit.edu; vk2635@rit.edu; mf8754@rit.edu; mxmvse@rit.edu)*

1. https://wavelet.net/erp-emp-training/

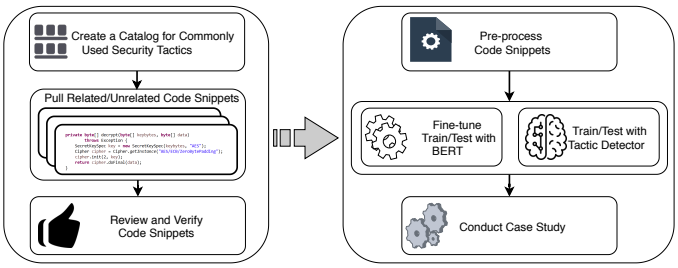

FIG. 1: An overview of the tactic selection, data set generation, and tactic classification processes used in the paper.

snippets related to security controls with high Precision, Recall, and F-Measure values. To the best of our knowledge, no *transformer models* were used to classify security tactics before this study.

- Finally, we provide an online appendix that includes our security control catalog, a *replication package* containing our dataset and scripts used to fine-tune, train and test our models, evaluation pipelines, and a supplementary reference data: **https://github.com/SoftwareDesignLab/security_tactics**

The rest of the paper is organized as follows: Section 2 reviews prior work about identifying design patterns and architectural tactics from the source code and provides a brief overview of the evolution of NLP methods and the BERT algorithm. Section 3 explains the approach used to select commonly used security tactics and review derived code snippets for each security tactic. Section 4 presents the experiment results with a discussion of potential practical use cases. Section 5 discusses threats to the validity of this work, and finally, Section 6 provides the concluding remarks.

2 PRIOR WORK

Design patterns provide reusable solutions for commonly encountered problems during the system and object design activities. The use of design patterns helps to enhance the quality of the source code, thereby improve its resilience to architectural flaws. Detecting design patterns [12] implemented in source code is useful for applications like reverse engineering, software maintenance, system comprehension, and documentation [13], [14]. Over the years, researchers have developed different approaches to detect design patterns in the source code. These approaches mainly consist of two phases, *i.e.*, the feature extraction and the design pattern detection. In the feature extraction phase, high-level abstract features are extracted from the source code or its intermediate representation, where intermediate representations could be an Abstract Syntax Tree (AST) [14], [15], Abstract Semantic Graph (ASG) [16] or a set of source code metrics [17]. Extracted features capture essential attributes of the underlying design patterns and are used in the pattern detection phase to identify whether a design pattern is implemented or not. Leveraging the most recent state-of-the-art architectures of NLP, this work generates context-aware vector representations from the code snippets and then uses them to train Machine Learning models to identify tactic related code pieces.

Architectural security controls play a key role in addressing reliability, performance, and security-related issues in software systems. Prior works have shown that the discovery of architectural security controls from the source code is possible using a variety of Machine Learning methods. Mirakhorli and Cleland-Huang [10] use a customized classifier trained on code snippets extracted from fifty open source software systems and show that commonly referred architectural security controls representing security, performance, and reliability aspects can be identified with recall rates above 70 percent. Another study [18] uses Machine Learning methods and lightweight structural analysis on 10 open source projects and achieves F-Measure values ranging between 0.59 and 0.96 while identifying heartbeat, scheduling, resource pooling, authentication, and audit trail related code snippets. Drawing inspiration from previous works in the field, this paper focuses specifically on the security domain, and trains the state-of-the-art text classification method BERT on the code snippets derived from Stack Overflow to show that security-related code snippets can be identified with relatively high accuracies.

Traditional token-based (bag-of-words) methods that have had difficulty in differentiating text pieces with different ordering of tokens have evolved to superior language models that use self-attention to learn better language representations [19]. Similar to the use of word embeddings on regular text [20], there is an increasing interest in learning code embeddings using vector representations of code-snippets [21]. Word embeddings have revolutionized NLP, leading to the development of pre-trained language models [11], [22], [23] that have achieved state of the art results in many NLP tasks. Similarly, effective techniques have been developed to learn code embeddings. These embeddings have been shown to capture semantic properties of the code-snippets and provide promising results for method and class name prediction [24], [25], semantic code search [26]–[28], code generation [29], code summarization [30], and several other tasks [31]. Building upon the evolutionary improvements in the NLP domain so far, this work proposes to use Bidirectional Encoder Representations from Transformers (BERT) to learn source code representations from code snippets, and then, use the learned representations as features to detect whether a security tactic is implemented or not.

2.1 Bidirectional Encoder Representations from Transformers (BERT)

Language models are trained on large data sets to learn word embeddings that can be used for various tasks. Word embeddings were popularized by Word2vec [32] and Global Vectors for Word Representation (Glove) [33]. These unidirectional models were trained on a large unlabelled corpus to learn representations that capture the semantic similarities between words. One limitation with these approaches is that they learn a single representation for each word in the corpus irrespective of the context in which the word is used. Embeddings from Language Models (ELMo) [34] tries to solve this problem by generating contextual embeddings. ELMo trains a deep bidirectional LSTM on the language modeling task to embed the surrounding context of a word

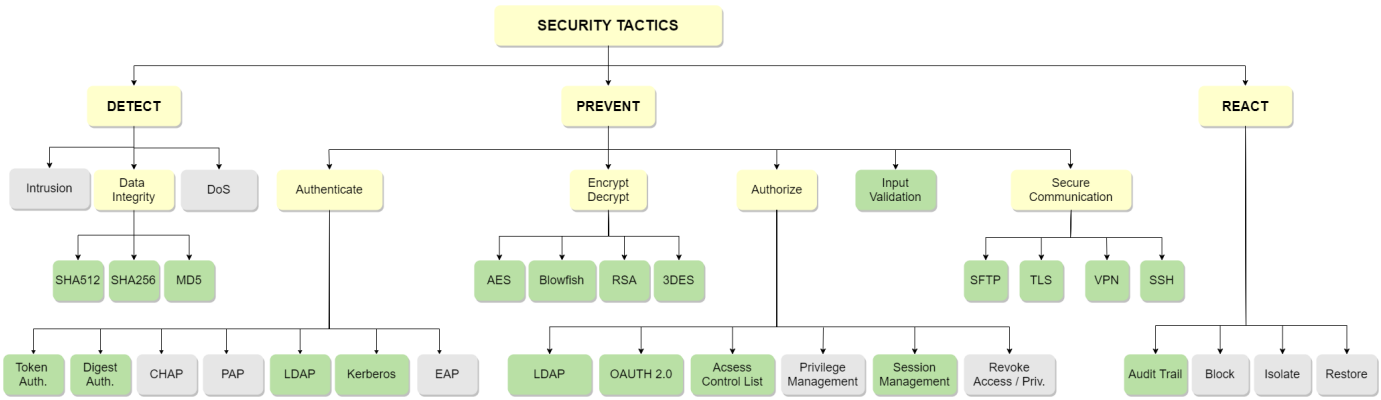

FIG. 2: Categorization of studied security controls. Yellow nodes show tactic categories. Green nodes represent security controls that have at least 25 related code snippets in Stack Overflow. Gray nodes represent security controls that had less than 25 related code snippets and are not included in experiments.

in its embedding. Then, the pre-trained ELMo language model can be used in other models that process text. This approach has achieved state-of-the-art results in many NLP tasks. However, the use of pre-trained language models is limited to the learned word embeddings. These word embeddings are used as fixed-parameters and the model needs to be trained from scratch for each task-specific dataset. There was no effective transfer learning technique to fine-tune these models on a different task. Universal Language Model Fine-Tuning (ULMFiT) [35] solves this problem by introducing a language model fine-tuning strategy to perform transfer learning. OpenAI's GPT-2 [36] replaces the LSTM blocks in language models with the decoder blocks from the Transformer model [19] and enables the language model to take advantage of the self-attention mechanism to learn better representations.

BERT [11], a pre-trained language model that can be fine-tuned to learn tasks with a fair amount of data, builds upon the recent developments in natural language processing. A transformer encoder [19] is bi-directionally trained using `Wikipedia` (https://wikipedia.com) and the `Book Corpora` (https://yknzhu.wixsite.com/mbweb).

BERT is pre-trained on two tasks:

- *Masked Language Modeling*: 15% of input tokens are randomly replaced by the [MASK] token and then the model attempts to predict the original token from the final output embeddings corresponding to those masked tokens. This enables the model to learn relationships between words within a sentence.
- *Next Sentence Prediction*: In next sentence prediction, the model is given two sentences A and B and it attempts to predict whether sentence B is the subsequent sentence in the corpus. The model uses the final output embedding corresponding to the first input token [CLS] for next sentence prediction. This enables the model to learn relationships between two sentences.

BERT has achieved state-of-the-art results on eleven NLP tasks [37]. It has been shown that pre-training enables the model to learn better contextual embeddings. To use BERT for downstream tasks, it is fine-tuned on a new labelled task specific dataset and then used to solve problems like question-answering and classification.

Larson *et al*. use BERT and a set of additional classifiers including Support Vector Machine (SVM), Multi Layer Perceptron (MLP), and Convolutional Neural Network (CNN) to analyze intent classification and out-of-scope prediction methods and state that BERT achieves the best in-scope accuracy, scoring 96% or above even when the training data is limited [38]. Another study [39] used part of the data set introduced by Larson *et al*. to compare the performance of BERT with simple LSTM models where the training data set included 15,101 entries for 150 intents (around 100 instances for each class). The results suggest that when less than 50% of the training data is used (around less than 75 training instances for label), the performance of the BERT classifier gets worse, compared to a simple LSTM architecture with one bidirectional layer and one unidirectional layer with 50 neurons in each layer.

3 METHODOLOGY

This paper studies a list of security controls that are widely used by the software community. Inspired by the NIST Cybersecurity framework [40], we define a high-level abstraction and categorize security controls into different groups according to the functions they are used for. Section 3.1 explains how security controls are selected and categorized, Section 3.2 summarizes how the data set for each tactic is generated and manually reviewed. Finally, Section 3.3 explains how the code snippets are pre-processed, BERT models are fine-tuned, longer code snippets are treated with mean pooling, and BERT's output sequences are classified. Figures 3 and 4 provide a high-level overview of these processes explained in Sections 3.1, 3.2, and 3.3.

3.1 Tactic Selection

Security controls are selected to cover different functions defined by the NIST Cybersecurity framework about detecting, preventing, and reacting to a cybersecurity incident [40]. The controls are grouped into different categories based on their purpose of use. The categorization tree in Figure 2 shows all studied categories and controls. A short description of each category and tactic is provided below:

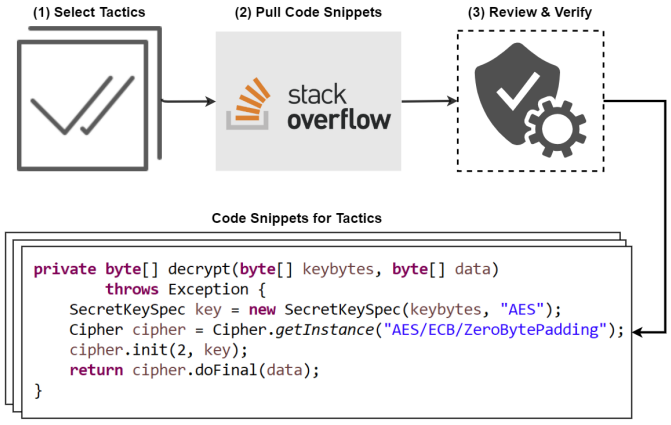

FIG. 3: An overview of the data set generation process.

3.1.1 Detect

Detect is an abstract category that includes controls used to detect if a system or data has been compromised. The main focus is to detect anomalous cybersecurity events and make sure that protective measures are in effect. Security controls in this category are divided into three subcategories:

- **Intrusion:** Intrusion controls monitor intrusion attempts, log information about them, and warn system administrators to take appropriate protective measures to address security issues.
- **Data Integrity:** Data Integrity ensures that transmitted data is reliable and consistent throughout its life-cycle. A well-known approach to achieve data integrity is through Check-summing [41], [42]. A checksum of the data is calculated before and after the data is transmitted and the two check sums are compared to ensure that the data has not been modified. The most commonly used secure hash and checksum algorithms are SHA256, SHA512 [43], and MD5 [44].
- **Denial Of Service (DoS):** This tactic monitors the system activities for Denial of Service attacks. Denial of service attacks flood the system with traffic and make it inaccessible to legitimate users [45]. Denial of service attacks are handled by dropping illegitimate traffic and making the system available for legitimate users [46].

3.1.2 Prevent

Prevent category is defined to include controls used to ensure that a system is protected from cybersecurity incidents. This category is further divided into five different sub-categories, each representing a different aspect of the protection:

- **Authenticate:** Security controls in this category validate the identity of a user or a system. Commonly used examples include controls used for user authentication (Token Authentication and Digest Authentication), and network authentication (Kerberos and Light Weight Directory Access Protocol (LDAP)). Password Authentication Protocol (PAP), Challenge-Handshake Authentication Protocol (CHAP), and Extensible Authentication Protocol (EAP) are other methods that are used for authentication.
- **Encrypt/Decrypt:** Security controls in this group are used to secure data by transforming it into a format that cannot be read by eavesdroppers. Widely used encryption methods include Advanced Encryption Standard (AES), Blowfish, Rivest-Shamir-Adleman (RSA) and Triple Data Encryption Algorithm (3DES).
- **Authorize:** This category includes controls used to validate access to system resources. Commonly used authorization controls include LDAP, OAuth 2.0, Access Control List, Privilege Management, Session Management, and Revoke Access/Privilege.
- **Input Validation:** Input Validation is used to make sure that only properly formed and sanitized secure input is allowed to enter into a system.
- **Secure Communication:** We define a new category named 'Secure Communication' to represent security controls that are used to establish end to end secure connections. Commonly used controls in this category are SSH File Transfer Protocol (SFTP), Transport Layer Security (TLS), Virtual Private Network (VPN) and Secure Shell (SSH).

3.1.3 React

This branch includes security controls used to respond or react to a cybersecurity incident. Once a cybersecurity event is detected, it is necessary to confine the impact of the event and restore the system capabilities that have been affected as soon as possible. An ideal cyber incident response should minimize the effects of the attack and find and fix its root cause to prevent future incidents. This tactic category is further divided into four sub-categories each representing a different task:

- **Audit Trail:** Audit Trail keeps records of the sequence of system events and user activities and these records can be used to identify malicious users, analyze security problems and detect violations.
- **Block:** Cyber incidents can be contained by blocking known malware sources, *i.e.*, domains or email addresses, closing related ports, tuning firewall filters to restrict access to certain software services that might be more vulnerable.
- **Isolate:** It is part of the containment process that makes sure that the effects of a cyber incident are confined, and damaged areas are isolated, with the incident investigated and the root causes identified.
- **Restore:** Restore controls ensure that software systems or assets that are affected from a cybersecurity incident are adequately restored after the incident.

3.2 Dataset Generation

This section explains how the code snippets for each security tactic are queried from Stack Overflow, how pulled code snippets are reviewed, and what type of security functions are included in each tactic data. An overview of these processes is shown in Figure 3.

The data set for each security tactic is generated by extracting tactic related and unrelated code snippets from Stack Overflow. The advanced search API (https://

api.stackexchange.com/docs/advanced-search) takes a free form text parameter and returns a list of questions. Stack Overflow matches the parameter with all question properties and returns the most relevant questions and answers. The questions and the topics tagged to them are curated and verified by the community, therefore we assume that the questions and the topics that they are tagged with are relevant. The advanced search API's underlying algorithm is undocumented; however, returned results show that it does not perform an exact string match only. For instance, we use "3DES" as the search parameter to search for questions related to the Triple DES tactic. However, returned questions include discussions that talk about 3DES, Triple DES and DESede. To get the maximum number of questions, we test multiple queries related to each tactic and choose the query that gives the highest number of relevant results. For example, for tactic Advanced Encryption Standard (AES) we tried 'advanced encryption standard' and 'aes' and found that 'aes' gives the most relevant results. The keywords used to query each security tactic are listed in Table 1.

As shown in Figure 3, after all of the questions are pulled for each security tactic, they are manually reviewed to ensure that the code-snippets associated to the questions are related to the queried tactic. To make sure that we have enough training samples for the training phase, we ignore security tactics that have lower than 25 related code snippets. Prior research [47] shows that a set of issues are observed in Stack Overflow answers including mismatched solutions, outdated solutions, incorrect solutions, and buggy code. Our finding, which is in line with this work, shows that although a sufficient number of questions are found for a security tactic and all pulled questions are associated with a variety of topics related to the tactic, the code-snippets included in the questions may not be related at all. Therefore, such code snippets are filtered out from the tactic related code snippets manually during the data review and verification process (, *i.e.*, step 3 of Figure 3). In general, pulled code-snippets implement a wide variety of functions related to a security tactic. For instance, LDAP includes code pieces that implement LDAP authentication or query LDAP server for user or directory information. Similarly, Kerberos code snippets provide implementations of Kerberos authentication with three different frameworks. Below we provide a summary of the tasks (functions) implemented by the code-snippets included for each security tactic.

3.2.1 Detect

Under the *Detect* category, we did not find enough samples for the **Intrusion and Denial of Service (DoS) controls.** For SHA-256, SHA-512 and MD5 message-digest controls, pulled code snippets include functions to hash data with MessageDigest (in the java.security package) or generate Hash-based Message Authentication Code (HMAC) SHA256 signature using the Java Cryptography Architecture (JCA) framework [48].

3.2.2 Prevent

There are not enough samples for CHAP, PAP and EAP controls under 'Authenticate' category. A summary of tasks included for other controls in this category are:

- **Digest Authentication:** Code pieces that implement Spring Security Digest Authentication [49] and Digest-MD5 authentication [50].
- **Kerberos:** Code snippets that implement Kerberos authentication using an extension of the Spring Security named Kerberos Spring Security [51], Java Authentication and Authorization Service (JAAS) Krb5LoginModule [52], and the Generic Security Service API [53]. Some of the code snippets include functions that use Kerberos to connect to Hadoop File System (HDFS).
- **Lightweight Directory Access Protocol (LDAP):** Functions that perform LDAP authentication and query LDAP server for user and directory information.
- **Token based authentication:** Code snippets that implement JSON Web Token (JWT) [54] authentication using the Spring Security framework.

For the 'Encrypt/Decrypt' category, a sufficient number of code snippets was found for each of the four controls. The functions included for each of these four controls are:

- **Advanced Encryption Standard (AES):** Encrypt or decrypt a password, message or file content using the Java Cryptographic Architecture (JCA) framework [48].
- **Blowfish:** Encrypt or decrypt password, message or file content using the JCA framework.
- **Rivest-Shamir-Adleman (RSA):** Generate keys using the Java KeyPair class, and encrypt or decrypt a password, message or file content using the JCA framework.
- **Triple DES:** Encrypt or decrypt a password, message or file content using the JCA framework.

Under the 'Authorize' category, there was not a sufficient number of samples for 'Session Management' and 'Revoke Access/Privilege' controls. The functions in the other four controls are:

- **LDAP:** Perform LDAP authentication and query LDAP server for user and directory information.
- **OAuth 2.0:** Implement Oauth 2.0 using the Spring Security OAuth project [55], access server resources with an access token.
- **Access Control List (ACL):** Implement access control using Spring Security ACL services [56] or Apache ZooKeeper [57].
- **Session Management:** Manage user session with Spring Session [58], manage database session with Hibernate Session.

For 'Input Validation', we include code snippets that validate arguments with Hibernate Validator and network requests with Spring Web Services Interceptor. For the 'Secure Communication' category, the functions included in the code snippets of each tactic are:

- **SSH File Transfer Protocol (SFTP):** Access remote directory using Java Secure Channel [59], download remote files and upload files to a server.
- **Transport Layer Security (TLS):** Establish a TLS connection between client and server and perform certificate authentication.

Tactic	Query Parameter	Sample Size	S1	S2	S3	S4	SMax
SHA512	sha512	100	98%	100%	-	-	2
SHA256	sha256	250	98.80%	100%	-	-	2
MD5	md5	400	98%	99.50%	100%	-	3
Token Authentication	token authentication	200	98%	100%	-	-	2
Digest Authentication	digest authentication	100	96%	99%	100%	-	3
LDAP	ldap	400	99%	100%	-	-	2
Kerberos	kerberos	150	98%	100%	-	-	2
AES	aes	400	98%	100%	-	-	2
Blowfish	blowfish	100	100%	-	-	-	1
RSA	rsa	400	96.5%	100%	-	-	2
3DES	3des	100	98%	100%	-	-	2
OAuth 2.0	oauth2	400	94.5%	99.75%	100%	-	3
Access Control List	acl	150	100%	-	-	-	1
Session Management	session management	350	97.71%	99.71%	99.71%	100%	4
Input validation	validation interceptor	50	98%	100%	-	-	1
SFTP	sftp	400	97.75%	99.75%	100%	-	3
TLS	tls	400	96.5%	99.75%	100%	-	3
VPN	vpn	50	96%	100%	-	-	2
SSH	ssh	400	99.5%	99.5%	99.5%	100%	4
Audit trail	audit trail	200	98.75%	100%	-	-	2

TABLE 1: The summary of the tactic data sets used. S1, S2, S3, and S4 columns represent the percentage of code snippets that are divided into 1, 2, 3, and 4 sequences for BERT, respectively. The SMax column represents the number of sequences that the longest input is divided into.

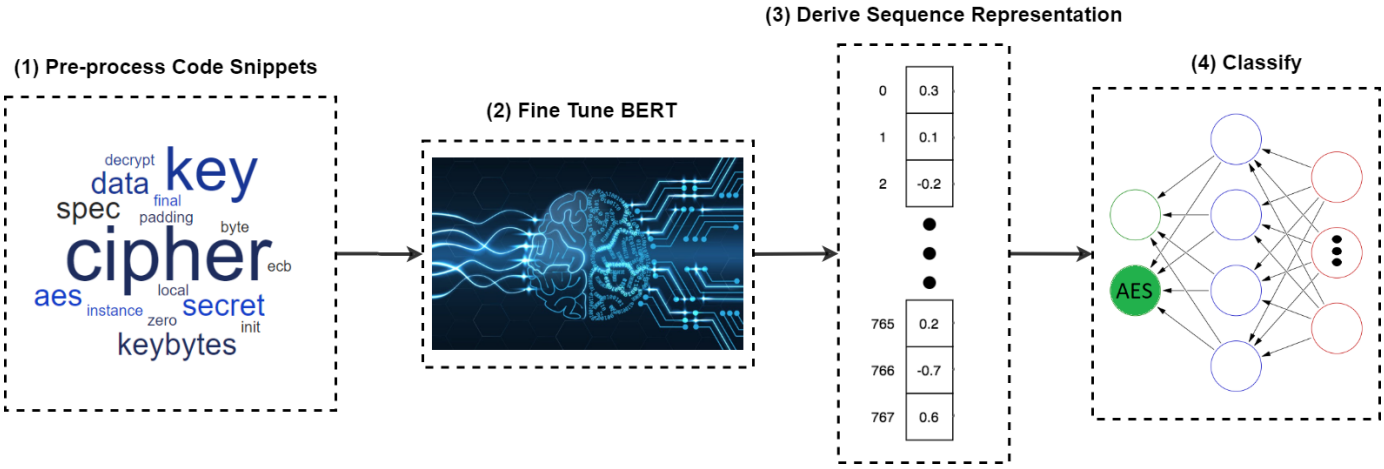

FIG. 4: Pre-process code snippets, fine-tune BERT and classify sequence representations.

- **Virtual Private Network (VPN):** Build VPN client using Android VPN Service and monitor network data.
- **Secure Shell (SSH):** Establish SSH connections using the Java Secure Channel framework and Java SSH client.

3.2.3 *React*

There are not enough code snippets for Block, Isolate and Restore controls in this category. The tasks included for the 'Audit Trail' tactic are used to implement logging, transaction management, audit via the Hibernate Envers module [60] and the Object Auditing and Diff Framework for Java (JaVers) [61].

In addition to the tactic related code snippets, we include code snippets that are not related to the security controls. Unrelated code-snippets are extracted from a set of 50,000 questions with 'Java' tag. We use the generic keyword 'Java' to make sure that the questions are general and not biased towards any topic in particular. We also review the code snippets and verify that related and unrelated questions do not have any tags in common.

Table 1 provides the attributes of the final datasets for each security tactic. The query parameter column lists the keywords used to query code snippets for each tactic. The sample size column provides the total number of related and unrelated code snippets in each dataset after the manual review process. In order to make sure that the number of samples for each security tactic is normalized, we create sample size categories for the number of related code snippets ranging between 25 and 200, with increments of 25. For example, the number of related code snippets found for SHA512 was 56, however only 50 randomly selected code snippets are used. Similarly, the number of code snippets for SHA256 was 139, but only 125 randomly selected samples are included. The training data for each security tactic is generated by merging an equal number of tactic related and unrelated code-snippets to have a balanced class distribution. For instance, for SHA512 and SHA256 security controls, there are 50 and 125 related samples respectively.

Since the number of unrelated samples are equal to related samples, the total number of samples for SHA512 and SHA256 is 100 and 250, respectively. The columns S1, S2, S3 and S4 represent the percentage of samples that are represented with 1, 2, 3, and 4 BERT input sequences. The SMax column shows the number of sequences that the longest code snippet in a dataset is divided into.

3.3 Pre-processing Code Snippets and Fine-tuning BERT

The process pipeline used to pre-process code snippets, fine-tune BERT, and classify its output sequences is shown in Figure 4. Once the training data is pulled and manually reviewed for each security tactic, we perform a number of standard NLP pre-processing steps to make sure that all code snippets are tokenized, normalized and cleaned properly before BERT models are fine-tuned. After code snippets are tokenized, we split all camel case and snake case words, and convert all code-snippets to lower case for normalization. To clean the code snippets, all punctuations, numbers, short words less than two characters, and long words with more than 25 characters are removed. We also remove keywords of the Java language, because these keywords are common for all security controls and do not help to understand the security functionality of a code snippet.

The distribution of the length of the code snippets might be different for each tactic data set. The BERT algorithm can process input sequences with a maximum length of 510 tokens, and a special treatment is needed when the number of tokens is greater than 510 to make sure all tokens in the input sequences are considered. Figure 5 shows the distribution of the number of tokens for the security controls that include relatively larger code snippets. Although the majority of the observed token lengths are less than 510, these security controls still include some code snippets that have more than 1000 or even 1500 tokens. In order to make sure that the methodology used in this paper is applicable to larger code snippets as well, we apply mean pooling and make sure that calculated means represent the average pooled representation of the entire input sequence in a code snippet.

Out of the maximum 512 input tokens that can be accepted by BERT, two tokens are reserved for special tokens, *i.e.*, CLS and SEP. Therefore, an input sequence can only be of 510 tokens, and extra tokens after the first 510 tokens are ignored by BERT. We divide large input sequences into multiple sequences of 510 tokens. If N is the number of tokens in a given code-snippet, then we divide the code-snippet into $m = N/510$ sequences. We get the BERT representation of the m sequences and use mean pooling to combine them to get the final representation for the entire input code-snippet.

3.4 Tactic Classifiers

We fine-tune the *bert-base-uncased* model [62] with the tactic datasets explained in Section 3.2. The base model has 12 layers of transformer encoder blocks, 768 hidden units and 12 self-attention heads. It is fine-tuned for code-snippet classification and used to predict whether a code snippet is related to a particular security tactic or not. Each BERT

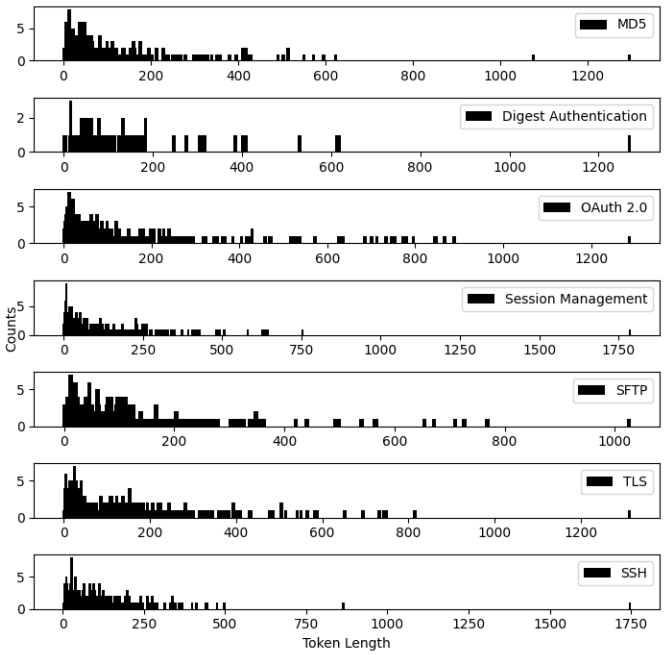

FIG. 5: The distribution of the token lengths of code snippets for security controls that have 3 or 4 in the SMax column in Table 1

model is trained for 10 *epochs* with a *batch size* of 16, *learning rate* of 2e-5, and a maximum sequence length of 512. A classifier with one hidden layer (including 32 neurons) and an output layer is added on top of BERT and the output vector corresponding to the [CLS] token is used to classify each input sequence. This vector is the average pooled representation of the entire input sequence and hence can be used for classification. The weighted sum of the output vector corresponding to the [CLS] token is calculated and passed through the classifier and the Softmax function is used to get the final probability indicating whether a tactic is implemented or not. During fine-tuning, all BERT parameters along with the classification layer parameters (W and B) are updated to maximize the log-probability for the predicted tactic.

To classify security tactics with an alternative method, we use the Tactic Detector (TD) introduced in our previous work [10]. TD is a customized version of the information retrieval method developed to automate the detection and classification of non-functional requirements (NFRs) [63]. In our previous work, using code snippets derived from 50 open source systems, we trained seven off-the-shelf classifiers including Support Vector Machine (SVM), C.45 Decision Tree (DT), Bayesian Logistic Regression (BLR), AdaBoost, rule learning with SLIPPER, Bagging, and Voting. We used these classifiers and the TD method to detect tactics in the Hadoop Distributed File System. The customized TD method outperformed other classifiers and detected tactic types with recall rates greater than 0.7. Leveraging findings of our prior work, this paper included TD as an alternative tactic detection method to classify the the same set of pre-processed code snippets used for BERT.

4 Experiments and Results

This paper uses BERT and TD as tactic classification methods and designs three sets of experiments and a case study to evaluate the performance of the proposed methodology:

- **Experiment 1 :** Related and unrelated code snippets of each tactic are used to predict whether a source file is tactic-related or not.
- **Experiment 2 :** Related and unrelated code snippets within the same category are merged to predict whether a code piece is tactic-related. We merge all controls under the yellow branches in the second level of the categorization tree in Figure 2, *i.e.*, Data Integrity, Authenticate, Encrypt/Decrypt, Authorize and Secure Communication.
- **Experiment 3 :** This is a multi-label classification experiment, where the tactic names in Experiment 2 are used as class labels. Only related code snippets from each tactic (within the same category) are combined to form the datasets for this experiment.
- **Case Study :** To evaluate the performance of the proposed methodology on a real life project, we conduct a case study on a large scale open source system, `Liferay portal` (https://github.com/liferay/liferay-portal), that is implemented in the Java language.

TABLE 2: Precision (P), Recall (R) and F-Measure (F) values obtained with BERT and TD during Experiment 1. The Sample Size (SS) column gives the total number of training code snippets used for each tactic.

		BERT			TD		
Tactic	**SS**	**P**	**R**	**F**	**P**	**R**	**F**
SHA256	500	1	0.99	**0.99**	1.00	0.93	0.96
SHA512	200	0.98	0.98	0.98	1.00	0.96	0.98
MD5	800	0.99	0.99	**0.99**	0.99	0.98	0.98
Token Authentication	400	0.96	0.96	**0.96**	0.99	0.84	0.91
Digest Authentication	200	0.93	0.96	**0.95**	1.00	0.82	0.90
LDAP	800	1	0.99	**0.99**	1.00	0.83	0.90
Kerberos	300	0.98	1	**0.99**	1.00	0.72	0.84
AES	800	0.96	0.91	**0.94**	0.98	0.81	0.88
Blowfish	200	0.96	0.96	0.96	1.00	0.98	**0.99**
RSA	800	0.99	0.99	**0.99**	0.97	0.95	0.96
3DES	200	0.93	1	0.97	1.00	0.96	**0.98**
OAuth 2.0	800	0.97	0.99	**0.98**	0.99	0.91	0.95
Access Control List	300	0.96	0.96	**0.96**	1.00	0.76	0.86
Session Management	700	0.97	0.97	**0.97**	0.99	0.86	0.92
Input Validation	100	0.86	1	**0.92**	0.94	0.68	0.79
SFTP	800	0.99	0.98	**0.98**	1.00	0.87	0.93
TLS	800	0.98	0.96	**0.97**	1.00	0.79	0.88
VPN	100	0.86	0.9	**0.88**	0.94	0.60	0.73
SSH	800	0.97	0.98	**0.97**	0.99	0.85	0.91
Audit trail	400	0.98	0.99	**0.99**	0.99	0.79	0.87

Precision, Recall and F-Measure metrics are used to evaluate the performance of each model. The Precision metric represents the fraction of correctly identified tactic-related code snippets within all tactic-related (positive) predictions. Similarly, Recall represents the fraction of correctly identified tactic-related code snippets. We calculate the harmonic mean of Precision and Recall to calculate F-Measure by

$$F - Measure = 2.\frac{Precision.Recall}{Precision + Recall} \quad (1)$$

All experiments are run with a stratified 10-fold cross validation approach. Each tactic dataset is split into 10 folds such that the distribution of the classes is the same in each fold. Each method is trained with nine folds and tested on the remaining fold. Each fold is used as test set, and at the end, the average of test metrics obtained from 10 folds is used to evaluate overall model performance.

4.1 Experiment 1: Binary Tactic Classification

In an enterprise level large software project, it could be very helpful to have an automated system to identify the files, classes or even packages that implement a security tactic, without going through a manual code review process. Given a specific security tactic, being able to generate the list of files or classes where the tactic is implemented gives software professionals a chance to review those files or classes quickly and identify security related architectural weaknesses.

The goal of this experiment is to determine how well the models are able to differentiate between the tactic related and unrelated code snippets for each tactic. Table 2 shows the Precision, Recall and F-Measure scores obtained with BERT and TD for each tactic. For BERT, the F-Measure values range from 0.88 to 0.99 and the average F-Measure for all controls is 0.97. Most models are able to effectively capture the distinctive features in the code snippets, even with small sample sizes. The F-Measure values of TD range between 0.73 and 0.99 and the average F-Measure for all controls is 0.91. Figure 6 shows the comparison of F-Measure values obtained with BERT and TD. In general, we obtain better F-Measure values with BERT, except SHA512, 3DES and Blowfish that have relatively smaller training data sizes. *Mann–Whitney U test* is used to compare the F-Measure values obtained with BERT and TD and the result shows that the difference is significant with a $p-$value of 0.001. Figure 7 shows that except AES, lower F-Measure values are obtained with BERT when the number of training samples is small. The lowest F-Measure values are observed for VPN and Input Validation are 0.88 and 0.92, respectively, and the sample size for these controls is 100.

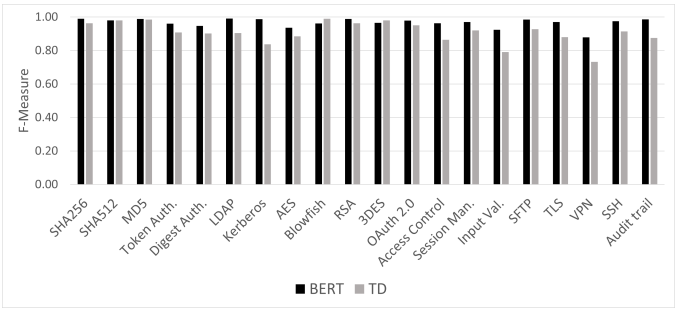

FIG. 6: A comparison of the F-Measure values for BERT and TD in Experiment 1.

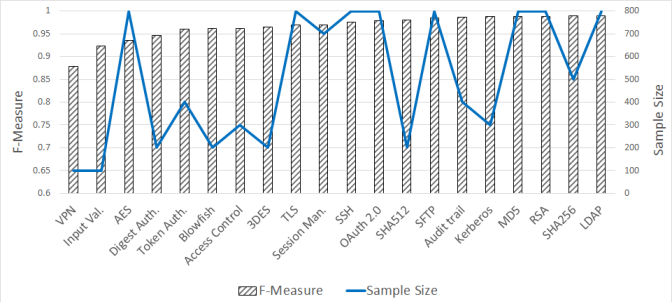

FIG. 7: F-Measure values obtained with BERT and the training data sample sizes in Experiment 1.

Similar to the prior work by Ezen-Can [39], the results of Experiment 1 indicate that in general lower F-Measure values are obtained with BERT when the training data set size is small.

4.2 Experiment 2: Binary Tactic Classification within Category

A large software project that have thousands of files, could potentially have hundreds of files or classes that implement a variety of security controls. In such a case, being able to identify the files or classes that implement one or more (potentially different) security controls is important to help code reviewers focus on the tactic related code pieces while identifying architectural flaws.

TABLE 3: Precision, Recall and F-Measure values for each category in Experiment 2.

		BERT			TD		
Tactic Category	**SS**	**P**	**R**	**F**	**P**	**R**	**F**
Data Integrity	1500	0.98	0.98	**0.98**	1.00	0.82	0.90
Authenticate	1700	0.96	0.97	**0.96**	1.00	0.69	0.82
Encrypt / Decrypt	2000	0.97	0.92	**0.94**	0.97	0.89	0.93
Authorize	2600	0.96	0.98	**0.97**	0.99	0.64	0.78
Secure Communication	2500	0.96	0.97	**0.96**	0.99	0.77	0.87

During this experiment, a category dataset is created by combining the tactic related and unrelated code-snippets of all security controls within the same category. The models are trained with the combined code snippets of each category, and 10 folds stratified cross validation approach is used to determine how well the models differentiate security related and unrelated code-snippets.

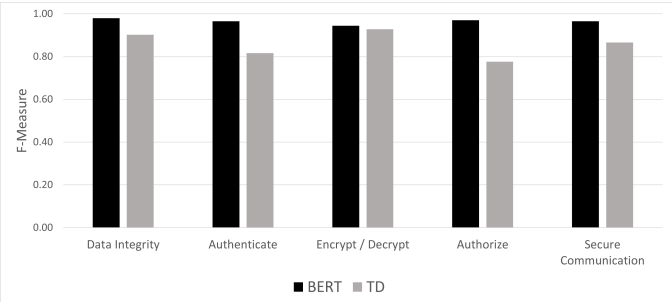

FIG. 8: A comparison of the F-Measure values for BERT and TD in Experiment 2.

Table 3 shows the Precision, Recall and F-Measure scores for each tactic category for BERT and TD methods, and Figure 8 shows the comparison of their F-Measure values. BERT models perform quite well with F-Measure values ranging from 0.94 to 0.98, where the average F-Measure for all categories is 0.96. Similarly, TD achieves F-Measure values ranging between 0.78 and 0.93 with an average F-Measure of 0.86. One reason for observing such higher F-Measure values (compared to Experiment 1) is that related code-snippets of the security controls within the same category have similar implementations. For example, the security controls under Secure Communication, *i.e.*, SFTP, SSH and TLS all include similar steps to establish a connection and then interact with it. SFTP and SSH are both implemented using the Java Secure Channel framework. Similarly, all security controls under the Encrypt Decrypt category are implemented using the Java Cryptography Extension (JCE) framework. Moreover, Digest Authentication, Kerberos and Token Based Authentication in the Authenticate category are all using the Spring Security framework.

4.3 Experiment 3: Multi-label Tactic Classification

Although it is more challenging compared to the cases discussed in Sections 4.1 and 4.2, given a list of security controls and a large software project repository, it could be quite helpful to be able to identify which security tactic is implemented in the files, classes or packages in the repository. That could help to pinpoint the code pieces that implement the security controls of interest and fix architectural problems if any.

Related code snippets within each category (of Experiment 2) are merged to form a dataset for multi-class classification where tactic names are used as class labels. The goal of this experiment is to measure the performance of the fine-tuned BERT models and TD while differentiating controls within the same category. There are two main challenges for this experiment. First, security controls within the same category might use common libraries or frameworks and have quite similar implementations. Therefore, it is more challenging to distinguish code snippets that have very similar implementations of the security controls in the same category. Second, the number of samples for each security tactic within each category is different and the distribution of the classes is imbalanced. For example, under the Encrypt Decrypt tactic category, AES, Blowfish, RSA, and 3DES have samples sizes of 200, 50, 200, and 50, respectively. Similarly, the classes are imbalanced for other categories as well. Models trained with imbalanced data sets might have a tendency to ignore the minority classes and over-fit on the majority ones. Some common methods [64] to balance class distributions are random under sampling, SMOTE [65], and random oversampling. Random under sampling could lead to a performance loss due to smaller overall sample sizes. It is hard to use SMOTE, because generating syntactically and semantically valid code-snippets is not a trivial task. Therefore, we perform random oversampling for the minority classes in each category to have a balanced class distribution for the security controls within the same category.

Table 4 shows Precision, Recall and F-Measure values for each security tactic with BERT and TD. The average F-

TABLE 4: Precision (P), Recall (R) and F-Measure (F) values obtained with BERT and TD during Experiment 3. The Sample Size (SS) column provides the total number of training code snippets for each tactic in each tactic category.

			BERT			TD		
Tactic Category	**Tactic**	**SS**	**P**	**R**	**F**	**P**	**R**	**F**
Data Integrity	SHA512	200	0.9	0.97	**0.93**	0.67	0.03	0.06
	SHA256	200	0.96	0.9	**0.93**	0.58	0.73	0.64
	MD5	200	1	0.99	**0.99**	0.60	0.98	0.74
Authenticate	Token Base Authentication	200	0.79	0.82	0.8	0.80	0.86	**0.83**
	Digest Authentication	200	0.79	0.77	0.78	0.89	0.87	**0.88**
	Kerberos	200	0.81	0.84	**0.82**	0.92	0.71	0.80
	LDAP	200	0.9	0.92	**0.91**	0.90	0.87	0.88
Encrypt / Decrypte	AES	200	0.78	0.76	0.77	0.79	0.75	0.77
	Blowfish	200	0.89	0.99	0.94	0.97	0.94	**0.96**
	RSA	200	0.9	0.95	**0.93**	0.85	0.88	0.87
	3DES	200	0.95	0.9	**0.93**	0.96	0.84	0.89
Authorize	LDAP	200	0.97	0.96	**0.97**	0.98	0.89	0.93
	Oauth 2.0	200	0.92	0.97	**0.94**	0.88	0.92	0.90
	ACL	200	1	0.98	**0.99**	1.00	0.82	0.90
	Session Management	200	0.98	0.95	**0.96**	0.91	0.92	0.91
Secure Communication	SFTP	200	0.86	0.95	**0.9**	0.81	0.83	0.82
	TLS	200	0.96	0.96	**0.96**	0.91	0.75	0.82
	VPN	200	0.99	1	**0.99**	1.00	0.65	0.79
	SSH	200	0.95	0.83	**0.89**	0.77	0.69	0.72

Measure scores of BERT for the Data Integrity, Authenticate, Encrypt / Decrypt, Authorize, and Secure Communication tactic categories are 0.95, 0.83, 0.89, 0.97, and 0.93, respectively and the overall average for the experiment is 0.91. Figure 9 shows the comparison of the F-Measure values obtained for each tactic with BERT and TD. Most F-Measure values obtained with BERT are above 0.9, and BERT outperforms TD during this experiment. *Mann–Whitney U test* test is used to run statistical significance tests on the F-Measure values obtained with BERT and TD and the result shows that the difference between the two sets of F-Measure values is significant with a $p-$value of 0.002.

The results also suggest that the trained models are not that successful while differentiating security controls with very similar implementations. Security controls with lower performance scores tend to have very similar implementations with other security controls in the same category. For example, both SFTP and SSH use the Java Secure Channel framework and have similar implementations, therefore their F-Measure values are lower, compared to the F-Measure values obtained for TLS and VPN. As an example, the confusion matrix of the 10 folds BERT multi-label classifier for the Secure Communication category is shown in Figure 10. The values in the confusion matrix show that SFTP is misclassified as SSH, and similarly SSH is misclassified as SFTP due to the similarity of their implementations. TLS and VPN are more distinguishable from other security controls compared to SFTP and SSH.

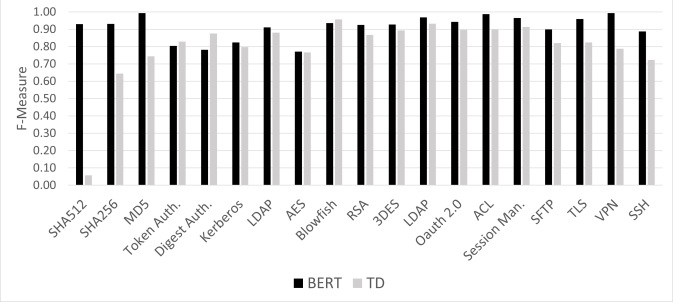

FIG. 9: A comparison of the F-Measure values of BERT and TD in Experiment 3.

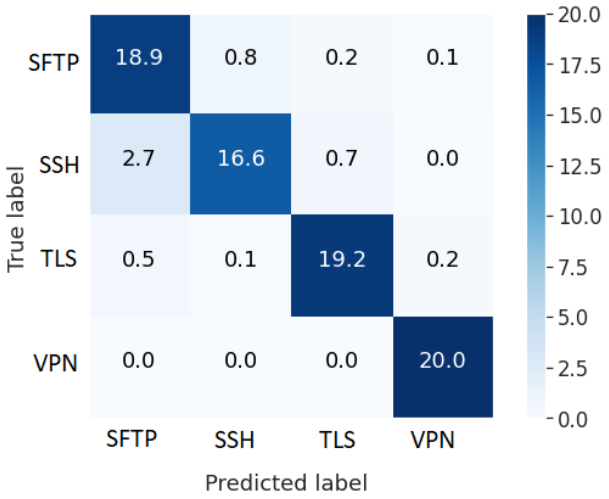

FIG. 10: The average of the confusion matrix entries obtained in each fold while classifying tactics in Secure Communication category.

4.4 Experiment 4: Case Study

To evaluate the performance of BERT and TD in a real-life large-scale project, a case study was conducted on a popular open source project, Liferay[2] portal. Liferay is a well-known and widely used portal, mostly written in Java language. Liferay was chosen for the case study due to its vast application as a popular platform in various domains, including B2B online revenue, customer service, resource management, and system integration. A team of three experts studied over 43K of java files in the Liferay project to identify tactic-related files. During this manual inspection, 210 instances of LDAP, 96 instances of MD5, 84 instances of SHA, and 389 instances of OAUTH2 related

2. https://github.com/liferay/liferay-portal

files were found. The team labeled all files as positive or negative based on their relationship with the mentioned tactics. For instance, 210 files were labeled as LDAP, while the remaining 42790 files were labeled as unrelated to LDAP. To create a test dataset that includes a fair number of related and unrelated test cases, we followed the 20% - 80% ratio between related and unrelated test cases. In that regard, from the files labeled as unrelated to LDAP, the team randomly selected 840 files to be included in the case study. Table 5 presents the sample size of test cases for each tactic under the **SS** column.

During the case study, we investigated the performance of both the Tactic Detector, and the Bert model. We first trained a binary model for each tactic using its data set summarized in Table 1, where 200 files are labeled related (positive) for MD5, LDAP, and OAuth 2.0 and 150 files are marked positive for SHA. Then, we used the manually labeled and reviewed Liferay Java files as the test data set, to measure the performance of each trained model. Before predicting whether a tactic is implemented or used in a file, we performed a set of pre-processing steps. All Liferay Java files include the same *comment* block expressing copyright-related information at the beginning of the file. We removed these comment blocks since they are common to all files and do not provide file-specific information. Other pre-processing steps included tokenizing each file, removing punctuation and Java language keywords, splitting each token based on camel case convention, removing stop words and numbers, and finally, removing lengthy (over 50 characters) and short (less than 2 characters) tokens.

Table 5 demonstrates the results of the experiments during the case study. Based on the recalls shown in this table, BERT model was able to correctly identify 100%, 84%, 65%, and 92% of LDAP, MD5, SHA, and OAUTH2 cases respectively. TD, on the other side, identified 79%, 75%, 54%, and 79% of LDAP, MD5, SHA, and OAUTH2 tactic-related test samples. BERT outperforms TD when identifying tactic-related cases, however based on the recalls of unrelated cases shown in Table 5, TD identifies a higher percentage of unrelated cases. Another measure provided in Table 5 is the precision of the predictions. Although BERT results show a slightly better (overall) precision for unrelated cases, TD has a significantly better precision for the tactic-related test cases.

We investigated the reasons for relatively high rate of false-positives in tactic-related predictions by BERT model. We particularly identified three main drivers. First, the imbalanced nature of the test data could be a driver for high false positives. For instance, there are only 210 LDAP samples whereas the number of unrelated code snippets to LDAP is 840 samples. Based on this distribution of the data, even if the model correctly identifies all the LDAP samples (210 cases) and incorrectly identifies only 20.12% of the unrelated samples (169 cases), the precision of LDAP would be 55%. Second, the existence of security-related terms and keywords in the files that are labeled as *Unrelated* may result in false-positive predictions. For instance, we found "User," "Permission," "Access," and some other security-related terms in the files classified as false-positive. If a file implements a security related feature (even if it is not exactly the tactic under study), there is a chance that the model labels that file as a tactic implementation. Lastly, BERT model needs a large training dataset to be able to be fine tuned [66]. Therefore, training such a model by a dataset of 400 training instances (LDAP, MD5, OAUTH2) or even fewer (SHA), can result in a model that does not perform well when it comes to test cases.

```
15   package com.liferay.portal.util;
 :
29   public class PropsValues {
 :
461          public static final boolean CAS_IMPORT_FROM_LDAP = GetterUtil.getBoolean(
462                  PropsUtil.get(PropsKeys.CAS_IMPORT_FROM_LDAP));
 :
1859         public static final boolean OPEN_SSO_LDAP_IMPORT_ENABLED =
1860                 GetterUtil.getBoolean(
1861                         PropsUtil.get(PropsKeys.OPEN_SSO_LDAP_IMPORT_ENABLED));
 :
```

FIG. 11: Part of a Java file in Liferay implementation module which normally needs manual inspection to be identified as Unrelated to the LDAP tactic.

In general, the F-measure of TD approach shown in Table 5 is relatively better compared to the BERT. However, considering the lower amount of false-negatives for tactic-related predictions of BERT compared to TD, we believe that BERT can filter out source files that do not implement a specific tactic with a higher confidence compared to the TD, thereby decrease the time needed for manual code inspection. The contribution of the BERT could be more significant when a software project is composed of large number of files and an automated method is needed to conservatively rule out the modules or files that do not implement a specific security tactic. For example, Figure 11 shows part of a Java file in the Liferay portal which defines some static values to be used by other classes, but does not include a tactic implementation. For instance, at line 461 and 462, the programmer defines a Boolean variable CAS_IMPORT_FROM_LDAP and assigns a value to it. Furthermore, through the lines 1859 to 1861, another Boolean variable OPEN_SSO_LDAP_IMPORT_ENABLED is defined and initialized. In both the cases, despite the fact that these lines include the term *LDAP*, the file does not actually use or implement the LDAP tactic. During the case study, this file was correctly identified as *Unrelated* to the LDAP tactic. However, it normally takes time and effort for a programmer to manually inspect such files and decide whether a certain tactic is implemented or not. In addition, this manual process could be error prone.

Due to its high recall for tactic-related instances, the BERT approach presented by this work can also enable architectural diagrams generators to show tactic related modules/files and their dependencies. Figure 12 shows part of the LDAP related folders and files in the Liferay project and their dependencies to other modules and files. The files shown on the left hand side are correctly identified as LDAP related and have dependencies to other services and utilities in the Liferay project shown on the right hand side. In projects with very large number of files, packages, and modules, manually tracing module dependencies and judging whether a file implements or uses a tactic is a tedious and risky task. Leveraging the automated approach proposed by this paper, files that do not implement or use a tactic could be automatically ruled out from the manual

TABLE 5: Precision, Recall and F-Measure values per tactic for BERT and TD in Experiment 4.

Tactic	Label	SS	BERT			TD		
			P	R	F	P	R	F
LDAP	LDAP	210	0.49	1	0.65	0.73	0.79	**0.76**
	Unrelated	840	1	0.74	0.85	0.95	0.93	**0.94**
MD5	MD5	96	0.51	0.84	0.64	0.99	0.75	**0.85**
	Unrelated	384	0.95	0.80	0.87	0.94	1.00	**0.97**
SHA	SHA	84	0.83	0.65	**0.73**	0.92	0.54	0.68
	Unrelated	336	0.92	0.97	**0.94**	0.89	0.99	**0.94**
OAUTH2	OAUTH2	389	0.25	0.92	0.39	0.73	0.79	**0.76**
	Unrelated	1556	0.94	0.30	0.45	0.95	0.93	**0.94**

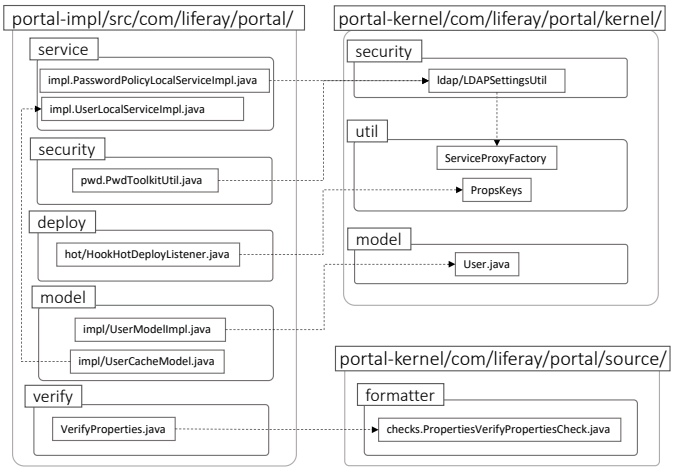

FIG. 12: LDAP related files detected by the BERT model (left) and their dependencies to other files in different folders and packages.

inspection process, and that could lead to a significant reduction in the time and effort needed to identify tactic-related files.

4.5 Use Cases

The results of this work have a couple of practical implications for the security and privacy of software projects in real life:

① **Identification of Security Tactics in Source Code:** This work can be used to pinpoint parts of a software project that implement a security tactic and help to identify existing architectural flaws earlier. Fine tuned BERT models or TD can be used to identify whether a given security tactic (or either one of a group of security controls) is implemented in the files or classes of a software project in Java language. The trained models can further help to identify the type of security controls that are implemented in a project.

② **Security Requirement Tracing:** Given a set of security requirements, the models can be used to trace these requirements and check if they are properly implemented.

③ **Visualization of Tactic Related Modules and their Dependencies:** The output of the proposed method can be used to produce visualizations for the software architecture to highlight modules or packages where security controls are used or implemented and modules that they have a dependency with.

5 Threats to Validity

Perry *et al*. state that external, construct and internal validity are three types of validity threats that should be addressed in research studies [67]. In this section, we briefly discuss the threats that may affect the results of this study, and the measures taken to mitigate them.

External validity evaluates whether the results of a study can be generalized or not and is mostly related to the selection or construction of the datasets used. All tactic related and unrelated code snippets were pulled from the public API of the Stack Overflow platform and reviewed carefully to make sure that all included code pieces were labelled correctly, with a consistent approach. However, it is very difficult to argue that all possible implementations of all security controls are covered by the code snippets in our data sets.

Construct validity threats can occur if a study has errors in the experiment methodology and the tests do not measure what is claimed properly. To mitigate construct validity threats, we use code snippets committed by the diverse user base of the Stack Overflow platform, rather than including code samples from a limited number of software projects. Furthermore, we apply 10 folds cross validation to make sure that we report consistent results for data sets with limited number of code samples.

An internal validity threat occurs if cause effect relationships between the dependent and independent variables in the experiments are not established properly. One limitation of this study that might affect reasoning about evidences more generally is that all code snippets in the generated data sets are implemented in Java language and some tactic implementations are based on common third party frameworks and libraries. Further research with more data sets is needed to repeat designed experiments for security controls implemented with other frameworks and programming languages.

6 Conclusion

This paper creates a comprehensive list of commonly used security controls and categorizes them into three abstract categories, *i.e.*, *Detect*, *Prevent*, and *React*. It pulls a set of tactic related and unrelated code snippets for these controls from the `StackOverflow` question and answer website to generate training and test data sets for conducted experiments. Using the Bidirectional Encoder Representations from Transformer (BERT) and the Tactic Detector (TD) from our prior work [10] it shows that tactic related code snippets can be identified with quite high F-Measure values. In order

to test the performance of the proposed approach on a real-life project, a case study is conducted where trained models are used to identify commonly used security tactics in the Liferay open source enterprise portal. Created data sets and scripts used to train the Tactic Detector and fine tune BERT are shared with the research community through GitHub, to make sure the results of the study are reproducible.

Given a set of code snippets, three experiments are designed to identify (1) the existence of an individual security tactic with a binary classification approach, (2) whether one of the controls in a given set is implemented or not, and (3) the type of implemented security tactic with a multi label classification approach. The results derived from the first two experiments show that fine-tuned BERT models and TD are able to achieve high Precision, Recall and F-Measure values while identifying whether security controls are implemented in a given code snippet or not. With BERT, the average F-Measure scores for the first two experiments are 0.97 and 0.96, respectively. Similarly, TD achieves average F-Measure values of 0.93 and 0.91, respectively. The results of the third experiment show that depending on the type of the security tactic and the size of its training set, the performance of the models may change. BERT and TD achieve F-Measure values up to 0.99 and 0.95, respectively using a multi-label classification approach. Finally, the case study conducted at the end shows the applicability of the developed approach on real-life projects.

REFERENCES

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Adison Wesley, 2003.

[2] M. Hafiz, P. Adamczyk, and R. E. Johnson, "Growing a pattern language (for security)," in *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2012. New York, NY, USA: ACM, 2012, pp. 139–158.

[3] R. Kazman, M. Klein, and P. Clements, "Atam: A method for architecture evaluation," *Software Engineering Institute*, 2000.

[4] R. Taylor, N. Medvidovic, and E. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. John Wiley and Sons, 2009.

[5] G. Fairbanks, *Just Enough Software Architecture A Risk-Driven Approach*. Marshall & Brainerd, 2010.

[6] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, pp. 40–52, October 1992. [Online]. Available: http://doi.acm.org/10.1145/141874.141884

[7] J. Bosch, *Design and use of software architectures: adopting and evolving a product-line approach*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

[8] M. Mirakhorli, J. Carvalho, J. Cleland-Huang, and P. Mader, "A domain-centric approach for recommending architectural tactics to satisfy quality concerns," in *Twin Peaks of Requirements and Architecture (TwinPeaks), 2013 3rd International Workshop on the*, July 2013, pp. 1–8.

[9] M. Mirakhorli and J. Cleland-Huang, "Tracing architectural concerns in high assurance systems (nier track)," *Proceedings - International Conference on Software Engineering*, pp. 908–911, 05 2011.

[10] M. Mirakhorli and J. Cleland-Huang, "Detecting, Tracing, and Monitoring Architectural Tactics in Code," *IEEE Transactions on Software Engineering*, vol. 42, no. 3, pp. 205–220, 2016.

[11] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[13] M. Al-Obeidallah, M. Petridis, and S. Kapetanakis, "A survey on design pattern detection approaches," vol. 7, pp. 41–59, 12 2016.

[14] M. Zanoni, F. A. Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *Journal of Systems and Software*, vol. 103, pp. 102 – 117, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215000217

[15] F. A. Fontana and M. Zanoni, "A tool for design pattern detection and software architecture reconstruction," *Inf. Sci.*, vol. 181, pp. 1306–1324, 2011.

[16] Z. Balanyi and R. Ferenc, "Mining design patterns from c++ source code," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, Sep. 2003, pp. 305–314.

[17] A. K. Dwivedi, A. Tirkey, R. B. Ray, and S. K. Rath, "Software design pattern recognition using machine learning techniques," *2016 IEEE Region 10 Conference (TENCON)*, pp. 222–227, 2016.

[18] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar, "A tactic-centric approach for automating traceability of quality concerns," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 639–649.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[20] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[21] Z. Chen and M. Monperrus, "A literature study of embeddings on source code," 2019.

[22] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019.

[23] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," 2019.

[24] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290353

[25] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 38–49. [Online]. Available: https://doi.org/10.1145/2786805.2786849

[26] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: A neural code search," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 31–41. [Online]. Available: https://doi.org/10.1145/3211346.3211353

[27] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 933–944.

[28] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2019.

[29] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiský, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," 2016.

[30] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083. [Online]. Available: https://www.aclweb.org/anthology/P16-1195

[31] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. A. Sutton, "A survey of machine learning for big code and naturalness," *CoRR*, vol. abs/1709.06182, 2017. [Online]. Available: http://arxiv.org/abs/1709.06182

[32] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[33] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *In EMNLP*, 2014.

[34] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proc. of NAACL*, 2018.

[35] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," 2018.

[36] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[37] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 353–355. [Online]. Available: https://www.aclweb.org/anthology/W18-5446

[38] S. Larson, A. Mahendran, J. J. Peper, C. Clarke, A. Lee, P. Hill, J. K. Kummerfeld, K. Leach, M. A. Laurenzano, L. Tang, and J. Mars, "An evaluation dataset for intent classification and out-of-scope prediction," *CoRR*, vol. abs/1909.02027, 2019. [Online]. Available: http://arxiv.org/abs/1909.02027

[39] A. Ezen-Can, "A comparison of lstm and bert for small corpus," 2020.

[40] NIST, "The Five Functions of Cybersecurity Framework," https://www.nist.gov/cyberframework/online-learning/five-functions, (Accessed on 7/7/2020).

[41] D. Andriesse, H. Bos, and A. Slowinska, "Parallax: Implicit code integrity verification using return-oriented programming," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 125–135.

[42] G. Sivathanu, C. P. Wright, and E. Zadok, "Ensuring data integrity in storage: Techniques and applications," in *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability*. New York, NY, USA: Association for Computing Machinery, 2005, p. 26–36. [Online]. Available: https://doi.org/10.1145/1103780.1103784

[43] H. Handschuh, *SHA Family (Secure Hash Algorithm)*. Boston, MA: Springer US, 2005, pp. 565–567. [Online]. Available: https://doi.org/10.1007/0-387-23483-7_388

[44] A. Bosselaers, *MD4-MD5*. Boston, MA: Springer US, 2005, pp. 378–379. [Online]. Available: https://doi.org/10.1007/0-387-23483-7_249

[45] Cybersecurity and Infrastructure Security Agency, "Security tip (st04-015)," https://www.us-cert.gov/ncas/tips/ST04-015, November 2019, (Accessed on 7/7/2020).

[46] G. Loukas and G. Öke, "Protection against denial of service attacks: A survey," *The Computer Journal*, vol. 53, no. 7, pp. 1020–1037, 2010.

[47] C. Ragkhitwetsagul, J. Krinke, M. Paixão, G. Bianco, and R. Oliveto, "Toxic code snippets on stack overflow," *CoRR*, vol. abs/1806.07659, 2018. [Online]. Available: http://arxiv.org/abs/1806.07659

[48] "Java Cryptography Architecture(JCA) Framework," https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html, (Accessed on 7/7/2020).

[49] "Spring security digest authentication," https://docs.spring.io/spring-security/site/docs/2.0.x/reference/html/digest.html.

[50] "Digest-md5 authentication," https://docs.oracle.com/javase/jndi/tutorial/ldap/security/digest.html.

[51] "Kerberos, Spring Security," https://docs.spring.io/spring-security-kerberos/docs/current/reference/htmlsingle/index.html, (Accessed on 7/7/2020).

[52] "Kerberos, Java Authentication and Authorization Service," https://docs.oracle.com/javase/7/docs/jre/api/security/jaas/spec/com/sun/security/auth/module/Krb5LoginModule.html, (Accessed on 7/7/2020).

[53] "Kerberos, Generic Security Service API," https://docs.oracle.com/javase/jndi/tutorial/ldap/security/gssapi.html, (Accessed on 7/7/2020).

[54] "JSON Web Token(JWT)," https://tools.ietf.org/html/rfc7519, (Accessed on 7/7/2020).

[55] "Spring Security Oauth," https://spring.io/projects/spring-security-oauth, (Accessed on 7/7/2020).

[56] "Spring Security," https://spring.io/projects/spring-security, (Accessed on 7/7/2020).

[57] "Apache ZooKeeper," https://zookeeper.apache.org/, (Accessed on 7/7/2020).

[58] "Spring Session," https://spring.io/projects/spring-session, (Accessed on 7/7/2020).

[59] "Java Secure Channel (JSCH) for SSH2," http://www.jcraft.com/jsch/, (Accessed on 7/7/2020).

[60] "Hibernate Envers: An Easy Auditing / Versioning Solution for Entity Classes," https://hibernate.org/orm/envers/, (Accessed on 7/7/2020).

[61] "JAVERS: Object Auditing and Diff Framework for Java," https://javers.org/, (Accessed on 7/7/2020).

[62] "State-of-the-art Natural Language Processing for PyTorch and TensorFlow 2.0," https://github.com/huggingface/transformers, (Accessed on 11/30/2020).

[63] J. Cleland-Huang, R. Settimi, X. Zou, and P. Solc, "The detection and classification of non-functional requirements with application to early aspects," in *14th IEEE International Requirements Engineering Conference (RE'06)*, 2006, pp. 39–48.

[64] G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *SIGKDD Explor. Newsl.*, vol. 6, no. 1, p. 20–29, Jun. 2004. [Online]. Available: https://doi.org/10.1145/1007730.1007735

[65] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority over-Sampling Technique," *Journal of Artificial Intelligence Research*, vol. 16, no. 1, p. 321–357, 06 2002.

[66] L. Gong, D. He, Z. Li, T. Qin, L. Wang, and T. Liu, "Efficient training of bert by progressively stacking," in *International Conference on Machine Learning*. PMLR, 2019, pp. 2337–2346.

[67] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical Studies of Software Engineering: A Roadmap," in *Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2000, p. 345–355. [Online]. Available: https://doi.org/10.1145/336512.336586