Stack Overflow Considered Helpful! Deep Learning Security Nudges Towards Stronger Cryptography

Felix Fischer, *Technical University of Munich;* **Huang Xiao,** *Bosch Center for Artificial Intelligence;* **Ching-Yu Kao,** *Fraunhofer AISEC;* **Yannick Stachelscheid, Benjamin Johnson, and Danial Razar,** *Technical University of Munich;* **Paul Fawkesley and Nat Buckley,** *Projects by IF;* **Konstantin Böttinger,** *Fraunhofer AISEC;* **Paul Muntean and Jens Grossklags,** *Technical University of Munich*

https://www.usenix.org/conference/usenixsecurity19/presentation/fischer

This paper is included in the Proceedings of the 28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Open access to the Proceedings of the 28th USENIX Security Symposium is sponsored by USENIX.

Stack Overflow Considered Helpful! Deep Learning Security Nudges Towards Stronger Cryptography

Felix Fischer, Huang Xiao[†], Ching-Yu Kao[*], Yannick Stachelscheid, Benjamin Johnson, Danial Razar
Paul Fawkesley[◇], Nat Buckley[◇], Konstantin Böttinger[*], Paul Muntean, Jens Grossklags
Technical University of Munich, [†]Bosch Center for Artificial Intelligence
[*]Fraunhofer AISEC, [◇]Projects by IF
{*flx.fischer, yannick.stachelscheid, benjamin.johnson, danial.razar, paul.muntean, jens.grossklags*}*@tum.de*
{*huang.xiao*}*@de.bosch.com,* {*nat, paul*}*@projectsbyif.com,* {*ching-yu.kao, konstantin.boettinger*}*@aisec.fraunhofer.de*

Abstract

Stack Overflow is the most popular discussion platform for software developers. However, recent research identified a large amount of insecure encryption code in production systems that has been inspired by examples given on Stack Overflow. By copying and pasting functional code, developers introduced exploitable software vulnerabilities into security-sensitive high-profile applications installed by millions of users every day.

Proposed mitigations of this problem suffer from usability flaws and push developers to continue shopping for code examples on Stack Overflow once again. This motivates us to fight the proliferation of insecure code directly at the root before it even reaches the clipboard. By viewing Stack Overflow as a market, implementation of cryptography becomes a decision-making problem. In this context, our goal is to simplify the selection of helpful and secure examples. More specifically, we focus on supporting software developers in making better decisions on Stack Overflow by applying nudges, a concept borrowed from behavioral economics and psychology. This approach is motivated by one of our key findings: For 99.37% of insecure code examples on Stack Overflow, similar alternatives are available that serve the same use case and provide strong cryptography.

Our system design that modifies Stack Overflow is based on several nudges that are controlled by a deep neural network. It learns a representation for cryptographic API usage patterns and classification of their security, achieving average AUC-ROC of 0.992. With a user study, we demonstrate that nudge-based security advice significantly helps tackling the most popular and error-prone cryptographic use cases in Android.

1 Introduction

Informal documentation such as Stack Overflow outperforms formal documentation in effectiveness and efficiency when helping software developers implementing functional code. The fact that 78% of software developers primarily seek help on Stack Overflow on a daily basis[1] underlines the usability and perceived value of community and example-driven documentation [2].

Reuse of code examples is the most frequently observed user pattern on Stack Overflow [17]. It reduces the effort for implementing a functional solution to its minimum and the functionality of the solution can immediately be tested and verified. However, when implementing encryption, its security, being a non-functional property, is difficult to verify as it necessitates profound knowledge of the underlying cryptographic concepts. Moreover, most developers are unaware of pitfalls when applying cryptography and that misuse can actually harm application security. Instead, it is often assumed that mere application of any encryption is already enough to protect private data [13, 14]. Stack Overflow users also cannot rely on the community to correctly verify the security of available code examples [9]. Security advice given by community members and moderators is mostly missing and oftentimes overlooked. This is due to only a few security experts being available as community moderators and a feedback system which is not sufficient to communicate security advice effectively. Consequently, highly insecure code examples are frequently reused in production code [17]. Exploiting these insecure samples, high-profile applications were successfully attacked, leading to theft of user credentials, credit card numbers and other private data [13].

While mainly focused on the negative impact of Stack Overflow on code security, recent research has also reported that there is a full range of code snippets providing strong security for symmetric, asymmetric and password-based encryption, as well as TLS, message digests, random number generation, and authentication [17]. However, it was previously unknown whether useful alternatives can be found for most use cases. In our work, we show that for 99.37% of insecure encryption code examples on Stack Overflow a similar secure alternative is available that serves the same use

[1] `https://insights.stackoverflow.com/survey/2016#community`

case. So, why are they not used in a consistent fashion?

We take a new perspective and see implementation of cryptography as a decision-making problem between available secure and insecure examples on Stack Overflow. In order to assist developers in making better security decisions, we apply nudges, a concept borrowed from experimental economics and psychology to attempt altering individuals' behaviors in a predictable way without forbidding any options or significantly changing their economic incentives. Nudging interventions typically address problems associated with cognitive and behavioral biases, such as anchoring, loss aversion, framing, optimism, overconfidence, post-completion errors, and status-quo bias [5, 31]. They have been applied in the security and privacy domain in a successful fashion [4, 5, 7, 19, 22, 32]. In contrast to these approaches, which focused on systems for end-users, we translate the concept of nudges to the software developer domain by modifying the choice architecture of Stack Overflow. It nudges developers towards reusing secure code examples without interfering with their primary goals.

Our designed security nudges are controlled by a code analysis system based on deep learning. It learns general features that allow the separation of secure and insecure cryptographic usage patterns, as well as their similarity-based clustering and use-case classification. Applying this system, we can directly derive a choice architecture that is based on providing similar, secure, and use-case preserving code examples for insecure encryption code on Stack Overflow.

In summary, we make the following contributions:

- We present a deep learning-based representation learning approach of cryptographic API usage patterns that encodes their similarity, use case and security.
- Our trained security classification model which uses the learned representations achieves average AUC-ROC of 0.992 for predicting insecure usage patterns.
- We design and implement several security nudges on Stack Overflow that apply our similarity, use case and security models to help developers make better decisions when reusing encryption code examples.
- We demonstrate the effectiveness of nudge-based security advice within a user study where participants had to implement the two most popular and error-prone cryptographic use cases in Android [13, 17]: nudged participants provided significantly more secure solutions, while achieving the same level of functionality as the control group.

We proceed as follows. After reviewing related work (Section 2), we present our system design that combines deep learning-based representation learning with nudge-based security advice (Sections 3 – 6). Then, we present our model evaluation and user study (Sections 7 & 8), as well as limitations, future work, and conclusions (Sections 9 – 11).

2 Related Work

2.1 Getting Cryptography Right

Acar et al. [2] have investigated the impact of formal and informal information sources on Android application security. With a lab study, they found that developers prefer informal documentation such as Stack Overflow over official Android documentation and textbooks when implementing encryption code. Solutions based on advice from Stack Overflow provided significantly more functional – but less secure – solutions than those based on formal documentation. Work by Fischer et al. [17] showed that 30% of cryptographic code examples on Stack Overflow were insecure. Many severely vulnerable samples were reused in over 190,000 Android applications from Google Play including high-profile applications from security-sensitive categories. Moreover, they have shown that the community feedback given on Stack Overflow was not helpful in preventing reuse of insecure code.

Chen et al. studied the impact of these community dynamics on Stack Overflow in more detail [9]. Based on manual inspection of a subset of posts, they found that (on average) posts with insecure snippets garnered higher view counts and higher scores, and had more duplicates compared to posts with secure snippets. Further, they demonstrated that a sizable subset of posts from trusted users were insecure. Taken together, these works show that developers (by copying and pasting insecure code) are imposing negative externalities on millions of users who eventually bear the cost of apps harboring vulnerabilities [8].

Oliveira et al. focus on developers' misunderstandings of ambiguities in APIs (including cryptography), which may contribute to vulnerabilities in the developed code [26]. They studied the impact of personality characteristics and contextual factors (such as problem complexity), which impact developers' ability to identify such ambiguities. Likewise, Acar et al. [1] investigated whether current cryptographic API design had an impact on cryptographic misuse. They selected different cryptographic APIs; including some particularly simplified APIs in order to prevent misuse. However, while indeed improving security, these APIs produced significantly less functional solutions and oftentimes were not applicable to specific use cases at all. As a consequence, developers searched for code examples on Stack Overflow again.

Nguyen et al. [25] developed FixDroid, a static code analysis tool integrated in Android Studio which checks cryptographic code flaws and suggests quick fixes.

2.2 Security Nudges

Wang et al. [32] implemented privacy nudges on Facebook in order to make users consider the content and audience of their online publications more carefully, as research has

shown that users eventually regret some of their disclosure decisions. They found that a reminder nudge about the audience effectively and non-intrusively prevents unintended disclosure. Almuhimedi et al. [6] implemented an app permissions manager that sends out nudges to the user in order to raise awareness of data collected by installed apps. With the help of a user study they were able to show that 95% of the participants reassessed their permissions, while 58% of them further restricted them. Liu et al. [23] created a personalized privacy assistant that predicts personalized privacy settings based on a questionnaire. In a field study, 78.7% of the recommendations made by the assistant were adopted by users, who perceived these recommendations as usable and useful. They were further motivated to review and modify the proposed settings with daily privacy nudges.

2.3 Deep Learning Code

Fischer et al. [17] proposed an approach based on machine learning to predict the security score of encryption code snippets from Stack Overflow. They used *tf-idf* to generate features from source code and trained a support vector machine (SVM) using an annotated dataset of code snippets. The resulting model was able to predict the security score of code snippets with an accuracy of 0.86, with precision and recall of 0.85 and 0.75, respectively. However, security predictions were only available for the complete code snippet. It did not allow indicating and marking specific code parts within the snippet to be insecure. This lack of explainability is detrimental for security advice.

Xiaojun et al. [33] introduced neural network-based representation learning of control flow graphs (CFGs) generated from binary code. Using a Siamese network architecture they learned similar graph embeddings using *Structure2vec* [11] from similar binary functions over different platforms. These embeddings were used to detect vulnerabilities in binary blobs by applying code-similarity search. Their approach significantly outperformed the state-of-the-art [16] in both, efficiency and effectiveness, by providing shorter training times and higher area under the curve (AUC) on detecting vulnerabilities. The approach does not allow identification and description of code parts within binary functions that cause the vulnerabilities. To allow better explainability, we depend on our new approach to provide statement-level granularity. It enables identifying and classifying multiple code patterns within a single function.

Li et al. [21] developed VulDeePecker, a long short-term memory (LSTM) neural network that predicts buffer overflows and resource management error vulnerabilities of source code gadgets. Code gadgets are backward and forward slices, considering data and control flow, that are generated from arguments used in library function calls. Further, they use *word2Vec* to create embeddings for the symbolic representation of code gadgets. These embeddings are then used together with their security label to train a bi-directional LSTM. VulDeePecker outperforms several pattern-based and code similarity-based vulnerability detection systems with respect to false positive and false negative rates. However, their LSTM model has a very long training time. Our convolutional approach leverages transfer learning to achieve much faster training.

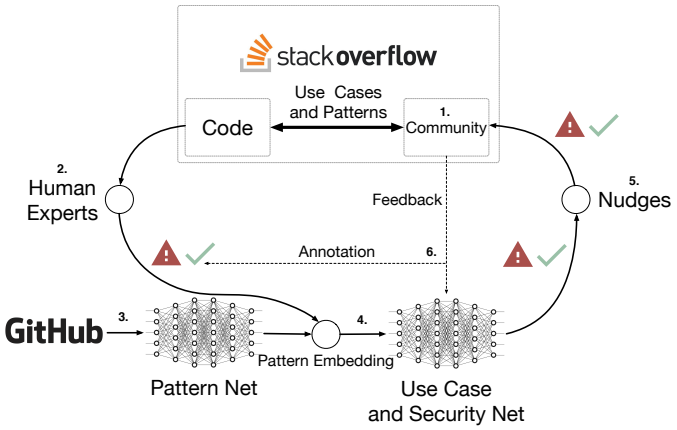


Figure 1: Learn-to-Nudge Loop Overview

3 Overview

We present an overview of our system design for nudge- and deep learning-based security advice on Stack Overflow in Figure 1. It depicts a learn-to-nudge loop that represents the interaction and interference of the community behavior, classification models and proposed security nudges on Stack Overflow. The community behavior on Stack Overflow (1) triggers the loop by continuously providing and reusing code examples that introduce new use cases and patterns of cryptographic application programming interfaces (APIs). In the initial step (2), a representative subset of these code examples is extracted and annotated by human experts. The annotations provide ground truth about the use cases and security of cryptographic patterns in the given code. Then (3), a representation for these patterns is learned by an unsupervised neural network based on open source projects provided by GitHub. In combination with the given annotations, the pattern embeddings are used to train an additional model to predict their use cases and security (4). Based on these predictions, we can apply security nudges on Stack Overflow by providing security warnings, reminders, recommendations and defaults for encryption code examples (5). Further, we allow assigned security moderators[2] within the community to annotate unknown patterns and provide feedback to predictions of our models (6). Therefore, our system creates a

[2] https://stackexchange.com/about/moderators

learn-to-nudge loop that is supposed to iteratively improve the classification models, which in turn help improving the security decisions made by the community and the security of code provided on Stack Overflow.

4 Nudge-Based System Design

We apply five popular nudges [30, 31] and describe their translation to security advice in this section.

Simplification A simplification nudge promotes building upon existing and established infrastructures and programs. We apply this nudge by integrating our system in Stack Overflow, a platform that is already used by the majority of software developers worldwide. By integrating developer tools on a platform that is already used by almost everyone, we unburden developers from installing additional tools. Moreover, it allows us to create awareness of the problem of cryptographic misuse in general.

Warnings A warning nudge aims at raising the user's attention in order to counteract the natural human tendency towards unrealistic optimism [5]. We apply this nudge by integrating security warnings on Stack Overflow. Whenever an insecure code example has been detected, a warning is displayed to the developer to inform about the security problem and potential risks in reusing the code sample.

Increases in Ease and Convenience (IEC) Research has also shown that users oftentimes discount security warnings. However, if they additionally describe available alternative options to make a less risky decision, warnings tend to be much more effective [5]. Therefore, our design combines security warnings with recommendations for similar code examples with strong cryptography. With this nudge, we make code examples with better security visible to the user. To provide an easy choice, we present the recommended code examples by displaying a list of the related posts. This aims at encouraging the user to consider the recommendations as it only demands clicking on a link.

Reminders Users might not engage in the expected conduct of paying attention to the warning and following the recommendations. This might be due to inertia, procrastination, competing priorities, and simple forgetfulness [5]. Oftentimes seeking functional solutions is considered as a competing priority to secure solutions [2]. Therefore, we apply a reminder nudge, which is triggered whenever the user copies an insecure code example.

Defaults The default nudge is the most popular and effective nudge in improving decision-making. Popular examples are automated enrollment in healthcare plans or corporate wellness programs, or double-sided printing which can promote environmental protection [5]. We apply this nudge by up-ranking posts that only contain secure code examples in Stack Overflow search results by default.

The goal of our approach is to thoughtfully develop a new user interface (UI) design that implements the proposed nudges (see Section 6) and to test whether it improves developer behavior on Stack Overflow. Please note that we do not intend to comparatively evaluate multiple UI candidates for our design patterns to identify the most effective one. We consider this out of scope for this paper and leave this task for future work.

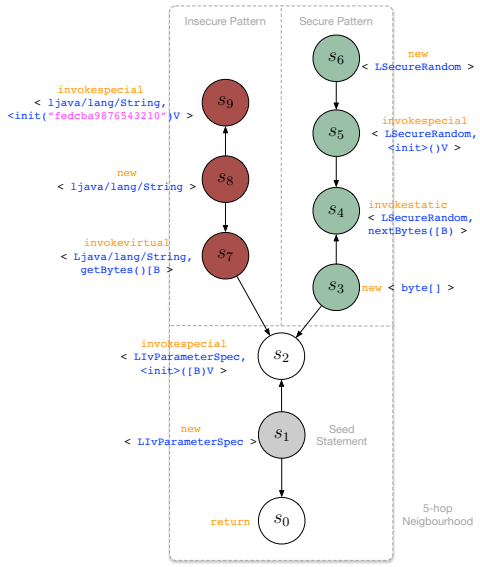
5 Neural Network-Based Learning of Cryptographic Use Cases and Security

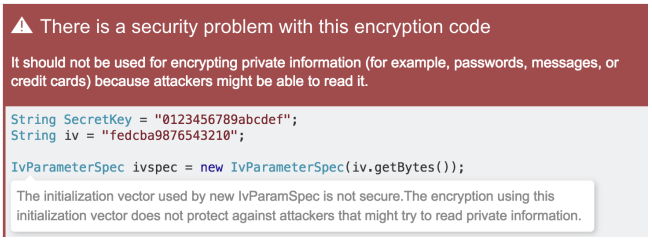
The nudge-based system design requires algorithmic decisions about the security and similarity of code examples. In order to display security warnings, code examples have to be scanned for encryption flaws. To further recommend helpful alternatives without common encryption problems, Stack Overflow posts have to be scanned for similar examples with strong cryptography.

Due to *Simplification* (see Section 4), we already chose a platform that provides us with a large amount of secure and insecure samples that contain cryptographic API usage patterns to learn from in order to design the code analysis approach [17]. Instead of defining rule-based algorithms [12, 13, 20] that would have to be updated whenever samples with unknown patterns are added to Stack Overflow, we simplify and increase the flexibility of our system by applying deep learning to automatically learn the similarity, use-case and security features from the ever-increasing dataset of available code on Stack Overflow. Based on the learned features, our models are able to predict insecure code examples and similar but secure alternatives that serve the same use case. However, newly added code examples that provide unknown use cases and security flaws might be underrepresented in the data and therefore difficult to learn. Therefore, we apply transfer learning where we reuse already obtained knowledge that facilitates learning from a small sample set of a similar domain.

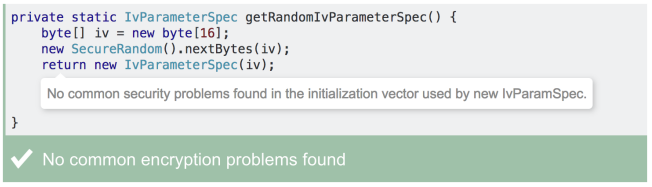
5.1 Cryptographic Use Cases

Stack Overflow offers a valuable source for common use cases of cryptographic APIs in Android. As developers post questions whenever they have a particular problem with an API, a collection of error-prone or difficult cryptographic problems is aggregating over time. Moreover, frequencies of similar posted questions, view counts, and scores of questions posted on Stack Overflow indicate very common and important problems developers encounter when writing





(a) Example for an insecure pattern: The initialization vector (IV) is created from a static string value stored in the code.



(b) Example for a secure pattern: A secure random source is used to generate the IV.

Figure 2: Example for a secure and insecure usage pattern of *new IvParameterSpec*. It shows the program dependency graph (PDG) of the 5-hop neighborhood of the seed statement s_1 for the secure and insecure code example displayed in (a) and (b). Next to each node in the graph we provide the shortened signature of the related statement, highlighting a subset of its attributes we store in the feature vector. Bytecode instruction types are highlighted yellow, Java types blue and constants magenta.

security-related code. Therefore, Stack Overflow can be seen as a dataset of different cryptographic use cases that are frequently required in production code. Previous work identified the most popular and error-prone use cases of cryptography in Android apps [17]. The authors scanned Stack Overflow for insecure code examples that use popular cryptographic APIs, e g. Oracle's Java Cryptography Architecture (JCA), and detected their reuse in Android applications. We summarize the identified use cases in Table 1.

Use Case Identifier	Usage Pattern Description	API Seed Statement
Cipher	Initialization of cipher, mode and padding	*Cipher.getInstance*
Key	Generation of symmetric key	*new SecretKeySpec*
IV	Generation of initialization vector	*new IvParameterSpec*
Hash	Initialization of cryptographic hash function	*MessageDigest. getInstance*
TLS	Initialization of TLS protocol	*SSLContext.getInstance*
HNV	Setting the hostname verifier	*setHostnameVerifier*
HNVOR	Overriding the hostname verification	*verify*
TM	Overriding server certificate verification	*checkServerTrusted*

Table 1: Common cryptographic use cases in Android

5.2 Learning API Usage Patterns

In order to predict similarity, use case and security of encryption code, we need to learn a numerical representation of the related patterns that can be understood by a neural network. Therefore, our first step is learning an embedding of cryptographic API usage patterns.

Usage Pattern As shown in Table 1, a cryptographic API element, e. g., *javax.crypto.Cipher.getInstance*, can have different usage patterns that belong to the same use case. A usage pattern consists of a particular API element, all statements it depends on, and all its dependent statements within the given code. In other words, a pattern can be seen as a subgraph of the PDG, which represents the control and data dependencies of statements. The subgraph is created by pruning the graph from anything but the forward and backward slices of the API element, as shown in Figure 2. We call this element the *seed statement*. This pruned graph can become very large and therefore might contain noise with respect to the identification of patterns. Our goal is to learn an optimal representation of usage patterns that allows accurate classification of their use cases and security. Ideally, the related subgraph is minimized to a *neighborhood* of the seed statement in the pruned PDG such that it provides enough information to solve the classification tasks.

Neighborhood Aggregation Our approach learns *pattern embeddings* for the K-hop neighborhood of cryptographic API elements within the PDG, as shown in Figure 2. To generate these embeddings we use the *neighborhood-aggregation* algorithm provided by *Structure2vec* [11]. This method leverages node features (e. g., instruction types of a statement node) and graph statistics (e. g., node degrees) to

inform their embeddings. It provides a convolutional method that represents a node as a function of its surrounding neighborhood. The parameter K allows us to search for a neighborhood that optimally represents usage patterns to solve given classification tasks. In other words, we learn the code representation in a way such that its features improve use case and security prediction of the code. As we will show throughout this work, this representation is very helpful for classifying cryptographic API usage patterns. We further argue that the learned pattern representation is not restricted to cryptographic APIs, as the used features are general code graph properties.

Neighborhood Similarity We learn pattern embeddings such that similar patterns have similar embeddings by minimizing their distance in the embedding space. Therefore, next to the neighborhood information, pattern embeddings additionally encode their similarity information. On the one hand, this allows us to apply efficient and accurate search for similar usage patterns on Stack Overflow [33]. On the other hand, we can transfer knowledge from the similarity domain to the use case and security domain. This knowledge transfer is leveraged by our use case and security classification models.

Code similarity is very helpful to predict code security. Therefore, we expect that the similarity feature of our pattern embeddings will improve the accuracy and efficiency of the security classification model. However, code similarity is oftentimes not enough for predicting security. Therefore, the main effort of our classification models lies in learning the additional unknown conditions where code similarity becomes insufficient.

To learn our embeddings, we apply a modified architecture of the graph embedding network *Gemini* [33].

5.3 Feature Engineering

The embedding network should learn a pattern embedding that is general enough to allow several classification tasks. This means that the embedding has to be learned from general code features or attributes, e. g., statistical and structural features [33] from each statement within the PDG representation of the code. Further, pattern embeddings should represent very small neighborhoods. As we want to minimize the neighborhood size K, patterns might consist only of a few lines of code. Therefore, considering only graph statistics as features might not be sufficient and may result in similar features for dissimilar patterns. In order to overcome these insufficiencies, we additionally combine structural and statistical with lexical and numerical features for each statement in a neighborhood.

Structure and Statistics We first create the PDG of the given input program using WALA[3], a static analysis framework for Java. Note that WALA creates a PDG for each Java method. Then, we extract the resulting statistical and structural features for each statement. We store the bytecode instruction type of a statement using a one-hot indicator vector. Additionally, we store the count of string and numerical constants that are used by the statement. We further add structural features by storing the offspring count and node degree of a statement in the PDG [33]. Finally, we store the indexes of the statement's direct neighbors in the graph.

Element Names and String Constants Method and field names of APIs are strings and have to be transformed into a numerical representation first. We learn feature vectors for these tokens by training a simple unsupervised neural network to predict the Java type that defines the given method or field name. Thereby, each name is represented in a one-hot encoding vector with dimension 23,545, corresponding to the number of unique element names provided by the cryptographic APIs [17]. To learn features, we use a network architecture with one hidden layer and apply categorical cross-entropy as a loss function during training. Finally, we apply the trained model on all names and extract the neurons of the hidden layer as they can be seen as learned features necessary to solve the classification task. This way, each name obtains a unique feature vector which preserves its type information. We use the same approach for learning feature vectors for the 763 unique string constants given by the APIs.

5.4 Pattern Embedding Network

Many code examples on Stack Overflow typically do not provide sound programs as they mostly consist of loose code parts [17]. In contrast to complete programs, compiling these partial programs might introduce multiple types of ambiguities in the resulting PDG such that the extracted statement features x_u are not sound [10]. Whenever we generate sound and unsound features x_s, x_u from a complete and a partial program, respectively, that provide the same usage pattern for a given seed statement, both sets of feature vectors extracted from the patterns might be different. Therefore, we need to learn a representation for patterns that preserves their similarity properties independently from the shape of the containing program. With a Siamese network architecture [33], we can learn similar pattern embeddings independently from the completeness of the code example. It learns embeddings from similar and dissimilar input pairs. We create similar input pairs by extracting sound and unsound features for the same pattern and dissimilar pairs by extracting sound and unsound features from different patterns. The

[3] `https://github.com/wala/WALA`

Algorithm 1 Neighborhood-aggregation algorithm

Input: PDG $G(V,E)$ input features $\{x_v, \forall v \in V\}$;
Output: Pattern embedding $p_v, \forall v \in V$

$\phi_v^0 \leftarrow 0, \forall v \in V$,
for $k = 1...K$ **do**
  for $v \in V$ **do**
    $\phi_{N(v)}^k \leftarrow AGGREGATE(\phi_n^{k-1}, \forall n \in N(v))$
    $\phi_v^k \leftarrow \tanh(W_1 x_v \cdot \sigma(\phi_{N(v)}^k)$
return $\{p_v = W_2 \phi_v^K, \forall v \in V\}$

trained model will then generate similar embeddings independently from the completeness of the program.

The pattern embeddings are generated with *Structure2vec* as depicted in Algorithm 1. We provide the abstract description of the algorithm and refer to *Gemini's* neural network architecture that gives information about its implementation, which we use as the basis for our approach. The update function calculates a pattern embedding p_v for each feature vector x_v of statements (i. e., nodes) $v \in V$ in the PDG $G(V,E)$. An embedding p_v is generated by recursively aggregating previously generated embeddings $\{\phi_n^{k-1}, \forall n \in N(v)\}$ of direct neighbors $N(v)$ in the graph, combining it with the weighted feature vector x_v. Unlike *Gemini*, which outputs an aggregation of p_v to return an embedding for the complete graph, our network returns the set of pattern embeddings $P = \{p_v, \forall v \in V\}$.

We give an overview of the pattern embedding network in Figure 3. Here, the insecure pattern $G(x_3, x_4, x_5, E)$ informs the embedding of its direct neighbors in each iteration step, finally informing the seed statement in iteration $k = 2$. After this step, the seed statement knows that it is part of an insecure pattern and its embedding preserves this information accordingly. We extract the pattern embedding ϕ_1^K of the seed statement and apply weights W_2. Note, we train W_2 based on classification loss of aggregated pattern similarity p_a as explained in Section 5.5. However, we use the trained model to generate and output embeddings for each individual pattern p_v in the graph (see Figure 3).

5.5 Training

For unsupervised training of pattern embeddings, we need to generate similar and dissimilar input pairs from data that provides ground truth. We use two different sets of PDGs that are compiled from the same source code. One set S contains the sound graph representations of the code, the other one the unsound graphs U. A sound graph $G_s(V_s, E_s)$ in the first set is compiled from a complete program using a standard Java compiler. An unsound graph $G_u(V_u, E_u)$ in the second set is generated by a partial compiler [10] that compiles each Java class of a program individually. Then we construct the feature vectors $X_s = \{x_s\}_{v_s \in V_s}$ and $X_u = \{x_u\}_{v_u \in V_u}$ for all

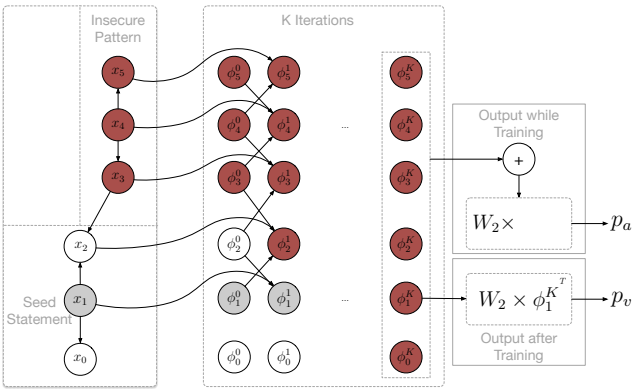


Figure 3: Pattern embedding network overview

statements v in the respective graphs.

To obtain ground truth for similar and dissimilar usage patterns we need to create similar pairs $\left(\langle v_s, v_u \rangle, 1\right)$ and dissimilar pairs $\left(\langle v_s, v_u \rangle, -1\right)$. However, we do not have information about the relationship of statements in $G_u(V_u, E_u)$ with statements $G_s(V_s, E_s)$, which is necessary to create these pairs. Note that source code statements do not correspond one-to-one with statements in the PDG. The compiler may divide a source code statement into multiple instructions, which may have different associated statements in the PDG. Since we use different compilers, the resulting PDG statements in V_s and V_u may look different even though they represent the same source code. Therefore, we aggregate all pattern embeddings from a method into p_a and use the resulting embedding for training. The network calculates the loss based on the cosine similarity of the aggregated pattern embedding pairs $\{\langle p_a^s, p_a^u \rangle\}_{x_s, x_u \in X_s, X_u}$ and their given similarity label $y \in \{-1, 1\}$

We downloaded 824 open source Android apps from GitHub and compiled the complete and sound graph $G_s(V_s, E_s)$ for each method. Further, we used the partial compiler to obtain the unsound graph for each method $G_u(V_u, E_u)$. After creating the feature vectors X_s and X_u from the graphs, similar method pairs $\left(\langle X_s, X_u \rangle, 1\right)$ were created by extracting X_s and X_u from the same source code, and dissimilar $\left(\langle X_s, X_u \rangle, -1\right)$ by extracting X_s and X_u from different source code. From the 824 downloaded apps, we extracted 91,075 methods to create 157,162 input pairs in total. These pairs have been split up into the training and validation set, where 80% have been randomly allocated for training and 20% for validation. Note that the intersection of both sets is empty.

5.6 Learning Use Cases and Security

From a given source code example, we want to be able to predict the cryptographic use case and security of patterns within the code. We apply transfer learning by reusing the

previously learned pattern embedding that already encodes their similarity information.

Pattern embeddings are learned unsupervised and we can obtain almost arbitrarily large training datasets from open source projects. However, code examples on Stack Overflow provide a very different distribution of data [17]. Many use case and security classes are under- or overrepresented and availability of encryption code examples is limited in general. We transfer knowledge from the similarity domain to the use case and security domain in order to tackle these problems. We argue that the similarity information preserved in our pattern embeddings will be helpful for classifying their use cases and security.

5.7 Labeling

We extracted 10,558 code examples from Stack Overflow[4] by searching for code that contained at least one of the seed statements. Each code sample has been manually reviewed in order to label use case and security of the contained usage patterns. Labeling was done by two security experts individually applying the labeling rules given by [12, 13, 17]. This leads to conservative binary security labeling, which might at times be too strict. For instance, depending on the context, MD5 can be the better trade-off and secure enough. However, our approach aims at developers that are layman in cryptography and we consider binary classification preferable to encourage safe defaults.

Initially, 100 samples for each of the different seed statements have been selected randomly to apply dual control labeling. After clearing up disagreements, the remaining samples have been annotated individually to speed up the labeling. The whole process took approximately 10 man days to complete. To evaluate individual annotation accuracy, we randomly selected 200 samples from both experts and report agreement of 98.32% on given labels. We further publish the annotated dataset in order to allow verification of annotation accuracy and reproduction of our results. Please refer to Appendix C for further details on the annotation process.

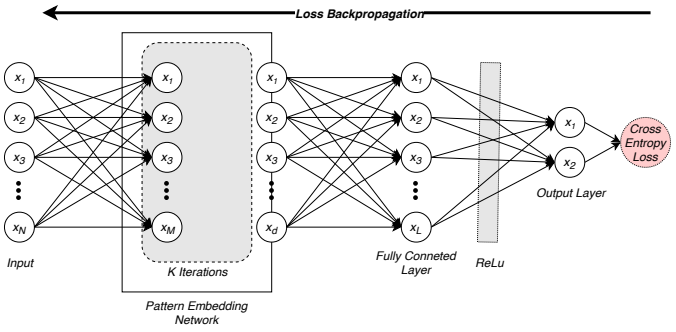


Figure 4: End-to-End Network Architecture

[4] I.e., from the Official Stack Overflow Data Dump.

5.8 End-to-end Architecture

We introduce an architecture that allows classification of different uses cases and security, while improving the pattern-embedding model in order to forward optimized code representations to the classification layer (see Figure 4).

To achieve this, we add a fully connected layer with dimension 1,024 using Rectified Linear Unit (ReLu) as the activation function on top of the pattern-embedding network. We use a softmax layer for binary and multi-class classification and trained our network to optimize cross-entropy loss. Applying transfer learning, we initialize the pattern-embedding network using the previously learned weights for pattern similarity (see Section 5.4). The end-to-end network now connects the pattern-embedding network with the classification layers. Within training, the latter backpropagates cross-entropy loss from the classification task all the way to the input of the pattern-embedding network. This allows the similarity network to adjust the pattern representation in order to better perform on the classification. Therefore, both coupled networks now generate a new pattern representation for the given classification problem in such a way that it is optimally solved.

For instance, security classification of *Cipher, Key* and *TM* rely on very different features. Only using the pre-trained "static" pattern embeddings might therefore be disadvantageous for some use cases. However, by dynamically customizing the pattern embedding with respect to classification loss minimization, the network learns a code representation that preserves the necessary code features to improve classification.

5.9 Training

The Stack Overflow dataset provides 16,539 pattern embeddings extracted from 10,558 code examples. Note that a single code example might contain several patterns, e. g. *IV, Key* is used to initialize *Cipher*. We test pattern embeddings generated from several models that were trained on different neighborhood sizes K and different output dimensions d for the embeddings. Thereby, we search for the optimal hyperparameter K and d to achieve the best performance on both classification tasks. We first train the network to learn the use case identifier of pattern embeddings using the complete dataset. Then, we train a different model to learn the security labels. Here, patterns with the same security label belong to the same class independently of their use case. Finally, we divide the dataset into combinations of several use case classes, testing the effect on performance of security prediction.

6 Security Nudges

The neural network architecture described in the previous section provides everything needed to apply the security nudges on Stack Overflow. In this section, we explain the design of each nudge including its implementation on Stack Overflow and how it applies the predictions from the similarity and classification models[5].

6.1 Security Warnings

Whenever an insecure code example is detected, a security warning, as shown in Figure 8, which surrounds the code, is displayed to the user. The warning is triggered by the prediction result of the security model that classifies each pattern in the snippet.

The difficulties in designing effective security warnings are widely known and have been extensively investigated. We base our approach on the design patterns of Google Chrome's security warning for insecure server communication, whose effectiveness has been comprehensively field-tested [3]. The header of the warning informs the user that a security problem has been detected in the encryption code of the sample. Note that we assume users with a very diverse background, knowledge and expertise in cryptography. Users and even experienced developers might not be aware of flawed or out-dated encryption that does not provide sufficient security. Therefore, we inform the user about the consequences that might occur when reusing insecure code examples in production code, e. g., private information might be at risk in an attack scenario.

We further provide code annotations for each seed statement in the code whose usage pattern has been classified as insecure (see Figure 2). The annotation is attached below and points at the statement. It gives further information about the statement, while additionally highlighting the consequences of reusing it. In order to select the correct annotation for the insecure statement, we apply use case prediction of the related pattern. Each use case identifier has an assigned security annotation to be displayed in the code snippet.

6.2 Security Recommendations

Security warnings should always offer a way out of a situation where the user seems to be unable to continue with her current action due to the warning. Whenever the user decides to follow the advice given by the warning, she would refuse to reuse the code example that was originally considered a candidate for solving her problem. In this situation, she has been thrown out of her usual user pattern as she has to restart searching for another example. Therefore, for each insecure code example, we recommend a list of similar examples, as shown in Figure 8, that serve the same use case and provide stronger encryption. Ideally, the user would only have to click on a single link to the recommended alternative. Our nudge design pattern does not claim that the recommended code is generally secure, as it still might contain insecure patterns that are unknown to the model. However, for simplicity, we refer to code examples, which do not contain any detected insecure patterns and do contain detected secure patterns, as *secure* code examples throughout the paper.

We create this list of recommendations by applying similarity search, use case and security prediction of usage patterns in code examples. We start with predicting the use case of each insecure usage pattern. I_q contains all insecure use cases of a method $q \in M_q$, where M_q is the set of query methods in the snippet. We create the set $\{I_q\}_{q \in M_q}$, which consists of the sets of insecure use cases over all methods in the snippet. Then, we generate the set of aggregated pattern embeddings $\{e_q\}_{q \in M_q}$, as described in Section 5.4. Afterwards, we analogously create the set of secure use cases for all target methods $\{S_t\}_{t \in M_t}$ where M_t is the set of methods available on Stack Overflow that only contain usage patterns our model has classified as secure. Likewise, we create the aggregated pattern embeddings $\{e_t\}_{t \in M_t}$. We rank M_t for given I_q, S_t and e_q, e_t based on ascending Jaccard distance $d_J(I_q, S_t)$, ranking pairs with the same distance using cosine similarity $cos(e_q, e_t)$. We create the ranked list of recommended posts R by adding the related Stack Overflow post for each t of the top-fifty results in the ranking. Beneath the security warning, we display a scrollable list of R, as displayed in Figure 8. Each post is displayed by showing the title of the related question. When the user clicks on the title, a new browser tab opens and the web page automatically scrolls down to the recommended code example, highlighting it with a short flash animation.

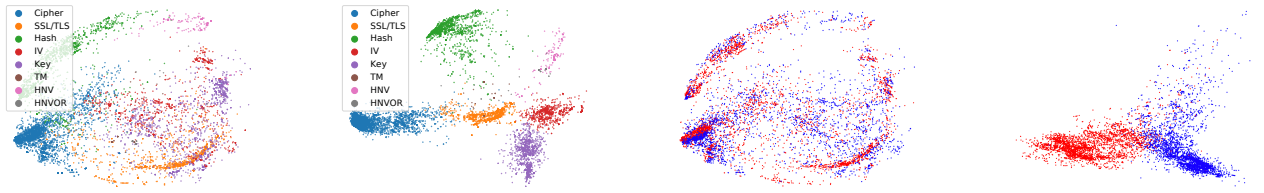
Recommended examples are displayed inside a green box, annotated with a check mark and message informing the user that no common encryption problems have been found within the code. This way, we avoid declaring the code example to be secure, which would be a too strong claim. However, the statement intends to be strong enough to reach the users and make them follow the advice. Similar to warnings, we provide code annotations for each statement in the code whose usage pattern has been classified as secure.

6.3 Security Reminders and Defaults

We further caution the user – in addition to prompting the security warning and recommendations – by blurring out the remainder of the web page, whenever a copy event of an insecure code example is triggered.

We additionally apply a search filter which up-ranks posts that only contain secure code examples. Posts with insecure code examples are appended to the list of secure posts. The original ranking of posts within its security class is maintained. This approach lowers the risk of reusing code exam-

[5] We provide further example figures of our nudges in the Appendix.



(a) Use cases before training (b) Use cases after training (c) Security before training. (d) Security after training.

Figure 5: Visualizing the pattern embeddings of different use cases and security using PCA. Each color indicates one use case in (a) and (b), and security in (c) and (d). The legend provides the use case identifier.

ples that have been predicted to be insecure. This also means that whenever a post consists of secure and insecure samples it is ranked lower than posts with only secure samples.

7 Model Evaluation

7.1 Pattern Similarity

We evaluate the learned pattern embeddings by measuring cosine similarity for all pairs in the validation set and calculate the receiver operating characteristic curve (ROC) given the similarity label $y \in \{-1, 1\}$ for each pair. Our approach reaches an optimal AUC of 0.978, which slightly outperforms *Gemini* with AUC of 0.971. *Gemini* was originally applied to similarity prediction of binary functions by learning embeddings for CFGs and may not be a suitable benchmark.

We observe that the model converges already after five epochs. For the remaining epochs, AUC stays around 0.978 and does not improve significantly. This allows for a very short training time, as five epochs only need 27 minutes on average on our system[6]. However, we choose the model with the best AUC for generating the pattern embeddings.

7.2 Use Case Classification

For training the use case and security models, we apply the dataset consisting of 16,539 pattern embeddings extracted from Stack Overflow split up into subsets for training (80%) and validation (20%). Note that the validation set is constructed such that none of its samples appear in the training set. Therefore, we evaluate the performance of use case prediction on unseen pattern embeddings.

Visualization To illustrate the transfer learning process, we plot the pattern embeddings in $2D$ using principal component analysis (PCA) before and after the training of the classification model. Figure 5(a) shows the complete set of pattern embeddings before training, displayed in the color of their use case. We observe that some use cases already build clusters in the plot, while others appear overlapping and intermixed. Therefore, we apply an additional neural network on top that leverages supervision on use cases in addition to the similarity knowledge preserved in the input embeddings. Figure 5(b) plots the pattern embeddings again after supervised training of the model. Here, we input the initial pattern embeddings into the trained model and extract the last hidden layer of the network to obtain new embeddings that preserve information about their use case. We observe that the new embeddings now create dense and separable clusters for each use case in the plot. The network has moved pattern embeddings that belong to the same use case closer together, and the resulting clusters further away from each other in the embedding space.

Accuracy The promising observations from the visualization of pattern embeddings are confirmed by the accuracy results of the classification model. We performed a grid search that revealed the optimal neighborhood size of $K = 5$. The average AUC for predicting the different use cases already achieves its optimum of 0.999 after 20 epochs. As already indicated by the PCA plots, pattern embeddings provide a very good representation of use cases as the average AUC for all classes before training (epoch zero) is already above 0.998. However, precision and recall of *IV, HNVOR* and *TM* start below 0.878 and have been improved up to above 0.986 within 30 epochs of training.

7.3 Security Classification

Visualization We start again with illustrating the transfer-learning process for security classification by plotting pattern embeddings before and after training. Figure 5(c) displays pattern embeddings before training with their respective security score, Figure 5(d) plots the new embeddings after training. Samples that were labeled as secure are depicted in blue, insecure samples in red. When comparing Figure 5(a) and Figure 5(c), we can already observe several secure and insecure clusters within the use case clus-

[6] Intel Xeon E5-2660 v2 ("Sandy Bridge"), 20 CPU cores, 240GB memory

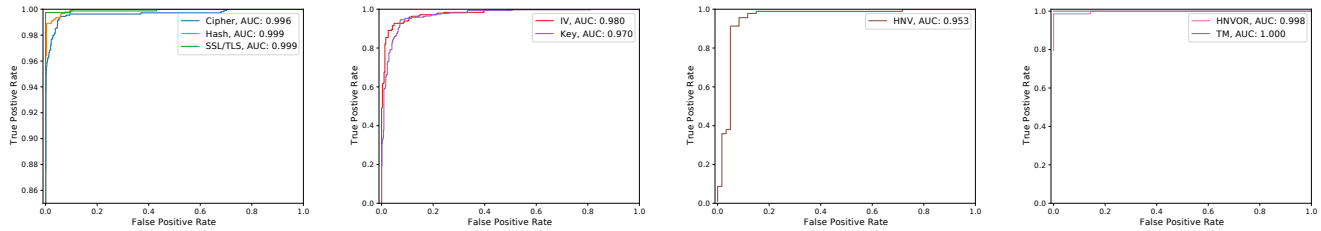


Figure 6: ROC for security classification of different use cases. The legend provides use case identifier and respective AUC.

ters, e. g., *Hash*, *Cipher* and *SSL/TLS*. However, again many secure and insecure samples appear to have a wide distribution because PCA does not plot them in dense clusters. After training the security classification model, we input the complete set of pattern embeddings and plot the last layer of the neural network for each sample in Figure 5(d) again. Now, we observe dense and separated clusters for secure and insecure samples. The network has adjusted the pattern embeddings such that samples within both security classes have been moved closer together in the embedding space. Samples with different security have been moved further away from each other, finally dividing samples into two security clusters.

Accuracy We trained a single model using the labeled dataset of 16,539 pattern embeddings. Thereby, a single model learns security classification for all use cases. Our grid search revealed $K = 5$ as the optimal neighbourhood size. The model provides a good fit because training and validation loss already converge after 50 epochs. A single epoch takes 0.58 seconds on average on our system, resulting in roughly five minutes for complete training time. We plot ROC curves for security prediction for each use case class in Figure 6. We observe that the three use cases *Hash*, *Cipher* and *SSL/TLS* that provide the largest percentage of samples in the dataset achieve the best results. The model achieves very good classification accuracy with AUC values of 0.999, 0.996 and 0.999, respectively, similar to *HNVOR* and *TM*. However, performance drops marginally for *IV*, *Key* and *HNV* to 0.980, 0.970 and 0.953, respectively.

Comparison In Table 2, we compare our approach on security prediction on Stack Overflow with [17], where the authors use *tf-idf* to create a feature vector as a representation for the complete input snippets and to train a SVM predicting its binary security score. Our deep learning approach (marked as CNN in the table) significantly outperforms their classifier in all use cases; especially *IV, Key* and *HNVOR*, where security evaluation heavily relies on data and control flow. In contrast to our approach, the work by Fischer et al. [17] does not inform the learning model about these properties, but solely relies on lexical features.

Moreover, our deep learning approach allows a higher level of explainability to the user. While [17] can only report security warnings for the complete snippet, our more fine-grained approach is able to directly highlight statements in the code and provides annotations that explain the security issue. Since we learn a representation of code patterns that allows prediction of different code properties beyond security, we can provide this additional explanation, which is crucial for developer advice.

	CNN		tfidf+SVM	
	AUC-ROC	Explanation	AUC-ROC	Explanation
Cipher	0.996	SW, CA	0.960	SW
Hash	0.999	SW, CA	0.956	SW
TLS	0.999	SW, CA	0.902	SW
IV	0.980	SW, CA	0.881	SW
Key	0.970	SW, CA	0.886	SW
HNV	0.953	SW, CA	0.922	SW
HNVOR	0.998	SW, CA	0.850	SW
TM	1.000	SW, CA	0.982	SW

Table 2: Performance and explainability comparison of security prediction on Stack Overflow. SW: Provides security warnings for the complete snippet. CA: Additionally provides code annotation that explains the issue in detail.

7.4 Recommendations

We applied our trained models in order to evaluate whether Stack Overflow provides secure alternative code snippets, which preserve the use case and are similar to detected insecure code examples. Thereby, we extracted all methods from the complete set of 10,558 snippets, generated their aggregated embeddings and separated them into two sets. The first set contains all 6,442 distinct insecure *query* embeddings and the second one all 3,579 distinct secure *target* embeddings. We created these two sets by applying the security model and predicted the security of each pattern within a given method. Finally, we ranked the embeddings based on their Jaccard distance, applying the use case model, and cosine similarity, as described in Section 6.2. We found 6,402 (99.37%) query methods that have Jaccard distance of 0.0 to at least one target method. This means that for almost every insecure method, a secure one exist on Stack Overflow

that serves the same use case. When additionally demanding code similarity, we found 6,047 (93.86%) query methods with a cosine similarity above 0.81 and 4,805 (75.58%) query methods with a similarity above 0.9 with at least one target method.

8 Evaluation of Security Nudges

To evaluate the impact of our system including the security nudges on the security of programming results, we perform a laboratory user study. Thereby, participants had to solve programming tasks with the help from Stack Overflow.

8.1 User Study Setup

Participants were randomly assigned to one of two treatment conditions. For the nudge treatment, we provided security warnings (Figure 2a) for insecure code examples, recommendations for secure snippets (Figure 2b) and recommendations lists attached to each warning (Figure 8). Further, security reminders were enabled. In the control treatment, all security nudges on Stack Overflow were disabled.

Participants were advised to use Stack Overflow to solve the tasks. In all treatments, we restricted Stack Overflow search results to posts that contain a code example from the set of 10,558 code examples we extracted from Stack Overflow[7]. Further, we applied a whitelist filter to restrict access to Stack Overflow in the Chrome browser. Any requests to different domains were redirected to the Stack Overflow search page. Participants were provided with the Google Chrome browser and Eclipse pre-loaded with two Java class templates. Both class templates provided code skeletons that were intended to reduce the participants' workload and simplify the programming tasks. Using additional applications was prohibited. All tasks had to be solved within one hour. We avoided security or privacy priming during the introduction and throughout the study. Moreover, we did not name or explain any of the security nudges on Stack Overflow.

8.2 Tasks

All participants had to solve five programming tasks related to symmetric encryption and certificate pinning. We chose these two use cases as they provide the most error-prone cryptographic problems in Android [17].

Symmetric Encryption The first three tasks dealt with initializing a symmetric cipher in order to encrypt and decrypt a message. Task *Cipher*: a symmetric cipher had to be initialized by setting the algorithm, block mode and padding. The main security pitfalls in this task are choosing a weak cipher and block mode. Task *Key*: a symmetric cryptographic key had to be generated. Participants had to create a key having the correct and secure key length necessary for the previously defined cipher. It had to be generated from a secure random source and should not have been stored in plaintext. Task *IV*: an initialization vector had to be instantiated. Like key generation, this task is particularly error-prone as choosing the correct length, secure random source and storage can be challenging.

Certificate Pinning Within these two tasks, a SSL/TLS context had to be created to securely communicate with a specific server via HTTPS. In the end, the program should have been able to perform a successful GET request on the server, while denying connection attempts to domains that provide a different server certificate. A solution for Task *TLS* would have been to select a secure TLS version to initialize the context. Task *TM*: the server's certificate had to be added to an empty custom trust manager replacing the default manager. This way, the program would pin the server's certificate and create a secure communication channel, while rejecting attempts to any other server with a different certificate.

8.3 Preliminaries and Participants

We advertised the study in lectures and across various university communication channels. 30 subjects participated in the study, however, three subjects dropped out, because they misunderstood a basic participation requirement (i.e., having at least basic Java programming knowledge). Of the remaining 27 subjects, 16 were assigned to the nudge treatment, and 11 to the control treatment. While being students, our sample varied across demographics and programming skill, but none of the self-reported characteristics systematically differed across the two treatments (see Appendix A for details).

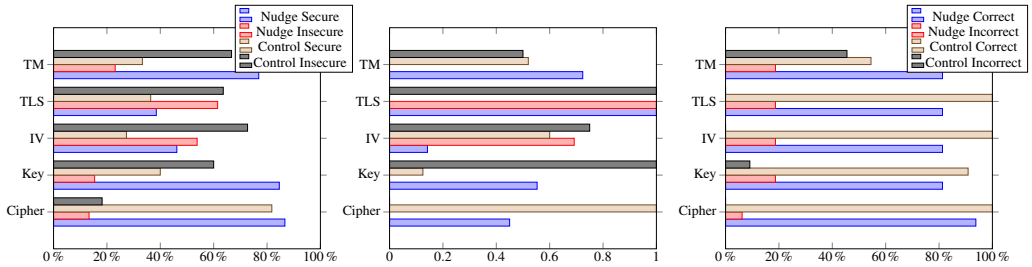
We followed well-established community principles for conducting security and privacy studies [29]. Participants were presented with a comprehensive consent form and separate study instructions on paper. Participants were compensated with 20 Euros.

After submission of the solutions, participants were asked to complete a short exit survey. We asked specific questions addressing the effectiveness of the security nudges and whether they were noticed by the participants. Also, we only asked demographic questions at this point to avoid any bias during the study. See Table 3 in the Appendix for details.

8.4 User Study Results

Functional Correctness Our system is not designed to address difficulties of programmers to deliver functionally correct code. However, it is important that using the system does not create obstacles to programmers. Participants predominantly submitted functionally correct code in both treatments

[7] This aims at simplifying search for participants. All Stack Overflow posts that contain a seed statement are available during the study.



(a) Security results (b) Copy-paste rate (average) (c) Correctness results

Figure 7: User study results for security, copy-paste rate and correctness of the submitted solutions across both treatments.

with some differences across tasks (cf. Figure 7c). Applying ordinal logistic regression (cf. Table 4 in the Appendix) indicates that the nudge treatment has – as anticipated – no effect on functional correctness of submitted tasks. However, non-professionals submitted significantly less functional code ($p < 0.05$). *Cipher* submissions are more often functional, irrespective of the treatment ($p < 0.05$).

Security Figure 7a shows the security results per task for both treatments. Performing ordinal logistic regression (see Table 5 in the Appendix), we show that the nudge treatment is significantly outperforming the control group in producing secure solutions (with an estimate of 1.303 and $p < 0.01$; Model 4). While the main effect of the nudge treatment dominates the regression models, we can observe from Figure 7a that comparatively more secure submissions are made for *TM* and *Key*. Indeed, pairwise testing using Chi-Square tests reveals $p < 0.001$ for both tasks. Participants from the nudge group provided 84.6% secure solutions for *Key* and 76.9% for *TM*, while 60.0% and 66.7% of the respective solutions submitted by the control group were insecure. These observations for *TM* are somewhat encouraging given previous findings: [17] have shown that reused insecure *TM* code snippets from Stack Overflow were responsible for 91% (183,268) of tested apps from Google Play being vulnerable. Only 0.002% (441) of apps contained secure *TM* code from Stack Overflow. Based on these insecure *TM* snippets, [13] were able to attack several high-profile apps extracting private data. Moreover, [27] found that only 45 out of 639,283 Android apps applied certificate pinning, while 25% of developers find certificate pinning too complex to use. [2] reported that tasks very similar to *TM* could not be solved with the help of simplified cryptographic APIs within a user study.

For *Cipher*, the nudge treatment performs very well, but only slightly better than the control treatment, as both achieved 86.7% and 81.8% secure solutions, respectively.

For *IV* and *TLS*, the nudge results are less desirable with 46.2% and 38.5% secure solutions, while performing better but not significantly ($p < 0.077$ and $p < 0.53$; Chi-Square) than the control treatment. To better understand these observations, we analyzed visited posts, copy-and-paste history and the submitted code of participants that provided insecure solutions for these two tasks. In the case of *IV*, we found that four insecure solutions reused insecure patterns from code snippets that were falsely marked as recommended code. To encounter false predictions of the security model was *a priori* extremely unlikely. Interestingly, the remaining insecure solutions were created by users combining secure code from different correctly marked recommendations (true negatives) into insecure code. Thereby, users reused the seed statement for *IV* from one snippet and initialized it with an empty array obtained from another code snippet that did not make use of *IV* at all. In the case of *TLS*, all insecure solutions were copied from code snippets that were clearly marked as insecure.

Copy-and-Paste Behavior We calculated the average copy-paste rate per task for both treatments, which reports the relative frequency copied code has also been reused in a submitted solution (see Figure 7b). Importantly, in the nudge treatment, *not a single* insecure copy-and-paste event was observed for *Cipher, Key* and *TM*, while secure code that was copied into the clipboard was reused at a rate of 0.45, 0.55, and 0.72 on average, respectively. This goes in line with observed security outcomes depicted in Figure 7a, where more secure than insecure solutions were provided for these tasks. However, insecure copy-and-paste events were observed for *IV* and *TLS*, partly explaining the comparatively higher number of insecure solutions. In the control treatment, the copy-paste rate for insecure snippets closely follows the observed frequencies of insecure results for all tasks except *Cipher*.

Warnings/Recommendations/Reminder Even though all users within the nudge group saw security warnings during their journey, we observed an insecure-to-secure copy event ratio of 0.27 for both treatments indicating that warnings alone are not sufficient for preventing users from placing insecure code on the clipboard. However, the copy-paste rate measuring the relative frequency of copy to paste events (see Figure 7b) offers more nuanced results. It shows that the nudge group tends to discard insecure copies, while pasting more secure copies into their solutions. This is most likely

the result of the reminder nudge, which was triggered by insecure copy events. As a result, users dropped copied insecure samples and started looking for a secure alternative. In contrast, the copy-paste rate for control shows that copied insecure snippets were not dropped, but rather pasted into the solution. Therefore, the interaction of several nudges was responsible for improving the security decisions of the participants. In the exit survey, users also marked the relevant nudges with high average Likert score values of above 4 (on 5-point scales).

We only observed 22 events where users clicked on a proposed recommendation link as shown in the bottom part of Figure 8a. Therefore, only 10.1% of secure posts (of 219 in total) were visited following such a proposed recommendation. However, 86.6% remembered the feature during the exit survey. With 4 paste and 8 copy events (i.e., a copy-paste rate of 0.5) only a very small amount of reused secure code in the submitted solutions was directly related to this nudge. Though contributing to the improvement of code security, we can state that this nudge was surprisingly the least effective one. The average Likert score for the list of recommendations was also comparatively low with 3.2.

9 Limitations

The response rate during recruitment for our developer study was quite low. However, we achieved a participation count per treatment which was very similar to comparable peer-reviewed studies (e.g., [2]). However, participation may introduce self-selection bias. Therefore, we avoided any security framing during recruitment and have no reason to believe that the final group of participants was systematically different in terms of security knowledge. The study was performed within a laboratory under strict time constraints. By enforcing a time limit, we intended to create a more realistic scenario and to obtain a comparable outcome for both treatments. Participants had to solve their programming tasks using a given code editor, browser, as well as operating system which they might not have been familiar with. Most of our participants were students, while only a minority had a professional background, which may limit the generalizability of our results. Professionals performed slightly better in achieving functional solutions, but not in security across both treatments. Therefore, comparisons among both treatments remain valid.

For implementing custom trust managers in Android (see Section 8.2), current best practices suggest a declarative solution which uses a static configuration file instead of Java code.[8] Being able to include other formats, such as formal documentation in our recommendations would additionally allow suggesting this solution. One possible way to achieve that is to create a link between code examples from Stack Overflow and natural language text in official documentation. I.e., we would have to extend our framework such that it embeds code examples and natural language text into the same vector space. This can be done with sequence-to-sequence models, which are usually applied for natural language translation. GitHub is currently testing a similar approach for their semantic code search engine.[9]

10 Future Work

Our recommendation approach may be subject to attacks. More specifically, in an adversarial setting, machine learning algorithms are often not robust against manipulated input data. Similar to efforts in malware obfuscation and spam filter bypassing, an attacker might be able to craft malicious code that gets mistakenly classified as secure. This way, the attacker could spread malicious code into the ecosystem on a large scale. However, a number of novel techniques have been proposed to counter the adversarial effect [18, 24, 28].

Stack Overflow provides code examples for almost each and every programming language. Since our framework learns the optimal code representation for a given classification task based on general code features, we do not see major issues in applying it to different programming languages. A language-specific compiler or a universal parser can be used to generate the PDG, which is then fed to our pattern embedding network (see Section 5.4). The representation learning of API-specific lexical features (see Section 5.3) is completely independent from the programming language and therefore straightforward.

We suggest to conduct additional UI testing as we might not have identified the optimal design, yet. Following Felt et al. [15], different security indicators such as alternative candidate icons and text have to be tested, for instance within user surveys or by repeating our developer study. Stack Overflow recently proposed a partnership program with academia that would allow to extend their developer survey and to test design tweaks on their website.[10]

11 Conclusion

In this paper, we propose an approach for deep learning security nudges that help software developers write strong encryption code. We propose a system design integrated in Stack Overflow whose components consist of several security nudges, namely *warnings*, *recommendations*, *reminders*, and *defaults* and a neural network architecture that controls these nudges by learning and predicting secure and insecure cryptographic usage patterns from community-provided

[8] https://developer.android.com/training/articles/security-config

[9] https://githubengineering.com/towards-natural-language-semantic-code-search/

[10] https://meta.stackoverflow.com/questions/377152/stack-overflow-academic-research-partnership-program

code examples. We propose a novel approach on deep learning optimized code representations for given code classification tasks and train a classification model that is able to predict use cases and security scores of encryption code examples with an AUC-ROC of 0.999 and 0.992, respectively. Applying this model within our nudge-based system design on Stack Overflow, we performed a user study where participants had to solve the most error-prone cryptographic programming tasks reported in recent research. Our results demonstrate the effectiveness of nudges in helping software developers to make better security decisions on Stack Overflow.

Acknowledgements

The authors would like to thank Fraunhofer AISEC for technical support, DIVSI for support of our research efforts, and the anonymous reviewers for their helpful comments.

References

[1] ACAR, Y., BACKES, M., FAHL, S., GARFINKEL, S., KIM, D., MAZUREK, M. L., AND STRANSKY, C. Comparing the usability of cryptographic APIs. In *IEEE Symposium on Security and Privacy* (2017), pp. 154–171.

[2] ACAR, Y., BACKES, M., FAHL, S., KIM, D., MAZUREK, M. L., AND STRANSKY, C. You Get Where You're Looking For: The Impact of Information Sources on Code Security. In *IEEE Symposium on Security and Privacy* (2016), pp. 289–305.

[3] ACER, M., STARK, E., FELT, A. P., FAHL, S., BHARGAVA, R., DEV, B., BRAITHWAITE, M., SLEEVI, R., AND TABRIZ, P. Where the wild warnings are: Root causes of Chrome HTTPS certificate errors. In *ACM Conference on Computer & Communications Security* (2017), pp. 1407–1420.

[4] ACQUISTI, A. Nudging privacy: The behavioral economics of personal information. *IEEE Security & Privacy 7*, 6 (2009), 82–85.

[5] ACQUISTI, A., ADJERID, I., BALEBAKO, R., BRANDIMARTE, L., CRANOR, L. F., KOMANDURI, S., LEON, P. G., SADEH, N., SCHAUB, F., SLEEPER, M., ET AL. Nudges for privacy and security: Understanding and assisting users' choices online. *ACM Computing Surveys 50*, 3 (2017), Article No. 44.

[6] ALMUHIMEDI, H., SCHAUB, F., SADEH, N., ADJERID, I., ACQUISTI, A., GLUCK, J., CRANOR, L. F., AND AGARWAL, Y. Your location has been shared 5,398 times! A field study on mobile app privacy nudging. In *ACM Conference on Human Factors in Computing Systems* (2015), pp. 787–796.

[7] BALEBAKO, R., LEON, P. G., ALMUHIMEDI, H., KELLEY, P. G., MUGAN, J., ACQUISTI, A., CRANOR, L. F., AND SADEH, N. Nudging users towards privacy on mobile devices. In *CHI Workshop on Persuasion, Nudge, Influence and Coercion* (2011), pp. 193–201.

[8] BÖHME, R., AND GROSSKLAGS, J. The security cost of cheap user interaction. In *ACM New Security Paradigms Workshop* (2011), pp. 67–82.

[9] CHEN, M., FISCHER, F., MENG, N., WANG, X., AND GROSSKLAGS, J. How reliable is the crowdsourced knowledge of security implementation? In *ACM/IEEE International Conference on Software Engineering* (2019).

[10] DAGENAIS, B., AND HENDREN, L. Enabling static analysis for partial Java programs. *ACM Sigplan Notices 43*, 10 (2008), 313–328.

[11] DAI, H., DAI, B., AND SONG, L. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning* (2016), pp. 2702–2711.

[12] EGELE, M., BRUMLEY, D., FRATANTONIO, Y., AND KRUEGEL, C. An empirical study of cryptographic misuse in Android applications. In *ACM Conference on Computer & Communications Security* (2013), pp. 73–84.

[13] FAHL, S., HARBACH, M., MUDERS, T., SMITH, M., BAUMGÄRTNER, L., AND FREISLEBEN, B. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *ACM Conference on Computer & Communications Security* (2012), pp. 50–61.

[14] FAHL, S., HARBACH, M., PERL, H., KOETTER, M., AND SMITH, M. Rethinking SSL development in an appified world. In *ACM Conference on Computer & Communications Security* (2013), pp. 49–60.

[15] FELT, A. P., REEDER, R., AINSLIE, A., HARRIS, H., WALKER, M., THOMPSON, C., ACER, M. E., MORANT, E., AND CONSOLVO, S. Rethinking connection security indicators. In *Symposium on Usable Privacy and Security* (2016), pp. 1–14.

[16] FENG, Q., ZHOU, R., XU, C., CHENG, Y., TESTA, B., AND YIN, H. Scalable graph-based bug search for firmware images. In *ACM Conference on Computer & Communications Security* (2016), pp. 480–491.

[17] FISCHER, F., BÖTTINGER, K., XIAO, H., STRANSKY, C., ACAR, Y., BACKES, M., AND FAHL, S. Stack overflow considered harmful? The impact of copy&paste on Android application security. In *IEEE Symposium on Security and Privacy* (2017).

[18] GANIN, Y., USTINOVA, E., AJAKAN, H., GERMAIN, P., LAROCHELLE, H., LAVIOLETTE, F., MARCHAND, M., AND LEMPITSKY, V. Domain-adversarial training of neural networks. *Journal of Machine Learning Research 17*, 59 (2016), 1–35.

[19] GROSSKLAGS, J., RADOSAVAC, S., CÁRDENAS, A., AND CHUANG, J. Nudge: Intermediaries role in interdependent network security. In *International Conference on Trust and Trustworthy Computing* (2010), pp. 323–336.

[20] KRÜGER, S., SPÄTH, J., ALI, K., BODDEN, E., AND MEZINI, M. CrySL: An extensible approach to validating the correct usage of cryptographic apis. In *European Conference on Object-Oriented Programming* (2018), pp. 10:1–10:27.

[21] LI, Z., ZOU, D., XU, S., OU, X., JIN, H., WANG, S., DENG, Z., AND ZHONG, Y. VulDeePecker: A deep learning-based system for vulnerability detection. In *Network and Distributed Systems Security Symposium* (2018).

[22] LIU, B., ANDERSEN, M. S., SCHAUB, F., ALMUHIMEDI, H., ZHANG, S. A., SADEH, N., AGARWAL, Y., AND ACQUISTI, A. Follow my recommendations: A personalized privacy assistant for mobile app permissions. In *Symposium on Usable Privacy and Security* (2016), pp. 27–41.

[23] LIU, B., LIN, J., AND SADEH, N. Reconciling mobile app privacy and usability on smartphones: Could user privacy profiles help? In *International Conference on World Wide Web* (2014), pp. 201–212.

[24] MIYATO, T., MAEDA, S., KOYAMA, M., NAKAE, K., AND ISHII, S. Distributional smoothing by virtual adversarial examples. *CoRR abs/1507.00677* (2015).

[25] NGUYEN, D. C., WERMKE, D., ACAR, Y., BACKES, M., WEIR, C., AND FAHL, S. A stitch in time: Supporting Android developers in writing secure code. In *ACM Conference on Computer and Communications Security* (2017), pp. 1065–1077.

[26] OLIVEIRA, D. S., LIN, T., RAHMAN, M. S., AKEFIRAD, R., ELLIS, D., PEREZ, E., BOBHATE, R., DELONG, L. A., CAPPOS, J., AND BRUN, Y. API Blindspots: Why experienced developers write vulnerable code. In *Symposium on Usable Privacy and Security* (2018), pp. 315–328.

[27] OLTROGGE, M., ACAR, Y., DECHAND, S., SMITH, M., AND FAHL, S. To pin or not to pin – Helping app developers bullet proof their TLS connections. In *USENIX Security Symposium* (2015), pp. 239–254.

[28] PAPERNOT, N., MCDANIEL, P., WU, X., JHA, S., AND SWAMI, A. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE Symposium on Security and Privacy* (2016), pp. 582–597.

[29] SCHECHTER, S. Common pitfalls in writing about security and privacy human subjects experiments, and how to avoid them. Tech. rep., Microsoft Research, 2013.

[30] SUNSTEIN, C. Nudging: A very short guide. *Journal of Consumer Policy 37*, 4 (2014), 583–588.

[31] THALER, R., AND SUNSTEIN, C. *Nudge: Improving decisions about health, wealth, and happiness*. Penguin, 2008.

[32] WANG, Y., LEON, P. G., SCOTT, K., CHEN, X., ACQUISTI, A., AND CRANOR, L. F. Privacy nudges for social media: An exploratory Facebook study. In *International Conference on World Wide Web* (2013), pp. 763–770.

[33] XU, X., LIU, C., FENG, Q., YIN, H., SONG, L., AND SONG, D. Neural network-based graph embedding for cross-platform binary code similarity detection. In *ACM Conference on Computer & Communications Security* (2017), pp. 363–376.

Appendix A: Additional Participant Data

Table 3 includes additional data about the 27 participants, who completed the study.

We also conducted a series of statistical tests to verify that the self-reported characteristics of the recruited participants did not systematically vary across treatments. Indeed, using the Mann-Whitney U Test, we found that participants did not differ in their reported age across treatments ($p = 0.79$). Applying Fisher's Exact Test, we also observed the absence of a statistically significant difference for country of origin ($p = 0.809$), gender ($p = 0.551$), level of education ($p = 0.217$), security knowledge/background ($p = 0.124$), and professional programming experience ($p = 0.315$). Using the Mann-Whitney U Test, we did not find any statistically significant difference for years of experience with Java programming ($p = 0.422$). We also did not find any reportable differences regarding participants' awareness of encryption flaws ($p = 0.363$) using Fisher's Exact Test. The percentage of participants who had to program Java as primary activity for their work ($p = 1$) or for whom writing Java code was part of their primary job in the last 5 years ($p = 0.696$) also did not differ across treatments (using Fisher's Exact Test).

Appendix B: Detailed Regression Results

Based on the user study data and self-reported survey responses, we follow an ordinal (Logit link) regression ap-

proach, which is primarily focused on evaluating the effectiveness of the nudge treatment.

First, we report a series of four models (M1 - M4) to evaluate whether the nudge treatment significantly impacts the *functional correctness* of the submitted programs for the five different tasks (see Table 4). We iteratively add factors to the regression model to also test whether programming expertise or security expertise positively impact the outcome variable. Most importantly, as the nudge treatment is not designed to address this aspect of programming, we did *not* expect any significantly positive effect. Indeed, across all model specifications that we tested, we did not observe any significant (positive or negative) effect. Regarding the different programming tasks, we found that the Cipher task was associated with a significantly increased likelihood of being functionally correct (M2 - M4). Further, not being a security professional (as reported by the participants) significantly impacts the likelihood that functional programs were submitted in a negative fashion (M3 - M4). In contrast, a higher degree of security knowledge (as reported by the participants) did not significantly impact the results (M4).

Note that the regression statistics for tasks IV and TLS are identical as the aggregate results for functional correctness happen to be the same (see Figure 7c).

Age				
Mean = 22.93	Median = 22	Stddev = 3.9	Min = 19	Max = 38
Country of Origin				
Germany = 16				Other = 11
Gender				
Male = 9				Female = 18
Achieved Level of Education				
Highschool = 15	Bachelor = 8		Master = 3	Ph.D. = 0
Professional at Programming				
Yes = 12	No = 15			
Security Background				
Yes = 10	No = 17			
Java Years Experience				
Mean = 3.81	Median = 3	Stddev = 2.304	Min = 1	Max = 8
Encryption Flaw Awareness				
Yes = 17	No = 10			
Java primary focus of job				
Yes = 5	No = 21	No Data = 1		
Java part of any job				
Yes = 12	No = 15			

Table 3: Detailed data about demographics of participants (N = 27). One missing response for the question whether Java is primary focus of current job.

Second, we report a series of four models (M1 - M4) to evaluate whether the nudge treatment significantly impacts the *security* of the submitted programs for the five different tasks (see Table 5). For consistency, we iteratively add the same factors to the regression model to also test whether programming expertise or security expertise positively impact the outcome variable.

Most importantly, as the nudge treatment is designed to improve the security of cryptography-related programming, we did expect a significantly positive effect. Indeed, across all model specifications that we tested, we did observe a significant and positive effect. Regarding the different programming tasks, we did not find that they significantly differed from each other regarding the security property (M2 - M4). Being a security professional did not impact the security of the submitted programs in a significant way (M3 - M4). Perhaps surprisingly, a higher degree of security knowledge (as

FACTORS	**M1**	**M2**	**M3**	**M4**
Treatment: Nudge	-0.460	-0.489	-0.263	-0.226
	(0.523)	(0.544)	(0.568)	(0.605)
Task: Cipher	-	**2.407***	**2.539***	**2.539***
		(1.105)	(1.125)	(1.125)
Task: IV	-	1.224	1.324	1.324
		(0.746)	(0.775)	(0.775)
Task: Key	-	0.892	0.974	0.974
		(0.690)	(0.72)	(0.721)
Task: TLS	-	1.224	1.324	1.324
		(0.746)	(0.775)	(0.775)
Not Professional	-	-	**-1.701***	**-1.698***
			(0.679)	(0.680)
Sec. Knowledge	-	-	-	-0.106
				(0.605)

Table 4: Results for Ordinal Regression of Functional Correctness. Series of non-interaction models (M1 – M4) with factors iteratively added. Significant values are highlighted in bold, and marked with: * $p < 0.05$. Standard errors are included in parentheses. The baseline for Treatment is Control (i.e., the unmodified Stack Overflow), and the baseline for Task is TM.

FACTORS	**M1**	**M2**	**M3**	**M4**
Treatment: Nudge	**0.920***	**1.018***	**1.113***	**1.303****
	(0.388)	(0.426)	(0.438)	(0.480)
Task: Cipher	-	1.388	1.377	1.405
		(0.745)	(0.754)	(0.758)
Task: IV	-	-0.963	-1.001	-0.990
		(0.654)	(0.665)	(0.668)
Task: Key	-	0.224	0.200	2.13
		(0.668)	(0.677)	(0.679)
Task: TLS	-	-0.963	-1.001	-0.990
		(0.654)	(0.665)	(0.668)
Not Professional	-	-	-0.702	-0.686
			(0.432)	(0.434)
Sec. Knowledge	-	-	-	-0.517
				(0.481)

Table 5: Results for Ordinal Regression of Security. Series of non-interaction models (M1 – M4) with factors iteratively added. Significant values are highlighted in bold, and marked with: * $p < 0.05$ and ** $p < 0.01$. Standard errors are included in parentheses. The baseline for Treatment is Control (i.e., the unmodified Stack Overflow), and the baseline for Task is TM.

(a) Security warning provided by the security and use case model

(b) Recommendation provided by the similarity and use case model.

Figure 8: Security warning and recommendations provided by the similarity, use case and security model. The security model predicted the usage pattern of *setHostnameVerifier* as insecure. Further, it predicted its use case *HNV*, being able to select and display the related security annotation under the insecure statement. Below the security warning the similarity, use case and security model provide the ranked list of recommendations, that contains code examples with similar and secure patterns of *HNV*. We display the recommended code example that appears when clicking on the first link in (b).

reported by the participants) did not significantly impact the results either (M4).

We created regression models including further demographic and explanatory variables. However, none of them had a significant effect on the security of submitted solutions.

Appendix C: Pattern Annotation Tool

Our security annotations generally comply with rules and annotation heuristics given by [12, 13, 17]. However, manual analysis of patterns was not restricted to simple application of these heuristics, but was based on detecting insecure patterns in general. Whenever an unknown pattern has been detected, both annotators discussed them until agreement on a label. For example, [13] only reports empty trust manager implementations, while many insecure *TM* patterns on Stack Overflow are not empty, but provide insufficient certificate verification (e. g., only validating that the certificate is not expired).

To further speed up the labeling process and manage the large amount of samples, we created a code annotation tool. It automatically iterates through code snippets and displays them to the user, using a source code editor. Seed statements were already highlighted in order to allow the annotator to detect relevant patterns quickly. The annotator was able to assign labels (e. g., secure/insecure) to different keyboard buttons. While iterating through the seed statements, the annotator would investigate the related pattern and label it accordingly. Moreover, the annotator had the option to add seed statements, that she wanted to have highlighted and labeled. Whenever the annotator identified new patterns or wanted to share and discuss a pattern, the related code snippet was marked and other annotators were notified to comment on it. After agreement, the pattern was labeled by the initial annotator. Further, annotation heuristics obtained during the discussion were shared among all annotators.