Towards Secure Integration of Cryptographic Software

Steven Arzt[1] Sarah Nadi[2] Karim Ali[1]
Eric Bodden[1] Sebastian Erdweg[2]
Mira Mezini[2]

[1]Secure Software Engineering Group, Technische Universität Darmstadt, Germany
[2]Software Technology Group, Technische Universität Darmstadt, Germany
[1]firstname.lastname@cased.de
[2]lastname@stg.tu-darmstadt.de

Abstract

While cryptography is now readily available to everyone and can, provably, protect private information from attackers, we still frequently hear about major data leakages, many of which are due to improper use of cryptographic mechanisms. The problem is that many application developers are not cryptographic experts. Even though high-quality cryptographic APIs are widely available, programmers often select the wrong algorithms or misuse APIs due to a lack of understanding. Such issues arise with both simple operations such as encryption as well as with complex secure communication protocols such as SSL. In this paper, we provide a long-term solution that helps application developers integrate cryptographic components correctly and securely by bridging the gap between cryptographers and application developers.

Our solution consists of a software product line (with an underlying feature model) that automatically identifies the correct cryptographic algorithms to use, based on the developer's answers to high-level questions in non-expert terminology. Each feature (i.e., cryptographic algorithm) maps into corresponding Java code and a usage protocol describing API restrictions. By composing the user's selected features, we automatically synthesize a secure code blueprint and a usage protocol that corresponds to the selected usage scenario. Since the developer may change the application code over time, we use the usage protocols to statically analyze the program and ensure that the correct use of the API is not violated over time.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

Onward!'15, October 25–30, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3688-8/15/10...
http://dx.doi.org/10.1145/2814228.2814229

Categories and Subject Descriptors D.2.2 [*Design Tools and Techniques*]: Software Libraries; F.3.2 [*Semantics of Programming Languages*]: Program analysis

General Terms Security, Languages

Keywords Software product lines, typestate analysis, API protocols, cryptography

1. Introduction

The Java Cryptography Architecure (JCA)[1], containing the Java Cryptography Extension (JCE), is designed to allow Java application developers to easily use cryptography. JCE separates the Application Programming Interfaces (APIs) developers use from the underlying implementations that can be provided by any provider (e.g., Java's default implementation, BouncyCastle[2] or FlexiProvider[3]). Despite such design efforts, the JCE APIs themselves offer a broad variety of different algorithms that in turn support a multitude of modes and configuration options. Additionally, each provider may support additional algorithms or worse, provide different default values for the same JCE API call. As a result, many Java software developers are challenged by the task to use and compose these API components correctly.

Previous research has already identified severe security vulnerabilities due to misuses of cryptographic APIs [25, 27]. The problem is that application developers typically lack cryptographic expertise while cryptographic libraries embody highly specialized knowledge that they fail to expose to clients at the appropriate level of abstraction. Suggested solutions to this problem include better API documentation [24, 27] and the use of program-analysis tools that check for certain misuse-related vulnerabilities [24, 25]. While both these solutions decrease the number of security vulnerabilities, proper documentation is hard to enforce and

[1] `http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html`

[2] `www.bouncycastle.org`

[3] `www.flexiprovider.de`

API developers are often bad at documenting their code [40]. On the other hand, even running a combination of multiple analysis tools will likely catch only some of the possible vulnerabilities.

In this paper, we propose a more comprehensive, long-term solution that combines the advantages of documentation and program analysis with ease of use and evolvability. Our solution separates API users from the domain knowledge required to understand these APIs through an expert system, the Open CROSSING Crypto Expert (*OpenCCE*).[4] OPENCCE (1) guides developers through selecting the relevant cryptographic components to use, (2) automatically generates the required code with the correct API calls for them, and (3) analyzes the final program to ensure that no threats have been introduced, neither during initial development, nor during program evolution. Note that the main goal of OPENCCE is not to detect intentional, malicious code, but to avoid unintentional mistakes by non-expert developers.

Our solution relies on the observation that most cryptographic libraries provide at least some degree of compile-time variability which, at least on a conceptual level, can be considered a software product line (SPL). *Software product lines* provide a systematic way of generating similar, yet different, products [20]. Cryptographic algorithms typically come with a large number of configuration options that select a number of different algorithmic variations, each of which may come with its own pre-conditions, post-conditions, security parameters, guarantees, and other API-related usage requirements. Most cryptographic solutions (e.g., securing a password, proving the authenticity of a message, encrypting a file using a password, etc.) are essentially parameterized combinations of common cryptographic algorithms. Thus, creating a cryptographic solution can be thought of as generating one product from a family of possible products.

We build OPENCCE as a tool for managing an SPL, where cryptographic components are abstracted through a *feature model* [32]. In OPENCCE, the feature model encapsulates domain knowledge such as what the different cryptographic algorithms are, which class of problems they can be applied to, and their limitations or dependencies. For example, several encryption algorithms have trade-offs between security, speed, and memory usage. Some algorithms depend on other algorithms; a digitial signatue for instance requires not only the signature algorithm itself, but also a hash function. However, not all combinations of signature algorithms and digests yield a secure digital signature. Furthermore, since correctly using an API to accomplish a task often requires calls to several methods in a specific order [41], we also express the *usage protocol* of such components in a separate specification that is linked to the feature model. For example, a usage protocol may specify that the application must first check the issuer of a certificate before passing this certificate on to the authentication primitive. Such usage protocols can then be translated into static analyses that automatically ensure correct integration. As developers extend and modify their applications over time, those static analyses are not only executed once, but continuously throughout the development and maintenance process.

[4] CROSSING is the name of the collaborative research center within which OpenCCE will be developed.

```
1  //custom method getRandomKey(keyLength, seed)
2  //to generate alphanumeric key of keyLength bits
3  //based on Java's Random class seeded with seed
4  byte[] key = getRandomKey(192, 1873);
5
6  Cipher cipher = Cipher.getInstance("AES");
7
8  SecretKey keyObj = new SecretKeySpec(key,"AES");
9
10 cipher.init(Cipher.ENCRYPT_MODE, keyObj);
11 byte[] cipherText = cipher.doFinal(input);
```

Figure 1. Example of using an AES cipher

The novelty of OPENCCE is that it provides a long-term solution to many security problems by integrating, for the first time, various well-established research areas in software-engineering and program-analysis with cryptographic domain knowledge. OPENCCE focuses on the cryptography domain because of the grave consequences that can result from cryptographic misuse, but the general idea can also be applied to other domains that require such expert knowledge. OPENCCE bridges the gap in both knowledge and vocabulary between domain experts and software developers.

2. Motivating Scenario

Let us meet Alice, a typical Java programmer with no cryptography background. As part of a web application she is developing, Alice has to securely (1) send messages and (2) store user passwords. Alice searches for ways to accomplish this in Java and realizes she needs to use the Java Cryptography Extension (JCE) APIs. While Alice does her best to make the code secure, she faces a number of challenges that make it likely for her to introduce vulnerabilities.

2.1 Securely Sending Messages

Through searching the web and asking colleagues, Alice realizes she requires a *cipher* to encrypt and decrypt the messages she needs to send securely. Because it is known to be fast, Alice decides to use the Advanced Encryption Standard (AES) cipher [22] and implements the code in Figure 1 which focuses only on the encryption part of the task. However, the code shown has the following problems.

First, Alice uses `java.util.Random` to generate the key she will use for encryption (Lines 1 - 4). She is not aware that the `Random` class is a weak *pseudo-random number generator* with outputs that are not independent. An attacker who observes only a few values can thus predict all future

ones. Additionally, the seed used for generating the key is the same for all encryption operations carried out by Alice's program. This reduces the effort for an attacker to break the cipher as the seed only needs to be guessed once. Even worse, this single seed is not even random, but hard-coded into the application and shared between all installations of the program. An attacker can simply obtain the seed by decompiling one instance of the program. Not specifying a custom seed, but rather relying on Java's default seeding would not have solved the issue either as the default seed is based on the system time which can be easily brute-forced. What Alice really needs is a *cryptographically secure* pseudo-random number generator combined with a proper source of entropy (e.g., output from `/dev/random` on Unix systems) for the seed. To this end, Alice should be using the Java API `java.security.SecureRandom` with the appropriate parameters.

Second, being unaware of the library's defaults, Alice makes a fatal mistake while initializing the AES cipher. If no block cipher mode is explicitly specified by the developer, the JCE uses the now known to be insecure *Electronic Codebook (ECB)* block cipher mode. ECB mode causes two equal blocks of plaintext to be encrypted to two equal blocks of ciphertext. Thus, patterns that are present in the plaintext are retained in the ciphertext, which can leak a substantial amount of information [24].

2.2 Securely Storing User Passwords

Alice also needs to securely store user passwords in a database for later authentication. As far as Alice is concerned, securely storing user passwords means that they should not be stored as plain text and it should be hard for an attacker to extract them. She, therefore, thinks that her encryption code above can also work for encrypting passwords and storing them in a database. When the user is authenticating, she can then decrypt the password and compare it to what the user enters.

While nothing is technically wrong with using encryption code to store a password, cryptography experts would argue that storing a hashed password is better since hashing is a one-way function (i.e., cannot be reversed) while encryption using symmetric ciphers can be reversed with access to the key. In that particular example, attackers could decompile the code to get access to the (deterministic) seed information and eventually the key. Hence, it is better for Alice to use a hashing algorithm in such a situation [4, Chapter 2.5.4].

However, merely using a hash function like SHA-1 on the password is not an optimal solution either as this would allow for efficient brute-force attacks using pre-computed tables such as Rainbow Tables [37]. Instead, she needs to use *salts*. A cryptographic salt is a random string that is generated for one specific password and that serves as an additional input to the hash function. Cryptographic libraries for Java (and other languages) typically provide a number of key-derivation functions that apply hashing algorithms in a sensible way, incorporating salts and often iterating the hashing multiple times to further harden the implementation against brute-force attacks that might become computationally feasible in the future. But how should Alice know? And even if she did know, how could she ensure to at least use *those* APIs correctly?

2.3 Problem Statement

We identify four separate, yet complementary, problems from the scenarios presented above. These problems motivate the goals we address with OpenCCE's design (Section 3).

(1) Developers need to spend considerable time to learn about cryptography. In our scenario, Alice needed to spend valuable time to search and understand how cryptography works and which algorithms to use to complete her tasks. This presents a large overhead for developers whose main expertise does not lie in the cryptography domain. At best, Alice spends considerable amount of time learning about the field and finds the right answers. At worst, Alice spends considerable time, but her understanding of cryptography methods and algorithms is flawed, making her application vulnerable to security attacks. Also, what about times in which a security assessment of a cryptographic algorithm changes, like in the cases of ECB or SHA-1, both of which were once thought to be reasonably secure? Will Alice, and all other developers for that matter, have to monitor cryptographic research papers for the rest of their lives?

> **Goal 1:** Reduce adoption barrier for non-crypto experts by *guiding developers to find proper solutions for their cryptography needs*.

(2) Developers are often unsure about the best cryptographic algorithm to use. As shown in Section 2.2, developers may have semantic misconceptions about the cryptography domain. That is, they believe that a particular algorithm or method is best suited for their application while in reality, it might conceptually be wrong. Another technique might be preferred or it might be completely insecure for that particular use. In practice, many developers still seem to believe that a secret is properly encrypted as long as it is somehow garbled.

> **Goal 2:** Present cryptographic information *using a terminology developers understand* and assist them with chosing appropriate components and component combinations from the design space.

(3) Developers misuse cryptographic APIs. Even after establishing some basic knowledge about cryptography and identifying the relevant APIs to use, developers often misuse these APIs. In a recent study, Egele et al. [24] found that 88% of the Android apps in the Google Play store that use

cryptography misuse the APIs in at least one way.[5] This shows how prevalent this problem is. While cryptographic libraries prevent developers from having to re-invent the wheel, the developers' lack of a deep understanding of the field (in addition to poor documentation) is still a barrier to the proper use of these APIs.

> **Goal 3:** Support *automatic generation of code that uses cryptography securely*.

(4) Code changes often break security. Even if code that uses cryptography is correctly generated, developers still need to integrate it into their applications. Without in-depth knowledge of how the crypto libraries are supposed to be used, this can lead to the introduction of new vulnerabilities. Furthermore, as applications evolve, API misuses, and therefore security vulnerabilities, may arise during later maintenance of the application. Adding new features might inadvertently break the security of existing features or the overall system due to poor understanding of the security model or the implications of the various cryptographic APIs used in the program. The mere information that a security issue exists is still not sufficient. The developer is rather interested in what change exactly broke security, what vulnerabilities it may have introduced, and how the problem can be fixed. For non-experts, these questions are hard to answer without further assistance.

> **Goal 4:** Support *automatic validation* of code that uses cryptography. This applies to both the initially generated integration code and to the context of *evolving host applications*. Explain discovered issues in a way *understandable to the developer* and *assist in resolving them*.

3. OpenCCE

To achieve the four goals discussed above, we propose OpenCCE, an interactive Eclipse plugin based on a software product line with an underlying *feature model* [32]. The goal of OpenCCE is to allow the automatic composition of such features based on user requirements defined as answers to high-level questions. This design accounts for the fact that most developers are not cryptography experts and are unlikely to be able to correctly pick and compose components manually.

We organize and structure the whole product line as well as all software artifacts involved in terms of *features*, units of functionality in the system [6]. Each cryptographic algorithm can be thought of as a feature (e.g., AES, RSA, SHA-1, etc.). The relationships between the various features can then be encoded in a feature model. Each feature in OpenCCE maps to two different artifacts: the code with the API calls

[5] Interestingly, especially for commercial apps with such flaws it is nevertheless quite common to advertise the usage of "military-grade cryptography".

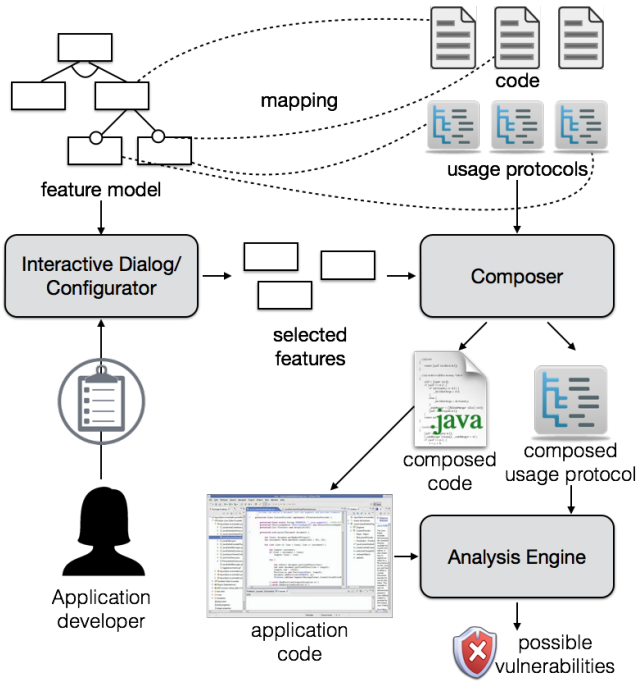

Figure 2. OpenCCE components and workflow

corresponding to the respective algorithm and a usage protocol. The *usage protocol* describes the technical restrictions on using the APIs for this algorithm (e.g., AES should not be used with ECB) as well as proper uses of the API. These restrictions are used to generate the static analyses that ensure the secure integration of the respective cryptographic algorithm during program evolution.

Figure 2 shows an overview of how OpenCCE works. The grey boxes show the main components of OpenCCE. An application developer (bottom left) first uses the *configurator* to specify her requirements. For example, OpenCCE would ask the developer *"Which one of the following tasks would you like to accomplish?"* The developer can then choose from a list of predefined high-level *tasks* such as *store a password, securely transfer a file, securely store a file,* etc. More details can then be elicited from the developer as required to further refine the selection to reach the right combination of algorithms needed to perform the task. For example the developer can also specify non-functional requirements such as low memory usage. Such an abstraction matches the developer's understanding of the task she has to accomplish without the need to understand cryptographic jargon.

The selected features are then passed to the *composer* that composes both the corresponding code files as well as usage protocols. The result is a composed piece of code that is added to the developer's application code as well as a composed usage protocol. This usage protocol is used by the *analysis engine* to run the static analyses needed to ensure correct integration.

```
// Key generation code
SecureRandom random = new SecureRandom();
byte[] salt = new byte[32];
random.nextBytes(salt);
PBEKeySpec spec = new PBEKeySpec(pwdChar, salt,
      1000, 128);
SecretKeyFactory skf =
      SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
SecretKey key = skf.generateSecret(spec);

// Encryption code
Cipher cipher =
      Cipher.getInstance("AES/CBC/PKCS5PADDING");
cipher.init(Cipher.ENCRYPT_MODE, key);
byte[] cipherText = cipher.doFinal(inputMsg);
```

Figure 3. Example of password-based encryption which requires the composition of a key derivation algorithm and a cipher

We now provide more details about the workflow outlined above. Throughout our explanations, we will use the scenario of a developer who wants to encrypt a message using a password. To accomplish such a task, both a key derivation algorithm and an encryption cipher are needed. The code that should be generated to accomplish this task is shown in Figure 3. This example can be seen as the correct version of the motivating example from Figure 1 in which AES was used in an insecure way.

3.1 Feature Model

In order to match the developer's requirements to cryptographic components, we need to capture the needed cryptography domain knowledge in a feature model that can be automatically reasoned about. After performing some domain analysis together with domain experts, we identify the following modeling notation requirements:

1. *Support hierarchal structure.* Encryption ciphers can be symmetric or asymmetric. Yet, they share common properties such as a performance rating. Symmetric ciphers in turn can operate on blocks or streams, but share the requirement of a secret key. On a higher level, there are different algorithm classes in cryptography such as encryption ciphers versus digests (used for hashing).
2. *Support non-Boolean values.* For most algorithms, we need to specify attributes such as key size, memory consumption levels, etc.
3. *Support reuse through some form of referencing.* Many tasks may need an encryption cipher, for example, and we do not want to create a new feature representing a cipher for each of these tasks. Instead, we want to have a single feature representing the cipher and reference it in the tasks. Similarly, on a finer granularity level, any symmetric cipher should specify the key length(s) it supports irrespective of whether it is a stream or block cipher. Thus, it would be simpler if we can model this as an attribute of symmetric ciphers in general and reuse it (similar to inheritence) instead of redefining it in each cipher.
4. *Support modeling commonalities and variabilities.* For example, two hashing algorithms have the same purpose, but may support different output lengths.
5. *Support automated reasoning.* We need to specify the user's requirements as constraints and get the algorithms that satisfy these constraints as a solution. In addition to basic reasoning of finding instances, we also need support for objective optimization. For example, if the application runs on a resource-constrained device, the developer might want to optimize for memory consumption. The developer may also have multiple (partially contradictive) objectives such as minimum memory consumption, but maximum performance.

The list above shows that traditional feature diagrams are not enough for our purposes. To model cryptography components, we need attribute-based feature models that also support referencing. Such requirements bring us closer to ontology modeling [3, 28] and meta-models (e.g., MOF [2]). However, previous work has already shown that feature models can be seen as views on ontologies (encoded as class diagrams) [21]. Clafer [14] bridges the gap between these different modeling notations by providing(class-based) meta-modeling language with first-class support for feature modeling. It provides an interesting middle-ground between feature modeling that supports product configuration and domain-specific languages (DSLs) that provide greater flexibility in expressing things like attributes and references [44]. In general, Clafer provides a simple syntax and comes with extensive tool support [5]. For a DSL, one would need to build much of the tool support from scratch and train developers in using it. Furthermore, the expressiveness of Clafer is sufficient for all cryptography algorithms we have encountered so far.

Figure 4 shows an example of a simplified model for ciphers, digests, and key generators in Clafer. Any concept (feature, attribute, etc.) defined in Clafer is called a *clafer*. This example exploits Clafer's ability to have abstract definitions that can then be reused in subsequent definitions. Lines 1– 7 define an abstract clafer called `Algorithm` that has a name and a performance level, properties shared between all algorithms. The performance level varies between 1 and 4 according to the constraint on Line 4. Additionally, the algorithm can be marked as secure or insecure through the `status` clafer. `Status` is defined as an *exclusive-or group* where only one value can be selected at a time, a common feature modeling notation. The `status` clafer marks algorithms that have theoretical or practical attacks against them as insecure.

Lines 9–10 introduce another abstract clafer called `Digest` that extends `Algorithm`. In other words, a digest (i.e., a

```
1  abstract Algorithm
2      name -> string
3      performance -> integer //Levels 1 - 4 (4 fastest)
4        [performance >=1 && performance <= 4]
5      xor status
6        secure
7        insecure
8
9  abstract Digest : Algorithm
10     outputSize -> integer //in bits
11 }
12
13 abstract KeyDerivationAlgorithm : Algorithm
14
15 abstract Cipher : Algorithm
16     memory -> integer //Levels 1 - 4 (1 lowest)
17         [memory >=1 && memory <= 4]
18 }
19
20 abstract SymmetricCipher : Cipher
21     keySize -> integer
22
23 abstract SymmetricBlockCipher : SymmetricCipher
24     blockSize -> integer
25
26 abstract Task
27   name -> string
28 }
29
30 Ciphers
31   AES128 : SymmetricBlockCipher
32       [ name = "AES with 128bit key" ]
33       [ performance = 3 ]
34       [ secure ]
35       [ memory = 1 ]
36       [ keySize = 128 ]
37       [ blockSize = 128 ]
38
39   AES256 : SymmetricBlockCipher
40       [ name = "AES with 256bit key"]
41       [ performance = 3 ]
42       [ secure ]
43       [ memory = 1 ]
44       [ keySize = 256 ]
45       [ blockSize = 128 ]
46     DES: SymmetricBlockCipher
47       [ name = "DES"]
48       [ performance = 2 ]
49       [ memory = 2 ]
50       [ secure ]
51       [ keySize = 56 ]
52       [ blockSize = 64 ]
53
54 DigestAlgorithms
55   md5: Digest
56     [name = "MD5"]
57     [performance = 4]
58     [insecure]
59     [outputSize = 128]
60
61   sha_1: Digest
62     [name = "SHA-1"]
63     [performance = 4]
64     [insecure]
65     [outputSize = 160]
66
67   sha_256: Digest
68     [name = "SHA-256"]
69     [outputSize = 256 ]
70     [secure]
71     [performance = 2]
72
73 KeyDerivationAlgorithms
74   pbkdf : KeyDerivationAlgorithm
75     [name = "PBKDF"]
76     [performance = 2]
77     [secure]
78 }
79
80 PasswordBasedEncryption : Task
81   [name = "Encrypt data based on a password"]
82   kda -> KeyDerivationAlgorithm
83   digest -> Digest
84   cipher -> SymmetricBlockCipher
85   [cipher.keySize > 128]
```

Figure 4. Clafer model for sample hash functions ciphers and key derivation algorithms. Model also includes a password-based encryption task

hash function) is a type of algorithm. It includes all properties from `Algorithm`, but also adds more specific properties such as `outputSize`. The `outputSize` property determines the fixed length of the output produced by this hash function, measured in bits. Generally, the higher the number, the harder it is to conduct a brute force attack against the hash function. Similarly, Line 13 introduces an abstract Clafer called `KeyDerivationAlgorithm`. In our example, it does not have any additional properties. Lines 15–24 define the different types of ciphers and the additional properties each type might add to `Algorithm`. Lines 26–27 define an abstract clafer called `Task` that we use to represent a cryptography-related task (e.g., secure password storage, email transmission, data encryption, etc.).

Lines 30–71 define several instances of `Cipher` and `Digest`. These are the actual available algorithms. Each of these concrete algorithms are just grouped for convenience. A concrete instance assigns values to the properties defined in the abstract clafer. For example, `md5` sets the `performance` property to 4 and the `outputSize` to 128, leaving no further variability for these properties. Similarly, Lines 73–77 show one instance of a key-derivation algorithm. Note here that, as opposed to regular feature modeling, we have to create actual instances and not just rely on generating all possible instances from `Digest` or `Cipher`, simply because some combinations do not exist. For example, even though a cipher with a key of 100 bits is a valid instance, there is no corresponding algorithm that supports that.

A task that developers can later choose through the configurator is represented by a concrete clafer derived from the abstract clafer *Task*. Line 80 defines a password-based encryption task. Such a task needs a key derivation algorithm (Line 82), a digest to use with the key derivation algorithm (Line 83), and a cipher to perform the actual encryption (Line 84). At the moment, we are using basic tasks such as *store a password*, *encrypt data based on a password*, etc. We identified these tasks through a preliminary evaluation of the top 100 Java cryptography-related questions (sorted

```
1 String CONS =
2   "<javax.crypto.spec.PBEKEYSpec: void
        <init>(char[],byte[],int,int)>";
3
4 return UsageProtocolAPI
5   .atCallTo(CONS)
6   .ifParameter(2, p -> p < 500)
7   .reportError("Not enough iterations");
```

Figure 5. TSJ4 Specification for Checking PBEKeySpec Iteration Count

by view count and score) asked on StackOverflow. We are currently preparing a questionnaire to extract more information on the concrete problems faced by these developers and the causes of requesting help from the community. In parallel, we are currently conducting an empirical study on the uses of cryptography in real-world application code to find more common tasks that OpenCCE should support.

We have already modeled several ciphers, digests, and key derivation functions from the cryptography domain. These include AES, DES, MD5, SH1, SHA256, and PBKDF. We are currently modeling the common JCE algorithms that all providers should support. However, we are constantly extending our models to encompass more algorithms and libraries. To make sure that our Clafer model is correct, we are spending a considerable amount of effort on domain analysis to gather all requirements. An ongoing collaboration with cryptographic expert researchers ensures the correctness of the resulting definitions. While we are initially modeling the existing algorithms ourselves after consulting with domain experts, developers of cryptographic components can themselves add new components to OpenCCE in the future. Such components can then be offered to application developers as solutions to the existing tasks. For instance, if a cryptography expert provides a new encryption algorithm that uses specific hardware, this algorithm will now appear in the list of solutions offered to the developer as long as it satisfies the given constraints.

3.2 Usage Protocols

A usage protocol specifies constraints on the APIs themselves. For example, it can specify that a salt must be created using `javax.crypto.SecureRandom` (instead of an insecure mechanism such as `java.util.Random`) or that `ECB` mode should not be used with `AES`. It can also enforce methods to be called in a certain order. For example, a random salt must be generated before a password-based hash is created.

We identify the following requirements for the usage protocol notation.

1. *Support variation in API-usage protocols.* The public key of an asymmetric cipher may be based on constant key material. On the other hand, a symmetric cipher must be configured with a secret key that cannot be guessed, and therefore must be based on random values or values only known to trusted entities. The usage protocol must thus allow for differences between different types of ciphers.
2. *Support specification inheritance* Some APIs may share parts of their specification: both the *Data Encryption Standard* (DES) and AES are symmetric ciphers and thus share the requirement of a non-guessable secret key based on random values. Ideally, one does not need to duplicate such basic specifications, but can share them with the means of a hierarchical model in which concrete specifications can inherit properties from abstract ones.
3. *Human-readable and machine-processable.* The usage protocol language should be simple enough such that developers of new cryptographic algorithms can write the specifications themselves. Such usability also includes the ability to integrate the language into development environments with syntax highlighting and code completion. At the same time, it should be machine-processable such that static analyses can be automatically derived from it.

```
1 String CONS =
2   "<javax.crypto.Cipher: javax.crypto.Cipher
        getInstance(java.lang.String)>";
3
4 return UsageProtocolAPI
5   .atCallTo(CONS)
6   .ifParameter(0, p -> !p.contains("/") ||
        p.contains("/ECB"))
7   .reportError("Insecure block cipher mode");
```

Figure 6. TSJ4 Specification for Checking Block Cipher Mode

To cater for these requirements, we use TS4J [16] to encode the usage protocols. TS4J uses the idea of *fluent interfaces* to build a domain specific language (DSL), in Java, with an underlying typestate analysis. Fluent interfaces are also referred to as *internal DSLs* since they are implemented purely through careful engineering of the host language APIs. They do not require any syntactic or semantic extensions to their respective host language. This allows full re-use of the host language development tools and compilers. For example, TS4J exploits the Java compiler type checker to enforce the well-formedness of the usage protocol specification.

Figure 5 shows a TS4J definition of the usage protocol restriction *PBEKeySpec must be used with at least 500 iterations*. This tells the TSJ4 engine that it should conduct a check at every call site of the Cipher class constructor. If the value passed in as the third parameter (zero-based indexing) evaluates the given lambda predicate to *true*, it shall report an error to the developer. Note that this definition is declarative: It does not specify how the static analysis shall be conducted or how the value of the third parameter shall be obtained. It only defines the cases in which the analysis shall report an error to the developer.

Figure 6 shows how the same principle can be applied to prevent the insecure ECB block cipher mode from being used. Again, a parameter check is applied to every call site of a certain method, this time only with a more complex predicate. We have extended TS4J to support such parameter checks as they are not supported in its original implementation. We have compiled a list of similar needed extensions (e.g., predicates on field values) to TS4J that we are currently in the process of implementing.

While a Java developer can easily understand the TS4J notation, cryptographic providers are not necessarily expert Java developers. Since our goal is to allow cryptographic providers to add new primitives and algorithms to OpenCCE, we plan to develop a more higher-level domain specific language they can use to specify such usage protocols. Such a language can then translate to TS4J.

3.3 Configurator

During configuration, the developer selects one of the tasks offered by Clafer model. The configurator uses the Clafer instance generator (ClaferIG)[6] to find model instances that can be used to accomplish the selected task. Instances are possible solutions to the model that respect all constraints. Essentially, the instance generator tries to assign a concrete value to any non-abstract clafer in the model.

Referring back to the Clafer model example shown in Figure 4, let us ignore the last line of the example (Line 85) for a moment and look at the instances that ClaferIG would generate. In this case, ClaferIG tries to assign concrete values to `kda`, `digest`, and `cipher` or, in other words, tell us what are the possible algorithms the developer can use to do the password-based encryption. ClaferIG generates 9 instances that combine the various available key derivation, digest, and cipher algorithms for the given task.

Through the configurator, the developer can also specify additional requirements. Assume that the developer had a requirement that the keys to use for encryption must be greater than 128 bits in length. After eliciting this requirement from the developer, we can add the constraint shown on Line 85 to further restrict the model. In this case, only 3 instances would be generated because instances containing `AES_128` or `DES` would be removed since their key length does not satisfy the specified constraint. In addition to instance generation, Clafer also supports multi-objective optimization [38]. This allows the user to, for instance, specify that they want instances that maximize both performance and security.

In some cases, there might not be a valid combination of features that satisfy the user requirements. For example, if the user specifies that they want a cipher key size greater than 256 bits, no valid instances can be generated for the password-based encryption task shown in Figure 4 since no such cipher exists in the model. OpenCCE is faced with the challenge to detect such inconsistencies and report them to the developer in an understandable terminology.

[6] https://github.com/gsdlab/claferIG

```
1  public class KeyDeriv {
2    private String algorithm = "";
3
4    public SecretKey getKey(String pwd) throws ... {
5      SecureRandom r = new SecureRandom();
6      byte[] salt = new byte[32];
7      r.nextBytes(salt);
8
9      PBEKeySpec spec = new
           PBEKeySpec(pwd.toCharArray(), salt, 1000,
           128);
10     SecretKeyFactory skf =
           SecretKeyFactory.getInstance(algorithm);
11     return skf.generateSecret(spec);
12   }
13 }
```

Figure 7. Generic key derivation code

```
1 public class KeyDeriv {
2   private String algorithm = "PBKDF2WithHmacSHA1";
3
4   public SecretKey getKey(String pwd) throws ... {
5     return original(pwd);
6   }
7 }
```

Figure 8. Refining the generic key derivation code in Figure 7 using FeatureHouse to use the PBKDF2 algorithm

In the background, ClaferIG uses a SAT solver to generate the instances that satisfy the given constraints[7]. We are currently developing the configurator as an Eclipse plugin and exploring the tradeoffs between a filtering approach and a step-wise approach.

The filtering approach is currently used in the online Clafer configurator[8] where all valid instances of a given model are generated. Such instances are then filtered based on the constraints enforced by the user selection. The downside to such a solution is that if the configuration space is large, an extremely large number of instances would be generated, which is time-consuming.

In a step-wise approach (which is our current design approach), we present only valid selections to the user at each step during the configuration process. The idea is that only correct value ranges will be presented to the user (e.g., values 1 to 4 for performance levels) and that only algorithms satisfying the current set of constraints are presented in each subsequent configuration step. This is as opposed to accumulating all user requirements and trying to satisfy them at the end of the configuration process. However, the challenge here is to minimize the number of intermediate calls to the SAT solver.

[7] Clafer relies on Alloy [31] or Choco [1] as backend solvers.

[8] `http://www.clafer.org/p/software.html`

```
1 public class KeyDeriv {
2   private String algorithm = "PBKDF2WithHmacSHA1";
3
4   private SecretKey
        getKey__wrappee__KeyDerivation(String pwd)
        throws ... {
5     SecureRandom r = new SecureRandom();
6     byte[] salt = new byte[32];
7     r.nextBytes(salt);
8
9     PBEKeySpec spec = new
          PBEKeySpec(pwd.toCharArray(), salt, 1000,
          128);
10    SecretKeyFactory skf =
          SecretKeyFactory.getInstance(algorithm);
11    return skf.generateSecret(spec);
12  }
13
14  public SecretKey getKey(String pwd) throws ... {
15    return getKey__wrappee__KeyDerivation(pwd);
16  }
17 }
```

Figure 9. Resulting composition of Figure 7 and Figure 8

3.4 Composer

Now that the right features (i.e., algorithms) needed to perform the task have been selected by the configurator, we need to compose the corresponding artifacts to create a solution. Recall that a feature maps to two types of artifacts: code files and usage protocols. Even though Clafer provides us with the necessary modeling support, it is not a feature-oriented framework and thus, provides no support for mapping or composition of artifacts. We use FeatureIDE [43], an open-source development framework supporting all phases of feature-oriented development [7], to provide such a support. However, FeatureIDE uses its own underlying simple, feature-modeling notation that does not fully support attributed feature models[9]. On the other hand, FeatureIDE provides the ability to map features to any artifact type (both code and non-code) and provides several composition engines (e.g., AHEAD [11] and FeatureHouse [8]). To make the best out of both worlds, we are currently working on integrating Clafer into FeatureIDE.

To compose different artifacts, we use FeatureHouse, an open-source framework for software-artifact composition by means of superimposition [8]. We first discuss the composition of the Java code followed by that of the usage protocols.

Java Code. FeatureIDE already supports mapping features to Java files and FeatureHouse already supports composing Java files. We discuss how we use FeatureHouse for our password-based encryption example. To avoid cluttering the code with details irrelevant to this example, we only show parts of the example that illustrate how composition works. Figure 7 shows code for a generic key derivation algorithm where the `algorithm` field that is passed as a parameter to the `SecretFactory` is a variability point determining which algorithm will be used. Figure 8 shows the code for a specific key derivation function, PBKDF2, that refines the code in Figure 7 by giving a concrete value to `algorithm`. If the user selections result in using PBKDF2, then the code from Figure 7 and Figure 8 will be composed using superimposition, resulting in the code shown in Figure 9.

The example above illustrates a couple of limitations that we will need to overcome. First, the composed code is still in the form of a whole class. This means that this generated Java class would be added to the user code, but the user would still need to call the `getKey(..)` method, in this example, from within her code. Even if this call is synthesized automatically, it still clutters the code structure through unnecessary extra classes. Ideally, we want to generate all the needed code (in the form of a code snippet for example) directly into the application at the developer's current cursor position. One option is to have features correspond to code snippets instead of whole Java classes and that we change the level of composition granularity in FeatureHouse to be at the statement level. FeatureHouse currently relies on structure names (e.g., field or method names) for the superimposition. Working at the statement level requires coming up with some labeling scheme that is difficult to determine in advance.

The second limitation is that there are actually additional "variability points" in the code shown. For example, the number of iterations and output size for the generated key can be selected by the user. If we place such options in the feature model, then it means that we have to map the user selections into the code. In other words, the code written for each feature should be parameterized such that some values are read from the feature model and can be changed at composition time.

We have currently implemented some preliminary examples using FeatureIDE, but are still investigating the best way to solve the two problems above.

Usage Protocols. Composition in FeatureHouse is not restricted to code files. In fact, FeatureHouse is designed to be language-independent and adapts many ideas from AHEAD [11] that provides algebraic transformations for composition that apply to any type of artifact. To add support for a new language in FeatureHouse, a grammar for the language needs to be provided as well as the rules to use for composition. Since TS4J uses Java with only some minor extensions to the language, providing such a grammar is straightforward. However, the composition of rules poses some challenges.

A simple approach would be to just append all the rules corresponding to all the feature selections. This is based on the reasoning that even if key derivation is used with encryption, for example, the individual rules of each still applies. However, it might be the case that a set of extra rules must be added to account for any feature interactions. An inter-

[9] There are recent early-stage prototypes providing such support, but none are fully developed yet [30, 35]

esting point, though, is that the usage rules themselves can be used to account for feature interactions between the code files. Since the usage protocols enforce certain ordering constraints on method calls, they can be used as feature interactions resolutions when two features have to be composed in a certain order (e.g., the key generation must happen before the encryption in the code). We are still in early stages of using TS4J in our context, and have, therefore, still not attempted designing the appropriate composition rules for it within FeatureHouse.

3.5 Analysis Engine

After the usage protocols of the features have been composed, the respective static analyses must be generated and executed. Since rules written in TS4J directly compile into static analyses, this step is a matter of simply executing the composed rule code. No further transformations and no specialized execution environment is required. TS4J is already implemented as an Eclipse plugin that can highlight the location of the errors and show the error messages to the developer. Developers already know this style of error reporting from normal Java syntax errors and are comfortable using it. Together with proper error messages and examples of valid crypto API usage, this presentation bridges the gap between cryptographic problems or usage protocol violations on one side and the developer's code on the other side.

The errors presented to the user can be related to incorrect parameters or incorrect sequences of calls. Accepting an SSL certificate and using its public key for signature verification without previously checking the certificate expiry date is one example of the latter. Interestingly, a misuse of exactly this nature has lead to a vulnerability that allowed remote code execution in the auto-updating mechanism of the official German ID card app provided by the federal government [29]. The possibility to enforce correct usage sequences makes the analysis more powerful than existing approaches such as FindBugs [9] that only checks for misuse patterns, effectively missing not yet discovered API misuses. A TS4J specification can, for instance, enforce a typestate property on the SSL implementation so that it is only allowed to enter the signature check method if the key check method has been called before and has not returned an error.

In general, the static analysis based on the usage protocols must be executed as the code evolves. We plan to design an incremental analysis based on TS4J that would run after every incremental compilation performed by Eclipse, giving immediate feedback to the developer. This is in line with Eclipse's normal incremental compiler and its Java syntax error reporting.

4. Evaluation

Once we have addressed all the remaining open challenges discussed above, we plan to evaluate OpenCCE on three fronts, as follows.

We will evaluate the ease of use of OpenCCE through a controlled experiment where two groups of developers would be given the same tasks, but only one of them would use OpenCCE during development. To evaluate usability, we will observe and record their usage and actions during development as well as interview them about their experience. To evaluate whether OPENCCE improves secure integration, we will also compare the number and severity of the vulnerabilities present in the code developed by both user groups. We will use a cryptography expert to evaluate such vulnerabilities. Vulnerabilities arising both from domain misconceptions (picking the wrong components or combining them incorrectly) and from implementation mistakes will be analyzed. Ideally, the average severity and number of vulnerabilities in the applications developed with OPENCCE will be much lower than in the control control group who did not use OPENCCE.

We will also evaluate our generated code and analysis. Evaluating the generated code can be two-fold as well. First, we can get feedback from cryptographic experts on the quality of the generated code. Second, we can also run existing analyses proposed in the literature on it to ensure it does not contain vulnerabilities. Since OPENCCE will also provide own analyses to check for the correct integration of the cryptographic components, we will also run these analyses on the generated code. This of course means that our analysis itself must be correct.

We will evaluate our analysis on a prepared test set which contains valid and invalid code compared to the specified usage patterns. Additionally, we can run our analysis on code repositories such as Android apps in the Google Play store or Java code using cryptography on GitHub. (Note that our analysis operates on the bytecode, allowing us to also find misuses in closed-source applications.) That way, we can also study the results of our analysis in less controlled settings. In the long term, the option to publish OpenCCE as an official Eclipse project can also provide rather large-scale evaluation based on implicit and explicit user feedback. Appropriate discussions with the Eclipse foundation are already underway.

5. Related Work

In this section, we explain how our proposed ideas relate to existing and ongoing work in the domains of Software Engineering and Program Analysis.

5.1 Misuse of Cryptographic Components

Our focus is on benign misuses of cryptography rather than intentional malicious code. Many studies have analyzed the usability of cryptography. Clark and Goodspeed [19] and Whitten and Tygar [45] explore misuses of cryptography components from the end-user's point of view. Their focus is on the usability of cryptographic systems which are targeted at end-users, but require far more cryptogra-

phy knowledge than what can be expected from an ordinary user. Others analyze misuses of cryptography APIs by application developers. For example, Lazar et al. [33] show that over 83% of the vulnerabilities they analyzed from the CVE database were due to misuses of cryptography libraries while only 17% were caused by implementation bugs in the cryptography libraries themselves.

Similarly, an earlier study by Georgiev et al. [27] showed that many applications misuse SSL certificate validation leading to sensitive information (e.g., credit card data) being leaked. They found that even major web applications such as Amazon and PayPal were vulnerable to these mistakes. The authors include a list of recommendations for application developers (e.g., do not use the SSL library's defaults) and for SSL library developers (e.g., make the semantics of APIs more explicit). However, the authors also realize that these are short-term fixes and advocate presenting developers with higher-level abstractions and finding ways to verify that the application does not violate the semantics of the API – both of which are achieved by OPENCCE's design.

5.2 Bug-Finding Tools

There has been an ongoing effort to develop bug-finding tools that focus on misuses of cryptographic APIs. Such efforts often target mobile applications since smart phones and tablets hold more private user data than other devices. In particular, the Android platform has attracted the attention of many research efforts in the past due to its popularity and openness. Fahl et al. [25] present MALLODROID, a static analysis tool that finds misuses of the SSL API in Android applications. Egele et al. [24] developed CRYPTOLINT, a static analysis tool that checks Android applications for common cryptographic misuses: hard-coded seeds, using AES in the non-secure ECB mode, and using weak password-based encryption. Ball et. al. [10] use binary decision diagrams (BDDs) to statically detect misuse of Windows device driver APIs. They convert C programs to Boolean programs and then apply *counterexample-guided abstraction refinement* to automatically detect whether a program is correct with respect to an API usage rule or find a true error.

Flanagan et al. [26] have proposed Extended Static Checking for Java to check programs for compliance with high-level design decisions by means of code annotations and automated theorem proving which may also be suitable for enforcing certain API usage protocols. However, code annotations must not place too much of a burden on the application developer or implementer of a new cryptographic algorithm. Ideally, the specification is as abstract and short as possible.

Typestate analysis attempts to find API-protocol violations. Analysis can be either performed ahead of time, in a compiler's tye system [13, 42] or after the fact, using static analysis [15, 17]. We use the typestate analysis implemented in TS4J [16] in OPENCCE. One problem that many of those approaches face is the lack of available property specifications. Property-inference techniques can address this problem in some situations by inferring likely valid properties, for instance from program runs [39, 41].

5.3 Formal Software Verification

In addition to program analysis techniques, formal verification has also been used to detect vulnerabilities in cryptography and security-related protocols. Mitchell et al. [36] introduce Murφ, a general-purpose state enumeration tool that analyzes cryptographic and security-related protocols. Microsoft Crypto Verification Kit [12] is another verification-based tool that ensures the correct implementation of TLS 1.0.

Protocol Composition Logic (PCL) by Datta et. al. [23] is a logic for proving security properties of complex protocols. It allows properties of inidividual steps to be combined to prove properties of the overall system. Such approaches are relevant as OPENCCE must also combine several cryptographic components to create a secure solution based on the user's requirements while only having specifications of the individual components.

While formal verification offers guarantees on the soundness of the result, full soundness is often too strict for real-world programs [34] due to an increased number of false positives. Therefore, it is not suitable to use in OPENCCE.

5.4 Improving Crytographic API Documentation

Cairns and Steel [18] argue for the need to make using cryptographic APIs easier. However, they mainly focus on designing the APIs themselves to be more useful and intuitive to use. On the other hand, OPENCCE uses automated techniques to support currently available APIs by suggesting correct usage patterns.

6. Conclusion

As cryptography becomes increasingly important to protect the vast amount of user data available in modern time, one needs to ensure that application developers are able to use it correctly. To address this need, we combine techniques and concepts from both software engineering and program analysis. Specifically, we argue that configuring and composing cryptographic components can be seen as creating a product from a software product line. To facilitate composition and check for correct integration, we encapsulate cryptography domain knowledge in a feature model supported by a program analysis component. This approach, which is at the core of the OPENCCE Eclipse plugin, is a practical solution to the misuse of cryptographic APIs. Non-expert developers should be able to secure sensitive data in their applications without intricate knowledge of cryptography.

We use our feature model to capture related domain knowledge and use it to guide the developer through high-level questions that are similar to how she thinks about the

problem. Secure code corresponding to the user's requirements is then generated automatically. Since the developer may change the generated code and how it is integrated with her application, we also apply static-analysis techniques to ensure the code complies with the usage specifications of each component. By alleviating the responsibility of secure integration of cryptographic software off application developers, misuse of APIs would be reduced leading to a more secure world for all end users.

Acknowledgment

We would like to thank Stefan Krüger for his help with FeatureHouse. This work is funded by the DFG as part of project E1 within the CRC 1119 CROSSING, and was further supported by the BMBF within EC SPRIDE and by the Hessian LOEWE excellence initiative within CASED.

References

[1] Choco solver. `http://choco-solver.org`.

[2] OMG: Meta Object Facility (MOF) Core Specification. Available at http://www.omg.org/spec/MOF/index.html.

[3] W3C. OWL web ontology language. Document available at http://www.w3.org/TR/owl-features/.

[4] R. J. Anderson. *Security engineering - a guide to building dependable distributed systems (2. ed.)*. Wiley, 2008.

[5] M. Antkiewicz, K. Bąk, A. Murashkin, R. Olaechea, J. H. J. Liang, and K. Czarnecki. Clafer tools for product line engineering. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, SPLC '13 Workshops. 2013.

[6] S. Apel, D. Batory, C. Kästner, and G. Saake. *Software Product Lines*. Springer Berlin Heidelberg, 2013.

[7] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[8] S. Apel, C. Kastner, and C. Lengauer. Featurehouse: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09. 2009.

[9] N. Ayewah, D. Hovemeyer, J. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *Software, IEEE*, 25(5):22–29, Sept 2008.

[10] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In Y. Berbers and W. Zwaenepoel, editors, *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006*. 2006.

[11] D. Batory. A tutorial on feature oriented programming and the ahead tool suite. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pp 3–35. Springer Berlin Heidelberg, 2006.

[12] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu. Cryptographically verified implementations for TLS. In *Proc. of the Conference on Computer and Communications Security (CCS)*. 2008.

[13] K. Bierhoff and J. Aldrich. *Modular typestate checking of aliased objects*, volume 42. ACM, 2007.

[14] K. Bąk, K. Czarnecki, and A. Wąsowski. Feature and metamodels in clafer: Mixed, specialized, and coupled. In *Proc. of the International Conference on Software Language Engineering (SLE)*, 2010.

[15] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *ICSE '10: International Conference on Software Engineering*. May 2010.

[16] E. Bodden. Ts4j: a fluent interface for defining and computing typestate analyses. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. ACM, 2014.

[17] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *Proceedings of the 21st European conference on Object-Oriented Programming*. Springer-Verlag, 2007.

[18] K. Cairns and G. Steel. Developer-resistant cryptography. In *A W3C/IAB workshop on Strengthening the Internet Against Pervasive Monitoring (STRINT)*, 2014.

[19] S. Clark, T. Goodspeed, P. Metzger, Z. Wasserman, K. Xu, and M. Blaze. Why (special agent) johnny (still) can't encrypt: A security analysis of the APCO project 25 two-way radio system. In *Proc. of the USENIX Security Symposium*, 2011.

[20] P. Clements and L. Northrop. *Software product lines: practices and patterns*, volume 59. Addison-Wesley Reading, 2002.

[21] K. Czarnecki, C. Hwan, P. Kim, and K. Kalleberg. Feature models are views on ontologies. In *International Software Product Line Conference*, SPLC '06, 2006.

[22] J. Daemen and V. Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer, 2002.

[23] A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol composition logic (PCL). *Electr. Notes Theor. Comput. Sci.*, 172:311–358, 2007.

[24] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in Android applications. In *Proc. of the Conference on Computer and Communications Security (CCS)*, 2013.

[25] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben. Why Eve and Mallory love Android: An analysis of android SSL (in)security. In *Proc. of the Conference on Computer and Communications Security (CCS)*, 2012.

[26] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*. 2002.

[27] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Proc. of the Conference on Computer and Communications Security (CCS)*, 2012.

[28] T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, 43(5):907–928, 1995.

[29] Heise. Neuer Personalausweis: AusweisApp mit Lücken. Heise Website. `http://www.heise.de/newsticker/meldung/Neuer-Personalausweis-AusweisApp-mit-Luecken-2-Update-1133376.html`.

[30] S. Henneberg. Next-generation feature models with pseudo-boolean sat solvers, 2011.

[31] D. Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[32] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Tech. rep., CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[33] D. Lazar, H. Chen, X. Wang, and N. Zeldovich. Why does cryptographic software fail? A case study and open problems. In *Proc. of the ACM Asia-Pacific Workshop on Systems (APSys)*, 2014.

[34] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[35] L. Machado, J. Pereira, L. Garcia, and E. Figueiredo. Splconfig: Product configuration in software product line. In *Brazilian Congress on Software (CBSoft), Tools Session*, 2014.

[36] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using mur-phi. In *Proc. of the IEEE Symposium on Security and Privacy*, 1997.

[37] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Advances in Cryptology-CRYPTO 2003*, pp 617–630. Springer, 2003.

[38] R. Olaechea, S. Stewart, K. Czarnecki, and D. Rayside. Modelling and multi-objective optimization of quality attributes in variability-rich software. In *Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages*, NFPinDSML '12. 2012.

[39] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking api protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12. 2012.

[40] M. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):27–34, Nov 2009.

[41] M. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering (TSE)*, 39(5):613–637, 2013.

[42] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering (TSE)*, 12(1):157–171, 1986.

[43] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

[44] M. Voelter and E. Visser. Product line engineering using domain-specific languages. In *Software Product Line Conference (SPLC), 2011 15th International*, Aug 2011.

[45] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proc. of the USENIX Security Symposium*, 1999.