UMLsecRT: Reactive Security Monitoring of Java Applications with Round-Trip Engineering

Sven Peldszus, Jens Bürger, Jan Jürjens

Abstract—Today's software systems tend to be long-living and often process security-critical data, so keeping up with ever-changing security measures, attacks, and mitigations is critical to maintaining their security. While it has become common practice to consider security aspects during the design of a system, OWASP still identifies insecure design as one of the top 10 threats to security. Furthermore, even if the planned design is secure, verifying that the planned security assumptions hold at run-time and investigating any violations that may have occurred is cumbersome. In particular, the configuration of run-time monitors such as the Java Security Manager, which could enforce design-time security assumptions, is non-trivial and therefore used in practice rarely. To address these challenges, we present UMLsecRT for automatically supporting model-based security engineering with run-time monitoring of design-time security specifications and round-trip engineering for propagating run-time observations to the design level. Following the established security-by-design approach UMLsec, security experts annotate system models with security properties that UMLsecRT automatically synchronizes with corresponding source code annotations for the automatic configuration of UMLsecRT's run-time monitor. To this end, UMLsecRT monitors these security properties at run-time without additional effort to specify monitoring policies. Developers can define mitigations for attacks detected at run-time in advance by adjusting the automatically synchronized annotations at implementation time. Triggered by a security violation, UMLsecRT can adapt the design-time models based on run-time findings to facilitate the investigation of security violations. We evaluated UMLsecRT concerning its effectiveness and applicability to security violations extracted from real-world attacks and the DaCapo benchmark, conducted user studies on the usability of the adapted models and the feasibility of UMLsecRT in practice, especially concerning countermeasures, and investigated the scalability of UMLsecRT. To study the applicability of the whole development process, we applied UMLsecRT in two case studies to the Eclipse Secure Storage and the electronic health record system iTrust.


1 INTRODUCTION

In today’s software, security is one of the most important quality aspects [1], [2], [3], [4], [5]. Already in 2021, OWASP identified the insecure design of systems as one of the biggest security threats [6]. While there are several approaches to support security at design time, such as using design-time models [7], [8], [9], but also statically during implementation [10], [11] and at runtime [12], [13], [14], few cover the coupling of these phases. Since practice has shown that it is likely that there will always be some kind of exploit, it is insufficient to consider the different approaches in isolation. While implementing well-designed measures to enforce the security design of a system, we should always assume that the system is vulnerable and actively monitor for violations of the fundamental security properties of the security design, such as the secrecy and integrity of data and services. Whenever a security violation is detected, it must be mitigated, and developers must be provided with information to investigate and improve the system.

During software development, different representations of a system are created, e.g., to plan the security of a system before its implementation [8], [15], [16], [17]. When changes are made, all of these representations must be updated to reflect the changes, and all security assumptions must be re-verified [18]. An automation of this process is commonly referred to as round-trip engineering [19], [20], [21]. Current round-trip engineering approaches focus only on maintaining architecture models during the normal development process [20], [21], [22], but neglect security. They focus on changes that occur as part of the usual development process, but for security we also need to consider unexpected changes, such as deploying the system with a different library version, or changing system behavior due to an attack. To the best of our knowledge, no existing approach to secure software engineering supports round-trip engineering, even though it is important for several security-related reasons.

While it is desirable to discover vulnerabilities as early as possible [23] and support for automating vulnerability detection and reaction should be provided already at design time, many security vulnerabilities are difficult to detect in design artifacts or source code [24], [25], [26]. This applies especially to vulnerabilities based on language features that are not entirely statically analyzable, such as Java reflection [27], [28]. Here we need the ability to enforce design-time security decisions at runtime. While several security monitors have been developed [29], [30], [31], [32], and the Java Security Manager is even part of the standard Java library [33], they have never been widely used in practice.
Reasons for the lack of usage include a huge effort to configure security monitors such as the Java Security Manager and a too coarse-grained granularity of supported security rules, which ultimately led to its deprecation in 2021 [34].

Furthermore, divergences between the design-time security assumptions and the implementation likely appear [35]. In this regard, it is essential to keep all artifacts synchronized to ensure the security of a system, e.g., when restructuring the system’s design [18]. However, even in a statically secure system, security violations can occur at runtime. To ease the investigation of violations, the design-time models should be automatically adapted to contain runtime observations and their relationships to the planned security design.

In this work, we propose UMLsecRT, which allows developers to specify fundamental security properties concerning secrecy and integrity in design-time models using established security engineering methods, or alternatively directly in the source code, and to monitor their compliance at runtime. We consider attackers that are able to inject custom Java code or manipulate code execution in a way that leads to a violation of the specified security properties. Violations and findings at runtime, like possible attack sequences, can be synchronized back to the model by adapting it. If a security property is violated, e.g., by a vulnerability introduced during an update or an attack, the system operator is notified and the system is brought into a safe state. What is considered a safe state in which situation is also handled within the security properties.

To ease applicability and avoid additional effort, we rely on security properties as specified in UMLsec [8]. UMLsec provides a well-known UML extension for modeling and verifying secure software systems, and has been applied in industry [36], [37], [38], [39], [40], [41]. Since early security engineering is essential, we assume that security annotations similar to UMLsec have already been applied to design-time models as part of the development process. Such security modeling can be supported by natural language processing [2], [42].

Figure 1 visualizes our approach for enforcing design-time security specifications at runtime and adapting system models based on runtime information. We explain the usage of the approach step by step:

1) For round-trip engineering, UMLsecRT requires a UML model that is consistent with the source code (e.g., a class diagram for the UMLsec security annotations used for demonstration in this work). If such a model is not available, UMLsecRT supports reverse engineering it from Java source code. If only security monitoring is required (thus neglecting round-trip engineering), developers can work only on the source code, to which the security annotations are then applied directly.

2) A developer annotates, assisted by tool support [43], the UML model, the source code, or both with security properties derived from the project’s requirements. Thereby, as part of the standard security-by-design procedures, the UMLsec annotations can be directly used for static security checks using the tooling of UMLsec.

3) UMLsecRT automatically synchronizes annotations added to the model with the source code and vice versa. If developers annotate only the UML model, UMLsecRT can automatically generate all source code annotations from the model. Optionally, developers can specify reactions to security violations by detailing the source code annotations.

4) The annotated Java source code is executed and the execution is monitored for security violations wrt. the security annotations added in the earlier steps.

5) UMLsecRT adapts the models based on the security-relevant data collected at runtime, e.g. by adding sequence diagrams describing detected violations.

We conclude the introduction by highlighting the contributions of this paper:

i) We introduce runtime security monitoring of Java applications concerning design-time security specifications, avoiding the huge overhead for manually configuring a security monitor (Section 3.2).

ii) We support synchronization of security annotations between design models and source code, reducing the burden of manually annotating an entire code base and making it easier to implement design-time security specifications (Section 3.1.1).

iii) We provide support for active countermeasures to mitigate attacks at runtime (Section 3.3).

iv) We support round-trip engineering by adapting the model with automatically generating sequence diagrams of attacks and call relationships missing in the model but monitored at runtime (Section 4).

v) We provide an evaluation of our approach based on security violations extracted from real-world attacks, on the DaCapo benchmark [44] and the iTrust electronics health records system [45], open-source applications of varying size, as well as an user study (Section 6).

The work is organized as follows: First, we introduce the necessary background in Section 2, consisting of Eclipse Secure Storage [46], our real-world running example, the UMLsec secure dependency property, used to exemplify our security monitoring, and an example of a security violation with respect to UMLsec. Section 3 covers how to annotate Java source code with security annotations corresponding to those of UMLsec, how to monitor for violations at runtime, and how to take countermeasures when security violations occur. In Section 4, we present how the design-time model can be adapted based on runtime observations. We present our tool support for UMLsecRT in Section 5 and an evaluation of the effectiveness, applicability, usefulness, and scalability of UMLsecRT in Section 6. We present the practical application of UMLsecRT to two real-world case studies, the Eclipse Secure Storage [46] and the electronics health records system iTrust [47], in Section 7. We discuss assumptions and implications for working with UMLsecRT in Section 8 and related work in Section 9. Finally, in Section 10, we conclude and give an outlook on future work.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
and assets in threat modeling. In Secure Dependency, the trust boundaries are expressed as a contract between calling and called objects. It ensures that call dependencies respect the security requirements of the data communicated along them. This results in a detailed specification of trust boundaries that are specific to individual security requirements and express which accessing objects are to be considered within the trust boundary. Security engineers either plan for an object to implement a measure that ensures the security requirement is not violated by accesses through that object, or the trust boundary must be extended.

We assume objects to consist of members (methods and fields) as well as lists of member signatures with the security property secrecy or integrity. If a member signature is not unique, it has to be qualified with the fully qualified names of the defining type. The following definition, adapted from the definition of «secure dependency» in [8] addresses secrecy:

A (sub-)system fulfills Secure Dependency iff for all call-dependencies d from a class C to a class D for all member signatures s ∈ D.members holds that s ∈ C.secrecy if and only if s ∈ D.secrecy. Meaning that either both, the source and target, have to guarantee a security property for the accessed member or none of the both.

The integrity case is entirely analogous to the secrecy case introduced above. In both cases, security properties are specified using the «critical» stereotype of UMLsec, which has two lists of member signatures secrecy and integrity, containing the members with the corresponding properties.

As in Figure 2, the class SecurePreferences is annotated «critical» and the secrecy list holds the signature get(String, String, SecurePreferencesContainer):String (visualized in the comment linked to the class), all classes with a dependency to this class that is stereotyped with «call» have to respect this secrecy security level. This is represented by a «secrecy» stereotype on the dependency and «critical» containing this signature, on the class SecurePreferencesWrapper.

As described above, UMLsec secure dependency allows us to detail trust boundaries and plan their implementation, focusing in particular on where security measures need to be implemented. When implementing a system specified by a UML model, the dependencies stereotyped with «call» are usually implemented as method calls and field accesses. Even if a model contains no violations, it must be guaranteed at runtime that the security properties specified at design time are preserved. Furthermore, the detection of all dependencies that may occur at runtime is statically undecidable, e.g., due to Java reflection [24], [28], violations caused by a shared library, or injected malicious code. In

Fig. 2: Eclipse Secure Storage annotated with UMLsec Secure Dependency Stereotypes.

Listing 1: Source code of a malicious application that reads Git passwords from the Eclipse Secure Storage.

```
public void readPassword(ISecurePreferences s) {
    ISecurePreferences git = s.node("git/gitlab");
    Method m = git.getClass().getMethod("get", ...);
    m.setAccessible(true);
    sendPassword((String) m.invoke(git));
}
```

Eclipse, for example, any installed plugin can read any password from the password store.

At runtime, we need to ensure that there are no unintentional accesses that cross a trust boundary, technically speaking, accesses to objects classified using UMLsec that were not planned for at design time. We consider an attacker model with two threats and an attacker who has gained extensive capabilities in the system.

2.4 Example of a Security Violation

Listing 1 shows how a malicious plugin can exploit the secure storage API to read stored passwords. To do so, it needs to access the get method, which according to Figure 2 is on the secrecy security level. To avoid detection by static analysis, it uses the Java Reflection API to access the get method of the ISecurePreferences class. To accomplish this, in line 2 the malware navigates to the ISecurePreferences instance that holds the desired passwords, and then accesses them in lines 3 through 5. First, it gets a Method object, sets it to accessible, and finally gets the value of that method and passes it to a sendPassword method.

2.5 Synchronization between Models and Code

Synchronizing models and code when changes are made to either is an essential part of model-based engineering. In the context of continuous evolution, we must ensure that the models represent the system and we must maintain traceability. Many existing approaches use graph transformations that provide model synchronization to deal with the issues arising from evolution [55], [56]. Similarly, we use a triple graph grammar (TGG) [57], a rule-based transformation that supports synchronization of changes made to both the source and target models.

We build upon our previous work [18], in which we extended a set of prototypical TGG rules of Leblebici et al. [58] for transforming Java programs into UML class diagrams to not only be able to extract UML class diagrams from Java source code but also to synchronize changes in both
it is specified in a rule set how the two models correspond with each other. Thereby, preserving mappings (isomorphisms) that express the change \( L \rightarrow L' \) and \( R \rightarrow R' \). This concept can be applied for both, changes on the statement level that we abstract, which cannot be recovered due to the possible revocation of TGG rules, this can lead to information loss was one of the major adaptations we had to make to the Leblebici et al.’s rule set. When considering security annotations, we will have similar problems due to scalability issues, and the security properties will not be available at runtime as required in this work.

Before we introduce how we use TGGs, we briefly introduce their theoretical background. When a TGG transformation is applied between two models, a correspondence model is built between the two models that captures which elements have been translated to each other. This correspondence model is then used to synchronize changes in one model by reflecting them in the other model. A set of rules defines how the elements from the two models correspond.

Formally, as shown in Figure 3, we have a triple of models consisting of a left model \( L \), a right model \( R \), and a correspondence model \( C \) between \( L \) and \( R \). The correspondence model is connected with \( L \) and \( R \) via two structure-preserving mappings (isomorphisms) \( l \) and \( r \) that express how the two models correspond with each other. Thereby, it is specified in a rule set \( RS \) how to construct \( L \) and \( R \). Whenever \( L \) or \( R \) is changed, which is captured as a morphism \( m \) that expresses the change \( L \rightarrow L' \), based on the rule set \( RS \) it is possible to calculate the morphisms \( d \) and \( m' \) that allow adapting the other models correspondingly to \( C' \) and \( R' \). This concept can be applied for both, changes \( L \rightarrow L' \) as outlined but also in the other direction for a change \( R \rightarrow R' \). For a formal proof of triple graph grammars based on double pushout graph rewriting, please refer to the original publication of Andy Schürr [57].

As an example of TGG rules and to intuitively explain how exactly TGGs work, Figure 4 shows two explanatory TGG rules from a rule set for translating between the MoDisco Java metamodel and UML models. The rule in Figure 4a shows how to translate classes contained in packages and the rule in Figure 4b shows how to translate methods contained in Java types to operations in UML classifiers.

The left side of the figure shows the elements from the MoDisco source code model and the right side the elements from the UML model. In between is the correspondence model of the TGG. Each rule consists of model elements (nodes and references) that are translated by this rule or serve as context and are expected to be translated by other rules. Elements that are newly translated by this rule are annotated with a ++ and highlighted in green. In black and without annotations we show the context needed to apply the rule. This context must be translated by other TGG rules before this rule can be applied. There may also be conditions on element attributes that must be met for a rule to be applied or will hold after translation of newly translated nodes, e.g., that the two class nodes in Figure 4a will have the same name.

By matching one side of the rules to a model, e.g., a MoDisco model that we want to translate into a UML class diagram, we can select possible rules to translate specific patterns. Each rule translates only a small part of the two models. For example, the rule for translating Java methods to UML operations will be applied to any method contained in any Java type (AbstractTypeDeclaration). However, this rule can only be applied if the context of the rule, in this case the AbstractType, has been translated using another TGG rule. In this case, the rule in Figure 4a would be one of the rules that translates a subtype of the required AbstractTypeDeclaration. Additional rules can translate other subtypes or combinations of Class nodes and references. By iteratively applying these rules, we can translate one model into another model.

If one of the two sides is changed, TGGs allow to reflect the changes on the other side. There are two variations to consider. First, the change may involve newly added elements. These elements are translated by additional rule applications as described above. Second, deletions are handled by revoking the rule applications that translated these elements, resulting in the deletion of the element created by the rule application in the other model.

If you only consider TGG rules that perform one-to-one translation, as shown in the example, TGGs are straightforward. However, it is not necessary to translate every element contained in a model. In our rule set, we use this to provide an abstraction from the source code. However, due to the possible revocation of TGG rules, this can lead to the deletion of untracked elements, e.g., all the details of the statement level that we abstract, which cannot be recovered by additional rule applications. Avoiding rules that lead to information loss was one of the major adaptations we had to make to the Leblebici et al.’s rule set. When considering security annotations, we will have similar problems due to the fact that we only want to specify countermeasures at the implementation level.
3 Runtime Enforcement of UMLsec Secure Dependency

We propose to couple design-time security with runtime security by introducing a notation for specifying security properties in Java source code that can be monitored at runtime. In addition, we discuss reverse engineering and synchronization of UML models annotated with UMLsec stereotypes and Java source code annotated with security annotations. We also show how we implement countermeasures to mitigate security violations at runtime.

3.1 Specification of Security Properties

In UMLsecRT, we mainly work with UML models and Java source code as well as its byte code. In the following, we introduce annotations to support both types of artifacts.

3.1.1 Security Annotations

To annotate UML models, we use the existing UMLsec stereotypes [8], focusing on the security dependency property, as exemplified in Section 2. More specifically, this includes the stereotypes «secrecy», «critical», and «call». In Figure 2, we demonstrated UMLsec security annotations on an implementation-level model. However, initial security properties are usually specified earlier, such as on domain models. Using inheritance relationships between UML models of different granularity, these security properties can be propagated from very abstract models to models close to the implementation [49].

Java annotations provide a mechanism similar to UML profiles for annotating Java source code. Java annotations support three different retention levels, in addition to being available only in the source code or at compile time, they can also be available at runtime as required in UMLsecRT. Thus, we have defined a set of Java annotations to support typical security properties that are aligned with the set of annotations introduced in UMLsec, so that source code (especially fields and methods) can be annotated.

Table 1 gives an overview of the Java annotations we define and their relationship to the corresponding UMLsec stereotypes. The Java annotations @Critical, @Secrecy, and @Integrity are used semantically identical to their UMLsec counterparts. In UMLsec’s «critical», all information about security levels is provided within the values secrecy and integrity. Similarly, we have defined the secrecy and integrity parameters, which, like «critical», provide arrays of member signatures. Typically, methods and fields are annotated by specifying them as part of the corresponding values of «critical». To avoid errors due to typos, and to maintain clarity in larger classes, we also support annotating methods and fields directly with @Secrecy and @Integrity, respectively.

In Listing 2, we have applied the UMLsecRT annotations to the Java source code of the SecurePreferencesWrapper class.

<table>
<thead>
<tr>
<th>UMLsec stereotypes</th>
<th>UMLsecRT annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>stereotype</td>
<td>tagged values</td>
</tr>
<tr>
<td>«critical» secrecy</td>
<td>Class @Critical secrecy, integrity</td>
</tr>
<tr>
<td>«critical» secrecy</td>
<td>Member @Secrecy</td>
</tr>
<tr>
<td>«critical» integrity</td>
<td>Member @Integrity</td>
</tr>
</tbody>
</table>

Listing 2: Source code excerpt from the Eclipse Secure Storage with UMLsecRT security annotations.

```java
1. @Critical(secrecy={"get(String,String, SecurePreferencesContainer):String")
2. public class SecurePreferencesWrapper implements ISecurePreferences {
3.   private SecurePreferences node;
4.   @Secrecy
5.   public String get(String key, String def) {
6.     return node.get(key, def, container);
7.   }
8. }
```

Fig. 5: TGG Rule for translating @Critical-annotations in an implementation-level model to «critical»-stereotype in a UML model.

The value secrecy={get(String, String):String} of «critical» is represented by a @Secrecy annotation on the get method in line 5 of the example. In addition, the secrecy security property is specified for a member with the signature get(String, String, SecurePreferencesContainer):String in the @Critical annotation on line 1, which is called on line 7.

Using the two presented mechanisms developers can specify the same security properties on both UML models as well as Java source code.

3.1.2 Synchronization with UML Models

To synchronize the UMLsec annotations with the UMLsecRT annotations in the source code (see step 3 in Figure 1), in addition to the mapping between the annotations from Table 1, a mapping between UML elements and Java source code as well as a synchronization mechanism are required. To this end, we extended the rule set from our previous work [18] to support the synchronization of the proposed security annotations in Java source code. We successfully applied the extended rules to the example that generated the annotations shown in Listing 2.

Figure 5 shows a rule from our extension to our TGG-based synchronization of models and code that allows UMLsecRT’s security annotations to be properly supported. We explain the extension and this rule below. The rule is used to translate «critical» stereotypes into @Critical annotations in the source code, as needed in our application scenario of propagating security annotations from the design models to the implementation. The values of this annotation are translated using separate rules.

In the shown rule we assume as context that a UML Classifier has been translated into an AbstractTypeDeclaration, e.g., the UML class SecurePreferencesWrapper into a corresponding
Java class, and that an AnnotationTypeDeclaration with the name Critical has been translated. If we find this context and there is an untranslated <critical> stereotype, we translate it to an Annotation of type Critical on the corresponding Class, which means we add this security annotation to the source code. The rule can also be applied in the opposite direction, as needed for reverse engineering UML models.

In our extension, we had to solve two challenges. First, the aforementioned abstraction of countermeasure specifications in UML models. Since these countermeasures are captured in parameter nodes owned by the annotation nodes in the MoDisco model, and there are no additional references pointing to these parameters, the only change that would result in the loss of this information is a change in the baseClassifier reference. If this reference was changed, the application of the rule would be undone, resulting in the deletion of the annotation in the source code, including all of its parameters. The annotation itself would be recreated by a new rule application, but the parameters would be lost. Since editors like Papyrus that we use do not allow this change, we can keep this rule. A more robust option would be to split the rule into two rules, one that just translates the annotation nodes and one that links them to the classifiers.

The second challenge is how to represent annotations and stereotypes in the two models. The stereotypes are an extension of the UML metamodel and can be used as typed nodes in the TGG rules, while the Java annotation types are part of the implementation model itself. As shown in the rule in Figure 5, we can still create a mapping by using the typed node in the UML model and matching the expected name in the type declaration in the implementation-level model. Since the UML to source code TGG rule set we are building on already supports Java annotations for other purposes, this new rule interacts with the existing rules. We had to make sure that no two rules matching Java applications would match at the same time. To do this, we extended the existing rules with name checks to ensure that the name of the AnnotationTypeDeclaration is not Critical, Secrecy, or Integrity, which are cases that need to be translated by our newly added rules.

### 3.2 Verification at Runtime

After specifying and statically verifying security properties, e.g., using our previous work [18], [59], the next step is to execute the annotated source code and monitor the execution for security violations (step 4 in Figure 1). To ensure that we detect every security violation concerning «secure dependency», we have to check all method calls and field accesses for their compliance with the specified security properties. The security mechanisms built into Java, such as the Security Manager, are not sufficient to implement UMLsecRT. They cannot be configured fine-grained enough (only at jar file or classpath entry level) and are only executed when checkPermissions is explicitly called [33].

We need to take action whenever a method is entered or exited, or a field is accessed. According to «secure dependency», we must consider two cases: (1) the accessed member is missing an annotation, or (2) the accessing member is missing an annotation. These two cases can occur at the same time.

To enable such monitoring, we use bytecode instrumentation to inject security checks into the running application.

The security checks are injected at the beginning and end of each method. Listing 3 shows conceptually what code needs to be injected into methods and is explained below.

Although the JVM maintains call stacks for all threads [60], required information such as annotated security properties is not accessible from these stacks. For this reason, the UMLsecRT provides a global set of stacks for call traces, one stack per thread. The stack is retrieved when a method is entered (line 1 of Listing 3). Whenever a method is entered, the safe dependency conditions are checked in line 5. To accomplish this, whenever such a relevant event occurs, we must examine the call trace backwards and check both that the originating method is annotated as required and that the accessed member is annotated as requested by the originating method. In line 2, the security annotations of the originating member are read from the stack, and in lines 3-4, the annotations of the currently instrumented method are built by reading them from the bytecode and hardcoding them into the injected code. Additionally, the method is pushed to the stack (line 6). After all statements of the method have been executed, but before the return is finally initiated, the method is removed from the stack in line 8.

```java
1 RTStack stack = RTStackManager.getStack(curThread());
2 RTAnnotation originating = stack.peek();
3 String[] secrecy = ... // Signatures on the secrecy level
4 RTAnnotation accessed = new RTAnnotation("Signature of this method", secrecySet);
5 check(originating, accessed);
6 stack.push(accessed);
7 ... // Original method code
8 stack.pop();
```

Listing 3: Code for monitoring, injected before and after method code.

Since field accesses are statically analyzable [33], we check them whenever a class is loaded. Depending on the developer’s preference, we can throw security exceptions immediately or instrument the field access so that it is thrown when the access is executed. An exception to this are reflective field accesses. Here, we instrument the Java reflection library methods to perform the necessary checks.

In Figure 6, we demonstrate security monitoring for the execution of EGitSecureStore using Eclipse Secure Storage, showing a control flow graph excerpt on the left and the executed monitoring steps on the right. First, the get(URIish)
method of the EGitSecureStore class is called to get a password for a Git URL. This method accesses the preferences field and calls the node method on the field to get the node that holds the password for the requested URL. The called node method returns a SecurePreferencesWrapper instance by calling the wrapper method of a SecurePreferencesContainer. On the returned SecurePreferencesWrapper instance, get(String, String): String (the implementation is shown in Listing 2) is called to get the stored password.

If one of the validations on the right side of the figure fails, we provide several responses to mitigate the violation. We discuss these responses in the next section.

3.3 Countermeasures

If a violation of UMLsec’s secrecy or integrity security property is detected, we provide four different countermeasures to investigate the violation and prevent damage:

1) **Logging**: Detailed logging of all actions from potential attacks for future analysis.
2) **Exception**: Throw a SecurityException when a violation is detected.
3) **Shutdown**: Stop the attack by shutting down the application when a violation is detected.
4) **Default values**: Return a statically defined field or method value instead of the actual value.
5) **Custom countermeasures**: Replace the illegal access to the security-critical member with a call to an operation that implements a countermeasure.

Some countermeasures can be used as the agent’s default response, while others must be customized for the security-critical members. Next, we describe the countermeasures in detail and how they can be combined by developers.

**Logging**: The simplest response is to log the call or access that led to the violation and everything that happened after that. This countermeasure can be combined with any other countermeasure. The log can be a classic textual log file or a sequence diagram, such as the one generated by our automated system adaptation described in Section 4. Logging alone will not prevent damage, but it will allow system developers to study the breach and adapt the system to prevent future damage.

**Exception**: The first active response is oriented on how the Java Security Manager responds to a security violation. In the same way, we allow to throw a SecurityException.

**Shutdown**: The next active countermeasure is to shut down the system and notify the system operator, i.e., provide the logs generated by the first countermeasure. Certainly, shutting down the entire system is undesirable in many cases. Considering systems used in critical contexts, the damage caused by a non-running system can be quite high, and from a risk assessment perspective, higher than a possibly limited data loss. In this sense, an attacker could knowingly cause a security incident to ultimately provoke a shutdown as the ultimate goal. However, for Eclipse secure storage, in combination with logging, this could be a valid option.

**Default values**: In addition to the previous countermeasures, to keep the system running and actively prevent damage, we support modifying return values and field values, and dropping write accesses in the case of a violation. This allows us to collect more detailed logs about the progress of the attack without revealing or modifying sensitive information.

For example, returning null is a common response to unforeseen or unusual situations. This prevents the system from exposing any real data to an attacker. For this reason, our security annotations support statically defined early return values or field values. While the previous countermeasures can be used as default strategies that can be selected when starting UMLsecRT, the active countermeasures are specific to individual members. They can be combined with any of the previous countermeasures to provide special handling for specific members.

**Custom countermeasure**: In many cases, realistic data cannot be specified statically, but must be generated dynamically to pass simple plausibility checks and not cause exceptions to be thrown. For example, an array must contain an expected number of entries that may depend on runtime information. In addition, there may be a need for additional countermeasures to put the system in a fail-safe state and protect other parts of the system from the ongoing attack.

Early return values are defined in both cases, static and dynamic, by a parameter earlyReturn of @Secrecy and @Integrity. This parameter can be any primitive type, String, null, or the name of a parameterless method within the class that should be called to dynamically generate a return value and put the system in a secure state. This method can perform any operation accessible from the scope of the accessed member.

Since the concrete realization of early return values is extremely implementation-specific, we decided not to extend UMLsec at the design model level, but to allow the detailing of security annotations at the implementation level. Our TGG synchronization rules support preserving this detailing when synchronizing security properties between design models and the implementation.

To prevent the accidental use of methods that provide countermeasures during regular program execution, we additionally provide @CounterMeasure: whenever a method annotated in this way is entered, UMLsecRT prohibits this call by returning null. We also provide static editing support for checking compatibility between the countermeasure method’s return types and their usages.

Listing 4 exemplifies the use of calling an additional method to put the system into a secured mode and get an early return value: secure(): String is called when a security violation of the secrecy property of the get method occurs at runtime. This method generates a random password that is returned instead of the real one and calls a function securedMode of a central entity. For example, the system could implement permission checks for critical operations, which could always be denied from now on. We allow developers to interact with all parts of the system using arbitrary Java source code, while still enforcing «secure dependency».
public class SecurePreferencesWrapper implements ISecurePreferences {
    @Secrecy(earlyReturn = "secure")
    public String get(String key, String def) {
        return node.get(key, def, container);
    }
}

Random random = new SecureRandom();
StringBuilder s = new StringBuilder();
for (int i = 0; i < 10 + random.nextInt(10); i++) {
    s.append((char) random.nextInt('z' - 'a') + 'a');
}
return s.toString();

Listing 4: Specification of a countermeasure.

4 Automated System Adaptation

After a security vulnerability is discovered, even if it has been mitigated by UMLsecRT, the system must be improved to reduce the attack surface with respect to that vulnerability. Especially for systems with plugins or accessible over the Internet, system models may not cover all possible ways to extend or access the system. Here, the data logged by UMLsecRT can be helpful. While a simple log file of what happened can be difficult to understand and map to the architecture. For specifying call sequences, UML provides sequence diagrams [61]. Sequence diagrams allow developers to easily understand which parts of the system are involved in a particular call sequence because the corresponding model elements are used directly in the diagram.

To address these issues, we propose an automated adaptation of the UML system models as 5th step in Figure 1. This automated evolution includes

1) adding missing UML elements to the system model,
2) and documenting security violations as sequence diagrams with references to involved UML elements.

Since generating such diagrams can be time-consuming and requires additional libraries, such as a library that provides the UML metamodel, UMLsecRT stores data in a custom format at runtime that can be used later for model customization. Figure 7 shows the format for recording information as a class diagram.

For each application monitored by UMLsecRT, when a security violation is detected, a Protocol is created containing information about the date and time the violation occurred (date) and the monitored application (application and path). Also, the current call stack is stored in Protocol and expanded as long as the monitored application is running.

A Call is recorded for each member on the stack or accessed later. They are stored in the order of their addition. To identify the member, this Call contains the signature of the member (member), the fully qualified name of the class defining the member (clazz), and the path from which the class was loaded (bin). Also, each call has a unique ID and contains the ID of the last call to the member from which the current call originated (prevID). Finally, information about violations or countermeasures is stored (violations).

In the remainder, we discuss how these evolution steps can be realized using the gathered data.
Figure 9: Sequence diagram automatically generated by UMLsecRT for visualizing an observed security violation.

To generate sequence diagrams, we need to translate our internal stack structure, as shown in Figure 6, into a sequence diagram. To do this, UMLsecRT translates each method call into a synchronous message in the sequence diagram. For each field accessed, a lifeline is generated, e.g., the lifeline s of type SecurePreferencesWrapper in Figure 9. UMLsecRT also creates lifelines for classes when static members of those classes are accessed, or UMLsecRT cannot determine the variable on which the method is called. While we need to create a message from a start node for the first element on the stack, for all other elements we always have a predecessor from which the corresponding message originates and to which the return message goes. Before the return message is added to the diagram, all predecessors are added to the diagram. Since the list of predecessors is ordered, we automatically get the correct sequence of messages. Algorithm 1 shows this procedure in detail.

As input, we take a Protocol conforming to the specification in Figure 7 and return a UML Interaction containing the sequence diagram. First we initialize three maps in lines 1–3. The first map, names2lifelines, maps pairs of domain and class names to lifelines. The second map provides direct access to a previously processed Call using its ID, and the third map provides access to the return message generated for a Call using its ID. We then initialize a new Interaction in line 4.

Next, we iterate over all the calls in order of addition, starting with the first added call. In each iteration, we first look in the map names2lifelines to check if we already created a lifeline in the interaction I for the current call or not. We get the lifeline if it already exists or create a new one otherwise. Then we look up if we already translated the predecessor of the current call of this iteration. This should always return a Call, except for the first recorded call, which has no predecessor.

In lines 8–17 we determine the appropriate UML message type to represent the current call and, if appropriate, the lifeline from which the current call originates. If we are processing the first call in the call sequence, prevCall is not defined and we will create an asynchronous message. There is also no lifeline from which this message will originate. Otherwise, we get the lifeline for the source of the call in line 9. Next, we distinguish between calls to constructors and calls to methods or fields. We assume that a constructor has the same name as the class in which it is defined. We will create a create message for constructors and a synchronous message for all other members.

In line 18, the message representing the call is created using the previously determined information. Note that UML always treats a message whose source is not set as a found message, and no special treatment is required for the initial call. Next, in lines 19–23, we create the reply messages for synchronous messages as well as the highlighting for active times of a lane. We also add the response message to the map containing all responses.

Finally, in lines 24 through 27, we specify the order of the messages. If no explicit order is given, the messages are added to the end of the lifeline. However, if a message has been called within the active time of another message, we
need to adjust this order. To do this, we retrieve the return message of the predecessor in line 24 and move the message created in this iteration before the retrieved reply. If there was no reply message for the predecessor, there is no need to adjust the order. Finally, at the end of each iteration, we add the call to the map of calls already processed (ids2calls). After processing all calls in the protocol, we return the generated interaction.

5 Tool Support

To evaluate UMLsecRT, we have implemented prototypical tool support for UMLsecRT. The generation of monitored call sequences as sequence diagrams and missing model elements that appear at runtime are also supported. In addition, we show how synchronization between source code and UML models can be achieved.

Figure 10 shows in detail the structure of our implementation of UMLsecRT as introduced in Figure 1. In both figures, Figure 1 and Figure 10, we use the same number labels for the respective steps. In the following, we explain the main features of the implementation in the order of the number labels. The implementation itself can be downloaded from [63].

5.1 Java Annotations and IDE Support

We have implemented the source code annotations specified in Section 3 as Java annotations and provide them to developers in an Eclipse plugin and as a Java library [63]. To further assist developers, we have also implemented a validation plugin for the Eclipse IDE [64] to validate UMLsecRT annotations. This validation ensures that the types specified in early return values are compatible with the annotated fields and return types of annotated methods. We cover not only statically specified early return values, but also the return types of methods called in case of security violations.

5.2 Synchronization of UML Models with Source Code

In steps 1 and 3 of Figure 10, we may need to reverse-engineer UML diagrams from source code and then keep both, including UMLsecRT annotations, in sync. To address this issue, we use a model transformation approach based on triple graph grammars (TGG) [57]. In particular, we use the graph transformation tool eMoflon [58], [65].

The prototypical rules of Leblebici et al. that we extended were written by them to evaluate an eMoflon feature that allows to recreate a correspondence model between two existing models [56]. In our previous work [18], we implemented the synchronization of model and source code by incorporating the TGG rules and improving their quality and support for synchronization of changes. For the needs of this work, we finally extended them with support for UMLsecRT annotations, call dependencies, and deployments.

Altogether, our rule set in eMoflon consists of 92 TGG rules. Listing 5 shows the conceptional TGG rule from Figure 5 implemented in the syntax of eMoflon. The elements from the two models are expressed in $\texttt{#source}$ and $\texttt{#target}$ blocks using an intuitive $\texttt{object:type}$ notion. As in the figures, additions are indicated by $\texttt{++}$ and also highlighted in green by the eMoflon editor. In contrast to the visual specification, eMoflon needs typed correspondences that are defined in a separate eMoflon file. We import this file as modisco.uml and explicitly state that this rule uses the correspondences specified in modisco.uml. The correspondences themselves are specified in the rules in the $\texttt{#correspondence}$ block. Simple conditions over attributes are expressed in the $\texttt{#attributeConditions}$ block. In the example, we use the $\texttt{eq}$ function that we imported from the AttrCondDefLibrary of eMoflon. Custom functions can be defined in separate libraries.

To synchronize using eMoflon, after developers have annotated the model or source code, we need a UML model, a model of the source code, and any changes applied to both. As source code model we use the MoDisco [66], [67] Java metamodel and the MoDisco tooling for parsing or generating source code. To determine the changes in the UML model, the prototype listens directly to model change events generated by a modeling tool, such as Papyrus [68], [69]. Papyrus is a powerful UML editor that directly supports the UMLsec profile if CARiSMA, the current tool support for UMLsec, is installed [70].
be accepted by developers with sufficient privileges. Since the signature check only needs to be performed when a class is loaded, this is a static overhead and has relatively little impact on long-running programs.

5.4 Detecting System Evolution Automatically

While we support synchronization of model and code, there may be associations between the model and source code that are not yet covered and cannot be statically detected. This is especially true for dynamic behavior introduced by libraries and reflective calls. When monitoring program execution, our implementation of UMLsecRT keeps track of every method that has been entered and not yet exited.

The prototype facilitates the graphical representation of the observed call flows by creating sequence diagrams (step 5 of Figure 10). Since our tool can keep track of every method and field that is accessed, we can continuously check if a call edge detected in the monitoring has corresponding elements in the model. If not, the tool can feed this information into the model by adding appropriate elements.

6 Evaluation

We evaluate the applicability of UMLsecRT and its tool support by considering five research questions:

RQ1–Effectiveness: Can we detect real-world security violations using UMLsecRT?

RQ2–Applicability: Can we monitor real-world Java programs with a reasonable runtime overhead?

RQ3–Overhead: What is the monitoring overhead compared to other monitors, and what influences it?

RQ4–Usefulness: How useful are the countermeasures and do the adapted UML models facilitate investigating security violations?

RQ5–Scalability: How does the TGG-based synchronization between models and code scale?

In the following, we present the research questions in detail, the methodology, and the results of the evaluation. We conducted the experiments on a system equipped with an Intel i5-6200U CPU, 8 GB RAM, and running Oracle JDK 8 on Ubuntu 20.04 LTS. All evaluation data and implementations are available in our replication package [63].

6.1 RQ1–Effectiveness of the Security Monitor

We study the effectiveness of UMLsecRT for detecting vulnerabilities and compare it to other security monitors.

6.1.1 Setup

For this evaluation, we studied the causes of real-world security violations related to the secrecy and integrity properties of «secure dependency», reproduced them, and evaluated the mitigation of these violations.

Common software weaknesses are collected in the Common Weakness Enumeration (CWE) using a unique ID for each entry [73]. However, the presence of a vulnerability that is an instance of a weakness does not imply that the vulnerability can be actively used to perform malicious actions. Nevertheless, vulnerabilities in software should be detected and fixed, which is not the scope of our work, but to prevent

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author’s version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3326366
the running system from being harmed if vulnerabilities are exploited. In Table 2 we briefly summarize the CWEs considered in our evaluation and how they are mitigated by UMLsecRT when instances of them are exploited.

A common benchmark for static weakness detection approaches is the Juliet test suite, which was created to study their effectiveness and accuracy. For many CWEs, the Juliet test suite provides a database of good and bad code examples [74], [75]. Unfortunately, it does not provide examples of how to exploit the weaknesses maliciously. Such an exploit is required to violate «secure dependency» at runtime and create a situation where a security monitor needs to intervene. However, the Juliet test suite provides a good basis for systematically implementing such cases.

For example, CWE470 – Unsafe Reflection states that using the Java reflection API to load classes based on external data is dangerous. To statically detect this vulnerability, you must determine whether the values passed to the reflection API (such as the name of a class to load) are generated from external data. A small change in how the name of a class is passed to the API can have a huge impact on detection, and collecting such different variations is the purpose of the Juliet test suite. While detecting all these different variations of a single weakness is challenging statically, concrete values can be inspected at runtime. Because the Juliet test suite is designed for static analysis tools, the examples for CWE470 end as soon as a class is loaded based on external data. The same is true for the other CWEs considered in Juliet.

At runtime, we cannot change the underlying implementation of a system to eliminate weaknesses, but we must mitigate the exploits. Since we can perform checks whenever a class is loaded, we need to evaluate not whether we can detect the loading of a class, but whether we can detect malicious actions that a class performs. Here we have three possibilities to consider: malicious method calls, as well as read and write accesses to fields. While all three are possible for CWE470, this is not the case for other CWEs. For example, write accesses cannot be used to expose sensitive data as considered in CWE200.

Therefore, based on the Juliet test suite and our research on CWEs, we created executable test programs to investigate the effectiveness of runtime monitoring. To do this, we iterated through all of the Juliet test programs and attempted to build an exploit that violates a security property of «secure dependency». We did this by directly annotating the source code with the security annotations, bypassing the model level. In each case where we successfully built an exploit, we consider, similar to Juliet, two types of test programs, positive and negative test programs. Each positive test program contains an exploit that must be detected during runtime monitoring. In summary, we have built test programs based on the 13 CWEs shown in Table 2. For example, the violation shown in Listing 1 of the running example is an instance of CWE829 that uses CWE470 to perform an illegal method call that results in data disclosure (CWE200). According to Listing 4, this is mitigated by calling a countermeasure. In this experiment, we throw a SecurityException whenever a «secure dependency» violation is detected by UMLsecRT. In Table 3, this case is used to test the secrecy case of a method call for the violation in the first row. Each negative test program corresponds to a positive test program in that it covers the same language construct but does not contain a security violation, e.g., because the security annotations are consistent.

All of the test programs are about the size of this example and, where possible, have been created for secrecy and integrity cases of field accesses and method calls. Our test programs cover calls to and from external libraries, reflective access to fields and methods, reflective instantiation of objects, code injections into a Javascript engine, and a deserialization attack. In total, we specified 13 different types of test programs with 37 expected security violations. For each expected violation, we have an additional test program where the same action takes place but no violation occurs. The total number of test programs is 74, and our replication package contains all of them [63]. Table 3 summarizes the test programs and which CWEs they address.

We tried to compare UMLsecRT with a variety of security monitors and we were able to implement «secure dependency» using the Java Security Manager and Larva [76].

<table>
<thead>
<tr>
<th>CWE</th>
<th>description &amp; mitigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Exposure of Sensitive Information</td>
</tr>
<tr>
<td>209</td>
<td>Sensitive Information in Error Message</td>
</tr>
<tr>
<td>226</td>
<td>Sensitive Information Uncleared in Resource</td>
</tr>
<tr>
<td>327</td>
<td>Broken Cryptography</td>
</tr>
<tr>
<td>328</td>
<td>Reversible One-Way Hash</td>
</tr>
<tr>
<td>470</td>
<td>Unsafe Reflection</td>
</tr>
<tr>
<td>481</td>
<td>Assigning instead of Comparing</td>
</tr>
<tr>
<td>486</td>
<td>Comparison of Classes by Name</td>
</tr>
<tr>
<td>498</td>
<td>Clonable Class Containing Sensitive Data</td>
</tr>
<tr>
<td>499</td>
<td>Serializable Class Containing Sensitive Data</td>
</tr>
<tr>
<td>502</td>
<td>Deserialization of Untrusted Data</td>
</tr>
<tr>
<td>586</td>
<td>Explicit Call to Finalize</td>
</tr>
<tr>
<td>807</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
</tr>
<tr>
<td>829</td>
<td>Functionality from Untrusted Control Sphere</td>
</tr>
</tbody>
</table>
In the following, we report on how we implemented the security checks and the problems we encountered.

**Java Security Manager:** To detect the security violations, we had to add explicit calls to its `checkPermissions` method to the test programs, in addition to the security annotations. In Listing 6, we show how we implemented the security check for the example from Listing 2.

At the beginning of each method for which a security property has been specified within the security annotations, we have added a call to the `checkPermissions` method with a `RuntimePermission` parameter. Similar to the security annotations, we specify the method signature for which a security property has been specified as a unique identifier in the `RuntimePermission`. We can then refer to the signature in the policy file of the Security Manager to specify who should have access to the corresponding method. If access is granted, the `checkPermissions` method does nothing, otherwise it throws a `java.security.AccessControlException`.

In policy files, we can grant permissions to code bases, which are classpath folders or jar files. Listing 7 shows a snippet of the policy file for the example. We grant permission for the signature `get(String, String)` recursively on the entire classpath entry `bin`, assuming that malicious classes are not part of the application and are loaded from somewhere else. This means that any class located in the `bin` folder is allowed to call the method `get(String, String)`. In the experiments, we put the malicious implementations in a separate classpath folder `bin-exploit`.

**BeepBeep:** Under this name, two stream processing approaches have been developed that have been used to verify security properties at runtime [31]. We have tried to express `secure dependency` in both.

The older version of BeepBeep uses LTL-FO+ [77] linear temporal logic to express constraints that can be monitored. A script on the BeepBeep website [78] generates AspectJ code from such a specification, allowing BeepBeep to monitor the constraints at runtime. The constraint is expressed on an XML representation of the call stack, which consists of call entries that have a `method` tag containing the name of the method that was called. Accordingly, we have expressed `secure dependency` in LTL-FO+, using some placeholders.

$$\forall \text{call/method} m_1 : \forall \text{call/method} m_2 :$$

$$X((m_2 \in \text{secrecy}(m_2) \land m_2 \in \text{secrecy}(m_2))$$

Since we need to check all pairs of methods, we have a global constraint in the call tree that must hold on all method calls $m_1$ and implies a constraint to check on every next method call $m_2$. The constraint is expressed on an XML representation of the call stack, which consists of call entries that have a `method` tag containing the name of the method that was called. Accordingly, we have expressed `secure dependency` in LTL-FO+, using some place holders. Due to the bi-implication of `secure dependency`, we must always consider the pair and cannot check on $m_1$ only in advance. Therefore, we have to check on every next method call $m_2$ that the `secure dependency` property must hold, i.e., $m_2$ is at the security level of secrecy for either $m_1$ and $m_2$ or neither of them. We express this condition using a helper function `secrecy`, which for a method name returns the method names that are on the secrecy security level.

Practically, we faced two problems that prevented the implementation of this constraint. First, it is not possible to implement the required helper function `secrecy` in BeepBeep because there is no way to inject additional information into the constraints. One would have to extend the BeepBeep call stack with this information and add set logic to LTL-FO+, or find a way to manipulate each AspectJ file after generating a stub. Second, all methods that need to have this check added must be explicitly declared. While this is a huge overhead, it is possible for the methods to be known at design time and could be automated. However, as discussed above, to ensure that the code injected or code from libraries, does not violate the security level, the check must also be run on them. Since LTL-FO+ only supports forward logic and we cannot name these methods in advance, this cannot be expressed using BeepBeep.

The newer BeepBeep3 has been used explicitly to verify security properties at runtime [31]. BeepBeep3 is a pure stream processing approach that allows the specification of processing pipelines by wiring different processing nodes. When used for security monitoring, the authors manually implemented aspects using AspectJ that use this stream processing for verifying security properties. If we wanted to use BeepBeep3, we would also have to implement our own aspects, resulting in the same problems as for BeepBeep, or we could inject the BeepBeep3 processing pipeline using the code instrumentation of UMLsecRT. In the same injected implementation, we have to manually extract the information about the annotations to pass it to a BeepBeep3 stream processing network. Considering the stream processing itself, there is currently no node that allows us to express the UMLsec Secure Dependency property, and we would have to implement a custom node to do so. What we could express using the nodes of BeepBeep3 is the handling of the call stack as demonstrated by Boussaha et al. [31]. However, they implemented this in a processing pipeline containing 9 processing nodes with complex wiring, while the same logic is implemented in 3 lines of code (`Stack.get/pop`) in the code injected by UMLsecRT. In summary, we could use BeepBeep3 in the code we inject using UMLsecRT to express `secure dependency` in combination with custom code, but it would just add more overhead. Therefore, we decided not to perform this experiment with BeepBeep3.

**Larva:** The Larva monitor [76] allows to express a state machine that models valid and invalid states, where the

```java
public String get(String key, String def) {
    System.getSecurityManager().checkPermission(new
        RuntimePermission("get(String,String)")
   );
    ... // Unchanged implementation of the method
}
```

Listing 6: Implementation of the test programs using the Java Security Manager.

```java
grant codeBase "file:${user.dir}/bin/*" {
    permission java.lang.RuntimePermission *get(String, String)*;
}
```

Listing 7: Example policy for the Java Security Manager.

$$G(\forall \text{call/method} m_1 : \forall \text{call/method} m_2 :$$

$$X((m_2 \in \text{secrecy}(m_2) \land m_2 \in \text{secrecy}(m_2))$$

$$\lor (m_2 \notin \text{secrecy}(m_2) \land m_2 \notin \text{secrecy}(m_2))))$$
transitions are triggered by the execution of methods of the monitored program. Furthermore, Larva allows to express variables that could be used to hold the last method call to compare its security properties with the next one. Conditions can be expressed for concrete objects. However, since «secure dependency» expresses constraints on the structure of a software system and not on concrete object instances, which are application specific and must be known by the one specifying the security constraint, we cannot use this to express Secure Dependency. Furthermore, like BeepBeep, Larva requires explicit specification of the methods that can trigger a state change. Also, Larva does not allow direct access to the required security annotations, although method triggers can be combined with custom Java code, but the method object itself is not accessible.

By generating configurations for all methods included in the implementation of the system, and using an external specification of security annotations in a JSON file that maps method names to security annotations, combined with custom Java functions injected into Larva, we were able to specify the Larva runtime monitor as shown in Figure 12. As for UMLsecRT, we will only explain the case of secrecy. We defined two variables, stack and secrecyMap, which hold the current execution stack and a map with all secrecy annotations for the methods of the implementation, which is initialized at the start of Larva by reading the values from a JSON file. As in UMLsecRT, we push a method onto the stack when it is executed and remove it when it returns, represented as enterMethod and exitMethod events in the state machine. Here we had to explicitly specify which method triggers which event for each method of the implementation. We also created a new parameter in these events that is initialized to a method name when the method is entered, to make this information accessible. When a enterMethod event occurs, we first check if Secure Dependency is fulfilled, using the same Java implementation as in UMLsecRT, which is provided as a checkSecureDependency method to the state machine that used the two method IDs to get the corresponding annotations to compare them. If this is not satisfied, the bad state is entered and Larva allows execution of arbitrary Java source code. Otherwise, monitoring and system execution continue.

In summary, this implementation allows us to monitor Secure Dependency for method calls, but not for field accesses. Since the Secure Dependency check is based on method names, unlike UMLsecRT, which works on concrete method and class objects, this monitor is vulnerable to the Comparison By Name vulnerability (CWE486).

6.1.2 Results
While the specification of the test programs was straightforward for UMLsecRT, this was more challenging for the Java Security Manager and Larva. In order to run the test cases, we had to add explicit calls to checkPermissions for the Java Security Manager, and we had to extract the test logic into a wrapper method for Larva, since we found that it did not allow the main method to be considered. While UMLsecRT and Larva support different types of security properties, the standard Java Security Manager does not. Therefore, we implemented the security checks using the Java Security Manager without distinguishing between the different security properties.

The results of the experiment are shown in Table 3. A checkmark indicates that the vulnerability was successfully mitigated, and a cross indicates that the vulnerability could not be mitigated. For some test programs, not all cases make sense, e.g., CWE209 – Sensitive Information in Error Message cannot lead to an integrity violation. UMLsecRT detected all the expected security violations without a single false positive (100% precision and recall). Also, none of the other security monitors detected any false positives, but they detected varying numbers of true positives, resulting in a recall of 30% for Java Security Manager and 27% for Larva. We discuss these results in detail below.

A general limitation of the Java Security Manager and Larva that we observed is that it is not possible to check field accesses. Accordingly, we consider all test programs with forbidden field accesses to have failed. An exception to this is reflective field access for the Java Security Manager, for which it provides the ability to check whether the use of Java reflection is allowed for the location from which the class was loaded, but not to check against the security property of the field. However, since some sort of security check can be expressed, we consider this a partial success. The same is true for method calls executed through Java reflection.

Since a primary goal of «secure dependency» is not only to protect against attacks, but also to mitigate security violations caused by bugs within the implementation, it is essential to consider illegal accesses on the method level. While UMLsecRT and Larva support this granularity, the granularity of the Java Security Manager does not allow us to specify security checks within a single classpath entry.

Finally, the Java Security Manager only allows us to check the invocation of methods under our control, as developers must explicitly call checkPermission, but not whether an external method we call provides the expected security properties. Similarly, in Larva, the methods to be included in the security checks must be explicitly defined, which prevents checking accesses involving unknown methods, making it impossible to check injected code or calls from external plugins. Since the library APIs used by the system are known to the developers, they can be added to the Larva configuration. However, they are not automatically updated when a library’s security guarantees change, which leads us to consider this case a partial success.

In summary, while the Java Security Manager can be effectively used to check incoming method accesses that originate from classes stored on a different classpath than the code to be protected, it does not provide sufficient granularity and expressiveness to enforce UMLsec security policies at runtime. While Larva allows to work on the required granularity, it is not possible to specify all the unknowns upfront as it would be necessary in Larva.
TABLE 3: Effectiveness of UMLsecRT and the Java Security Manager: ✓ – mitigated, (✓) – partly mitigated, × – not mitigated, N/A – not applicable

<table>
<thead>
<tr>
<th>Kind of Action Executed in the Test Programs</th>
<th>CWEs</th>
<th>UMLsecRT</th>
<th>Security Manager</th>
<th>Larva</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A plugin accesses critical members of the host</td>
<td>200, 226, 486, 807, 829</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>2. Internal bug: Security properties of source violated</td>
<td>200, 807</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>3. Internal bug: Security properties of target violated</td>
<td>200, 807</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>4. Accidental assignment to field but only read rights</td>
<td>481, 807</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>5. Dynamic loaded class accesses data</td>
<td>200, 226, 486, 807, 829</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>6. Injected JavaScript code into the Rhino engine</td>
<td>200, 226, 807, 829</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>7. Call printstacktrace of sensitive exception</td>
<td>200, 209</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>8. Reflective access to critical members</td>
<td>200, 226, 470, 807</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>9. Call to finalize with insufficient privileges</td>
<td>586</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>10. Cloning of a class containing sensitive data</td>
<td>200, 498</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>11. Serialization of class containing sensitive data</td>
<td>200, 499</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>12. Replacing class at deserialization</td>
<td>200, 502, 807, 829</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>13. Unsecure method/field in new library version</td>
<td>200, 326, 327, 328, 807</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>

6.2.1 Setup

To consider real-world programs with realistic program executions, we applied the monitoring component of UMLsecRT to the DaCapo benchmark suite [44]. DaCapo has been actively maintained and supported by the industry since 2006. In version 9.12, DaCapo consists of 14 real-world open source applications (the Tomcat benchmark is currently broken and therefore excluded by us [79], [80]) that perform typical tasks. These include indexing or searching large documents such as the King James Bible using Apache Lucene (luindex and lusearch) and transforming XML to HTML (xalan). A list of benchmarks is given in Table 4. Since the monitoring code is executed regardless of whether UMLsecRT annotations are present in the code or not, we do not need to annotate the DaCapo benchmark applications to evaluate the overhead of UMLsecRT.

As part of this research question, we performed two experiments. First, we measured the execution time for each DaCapo benchmark, both with and without monitoring. Since DaCapo is provided as an executable jar file that implements its own non-trivial class loading for the individual benchmarks, we could only monitor the benchmarks with UMLsecRT, as we could not modify the implementation as needed for the Security Manager and do not know all the methods of the benchmarks as needed for Larva. Second, to learn about the reasons for the expected slowdown, we profiled what percentage of the DaCapo benchmark’s execution time was spent on which tasks.

6.2.2 Results

We were able to monitor 10 benchmarks successfully, and had problems with 3 benchmarks using jython or geronimo. These throw a java.lang.VerifyError–Inconsistent stack height exception when the programs themselves use bytecode instrumentation after UMLsecRT has made changes. Since this exception is also thrown when we just insert code that does not change the behavior, the cause does not seem to be UMLsecRT. Despite these 2 programs, there seem to be no problems with monitoring real-world programs.

The execution times with and without security monitoring are shown on the right side of Table 4. On average, execution with security monitoring is 4.1 times slower than without. If we look at the details of the different benchmarks, we can see a notable difference in the slowdown between the individual benchmarks. h2 has a relatively small slowdown with a factor of 2.2, while Eclipse has the highest slowdown with a factor of 8.4.

Figure 13 shows the distribution of time needed for central parts of UMLsecRT during benchmark executions. These are instrumenting classes, checking security annotations, creating new annotation objects, representing members and their annotations, and retrieving the stack corresponding to the current member. The benchmarks in the figure are sorted in descending order by their slowdown.

TABLE 4: Benchmarks of the DaCapo benchmark and measured execution times.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>project characteristics</th>
<th>classes</th>
<th>methods</th>
<th>fields</th>
<th>execution time in ms plain Java</th>
<th>UMLsecRT</th>
<th>slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td></td>
<td>1,741</td>
<td>19,575</td>
<td>27,789</td>
<td>4,576</td>
<td>12,213</td>
<td>2.7</td>
</tr>
<tr>
<td>batik</td>
<td></td>
<td>2,121</td>
<td>66,734</td>
<td>350,799</td>
<td>4,195</td>
<td>14,145</td>
<td>3.4</td>
</tr>
<tr>
<td>eclipse</td>
<td></td>
<td>407</td>
<td>5,357</td>
<td>3,359</td>
<td>47,625</td>
<td>395,534</td>
<td>8.4</td>
</tr>
<tr>
<td>top</td>
<td></td>
<td>1,204</td>
<td>29,814</td>
<td>86,919</td>
<td>2,137</td>
<td>15,749</td>
<td>7.4</td>
</tr>
<tr>
<td>h2</td>
<td></td>
<td>441</td>
<td>13,745</td>
<td>6,884</td>
<td>7,906</td>
<td>17,699</td>
<td>2.2</td>
</tr>
<tr>
<td>luindex</td>
<td></td>
<td>491</td>
<td>6,313</td>
<td>2,869</td>
<td>1,204</td>
<td>29,814</td>
<td>86,919</td>
</tr>
<tr>
<td>lusearch</td>
<td></td>
<td>491</td>
<td>6,313</td>
<td>2,869</td>
<td>1,994</td>
<td>6,472</td>
<td>3.2</td>
</tr>
<tr>
<td>pmd</td>
<td></td>
<td>644</td>
<td>35,606</td>
<td>49,432</td>
<td>4,138</td>
<td>13,595</td>
<td>3.3</td>
</tr>
<tr>
<td>sunflow</td>
<td></td>
<td>220</td>
<td>1,653</td>
<td>990</td>
<td>7,154</td>
<td>19,251</td>
<td>2.7</td>
</tr>
<tr>
<td>xalan</td>
<td></td>
<td>1,419</td>
<td>52,200</td>
<td>72,989</td>
<td>4,879</td>
<td>19,046</td>
<td>3.9</td>
</tr>
</tbody>
</table>
We can see that the slowdown does not depend on a single activity. On average, 56% of the slowdown is due to instrumenting classes, 2.7% to checking security annotations, 5.4% to creating new annotation objects, and 35.9% to stack fetching. However, there are huge differences between the individual projects. Analyzing the data, we can identify two groups, one that spends most of its time on stack fetching and the other on class instrumentation.

Looking more closely at the execution times, we see that the projects with the lowest overhead are the ones that take the longest even in unmonitored execution. One exception is Eclipse, where the OSGi class loader and the structuring into plugins cause a high instrumentation overhead. A second exception is fop, where the high instrumentation overhead due to the many classes combined with the short runtime of the benchmark leads to a high slowdown. The very high instrumentation overhead for batik can be explained by the excessive number of fields that are all checked at class loading and the many methods that need to be instrumented. The same is true for fop, pmd, and xalan. All in all, the slowdown seems to decrease with execution time. This, as well as the average static instrumentation overhead of 56%, indicates that UMLsecRT has a lower slowdown for long-running applications than the measured average slowdown.

We performed an additional experiment to verify the hypothesis that UMLsecRT has an acceptable slowdown on long-running systems. We set up a Java Web application and monitored the application’s response time to simulated user input over an extended period of time with and without the UMLsecRT security monitor.

As a security-critical web application, we chose the iTrust [47] electronic health records system, which allows doctors and other hospital staff to manage patient treatments. The iTrust system was implemented as a class project over more than ten years using Java and Java Server Pages [45]. We deployed iTrust version 21 on a Tomcat 8 [81] server whose response times we measured using JavaMelody. We simulated user interaction using the Selenium Chrome IDE. Our simulated user behavior represents a doctor logging into iTrust, navigating to a statistic generation functionality, generating a statistic about the number of influenza cases in a region, which iTrust presents as a table and a generated graph, and then logging out of iTrust. In total, this behavior consists of 22 commands in Selenium, and we ran it 1,000 times with and without UMLsecRT.

As for DaCapo, running this experiment with the Java Security Manager would require us to modify all methods of iTrust. However, we were able to extract all methods as needed for Larva and generate the AspectJ monitoring code using Larva. Unfortunately, we were unable to run the aspects on the Tomcat server. We tried different versions of AspectJ and weaving commands, but we either got compile-time weaving errors or the aspects were not executed.

Before running the experiment, we measured the time it took to start Tomcat along with iTrust, as reported by Tomcat. We observed a comparable slowdown to the previous experiment with an average factor of 2.34 for starting Tomcat, 10 times with and 10 times without UMLsecRT.

If we look at the recorded response times in Figure 14, we can confirm our hypothesis. Running the 1,000 simulated sequences took about an hour each time. When we run iTrust without UMLsecRT, it has a relatively consistent response time of 19ms. With UMLsecRT, the average response time is 21ms. We can see that the response time with UMLsecRT decreases slightly after all classes have been instrumented. After 33 minutes, the response times fluctuate a bit, which seems to be due to Tomcat running background tasks and loading additional classes. Considering the average response time of 21ms, this is only 10% higher than without the security agent. Even in the worst case, for the peak after 20 minutes, the difference is only 9ms. Since we ran the experiment with iTrust deployed on the localhost, the observed slowdown in response times is negligible when considering the additional static overhead that is added due to network communication in a remote server deployment.

6.3 RQ3—Monitoring Overhead and Influencing Factors
In Q2, we investigated whether it is possible to monitor real applications with UMLsecRT. While we were able to measure the monitoring overhead for UMLsecRT, we were unable to monitor the DaCapo benchmarks and iTrust using the other security monitors. In this section, we explicitly compare the monitoring overhead of the different monitors.

We have also seen that UMLsecRT can, in principle, monitor long-running applications with acceptable overhead. To gain further insight, we will examine the overhead as a function of the number of classes to be instrumented and the frequency with which their methods are executed.

6.3.1 Setup
In order to systematically investigate the overhead of the runtime monitors, we generated Java programs of different sizes, which are reduced to the main factors influencing the monitoring. Since the logic implemented in the methods of a program does not interfere with the «secure dependency»
security property, we generated sequences of methods that call each other but do not contain any other logic except for printing the name of the executed method to the terminal and calling the next method in the sequence. To reflect the structure of real programs, we generated classes containing ten methods each. The first class contains a main method that calls the first method of the sequence in each class in a loop, where the number of iterations is given as a command line argument. In this way, we are able to explicitly measure the monitoring overhead with respect to two influencing factors, the number of methods and the number of times each class or method is executed.

We created multiple versions of each program to address the different security monitors. A version that runs without security monitoring and can also be monitored by UMLsecRT. A version in which every method is extended in its first line with a call to the `checkPermission` method for the Java Security Manager, and two versions that will be enriched with aspects using Larva and compiled using load-time and compile-time weaving of AspectJ.

We generated programs of 10 to 10,000 classes, containing between 100 and 100,000 methods (10 methods per class), and executed the method sequences in each of these programs between 1 and 2,000 times. Since the constant pool in Larva is limited to 65,536 constants, we could not weave the security aspects into the Java programs with 1,000 or more classes. We were also unable to apply Larva to a Java program with 500 classes due to another AspectJ error, but all smaller programs were monitored as expected.

### 6.3.2 Results

We measured 294 data points for each security monitor, except for Larva where we were only able to capture 168 data points. All of the raw data is included in our replication package [63]. To visualize these results in Figure 15, we used these data points to generate a 3D surface of execution time as a function of the number of monitored classes and the number of times each method was executed.

The green surface at the bottom of Figure 15 represents the execution times of the plain Java program without security monitoring. As expected, the execution times with the Java Security Monitor (orange surface) are only slightly longer than the plain Java program for only a few executions, since the monitoring code was already added at code generation. For many executions, execution times converge towards the execution times with UMLsecRT monitoring, shown by the blue surface above. As expected based on our observations in the experiments to answer O2, the static overhead increases mainly with the number of additional classes that need to be instrumented. On the left, we see the execution times with Larva using compile-time weaving. Contrary to expectation, we did not observe a significant difference between compile-time and load-time weaving. Since load-time weaving tends to be slightly faster, we decided to show these times in Figure 15. Even for the smallest program, Larva is significantly slower than UMLsecRT, and the execution times rise steeply.

Since Figure 15 shows only an interpolated grid calculated from the measured data, we show in Figure 16 two concrete data series for a fixed number of classes or a fixed number of executions. Note that while the number of executions per method is fixed in Figure 16a, the total number of method executions still increases due to additional classes containing additional methods that are executed. We have chosen the number of classes and the number of method executions so that there are no visible outliers in Figure 15 for any of the measured monitors. In Figure 16b one can clearly see the initial instrumentation overhead of UMLsecRT and how it relativizes over time. Since we chose a data series for 5,000 classes, we do not have a measurement for Larva, but it is included in Figure 16a. You can clearly see that it scales worse than monitoring with UMLsecRT or the Java Security Manager.

Our measurements confirm our assumption that UMLsecRT scales well for long-running applications. Due to the initial instrumentation overhead, we also observe significantly longer execution times of short-running applications in this experiment, as we also observed for the DaCapo benchmarks, but the overhead is static and relativizes for long-running applications. At monitoring a `secure dependency`, UMLsecRT outperforms Larva, which is only applicable to small applications and has significantly higher execution times than UMLsecRT for them. UMLsecRT even achieves a slightly better runtime overhead than the Java Security Manager for long-running applications.
6.4 RQ4–Usefulness of the Supported Countermeasures and Adaptations of Design Models

We conducted two user studies to evaluate the usefulness of UMLsecRT. First, we qualitatively investigated how the provided countermeasures are perceived by developers and whether they find them useful and practically applicable. Second, we quantitatively investigated whether the result of the adaptations made by UMLsecRT when a countermeasure is executed is a useful addition to the current way of displaying such information.

6.4.1 Setup

Since the aspects we are interested in are different for the countermeasures and the model adaptations, we conducted two conceptually different user studies. First, we qualitatively investigated the usefulness and applicability of UMLsecRT and in particular the countermeasures. Due to the broad scope of this study, we conducted interviews with experts. Second, we investigated the usefulness of adapting UMLsecRT in user surveys due to the direct comparability with the current state of practice.

Countermeasures: To gain insight into the usefulness of the countermeasures supported by UMLsecRT, their practical applicability, and the needs of the industry, we conducted semi-structured expert interviews. We sought security experts and experienced developers from industry to participate in our survey. We were able to recruit six experts from five companies to participate in our interviews. The participants had between 4 and 22 years of experience, with a median of 6 years.

The first interviewee works as a security expert for one of the world’s largest automotive companies. In the same domain, we had two experts currently working for two of the world’s largest automotive suppliers. One works as an innovation manager in the area of automotive security and the other as a security manager for general automotive technologies. Two experts work for the same consulting and software development company but for different clients, one in embedded software and the other in insurance software. The last expert was a software developer working for one of the world’s largest IT consulting companies as a web application developer on client projects.

In the interviews, we first introduced the general idea of UMLsecRT. Then, for each of our countermeasures, we asked the participants for their professional opinion on advantages, disadvantages, practical applicability in the industry, and what they would need that is currently missing.

Model adaptations: To investigate whether the adaptations of UMLsecRT are useful for inspecting security violations, we conducted a quantitative user study asking developers about the usefulness of three different representations of a security violation, two of which were generated by UMLsecRT. Therefore, we introduced the Eclipse Secure Storage, which is also used as a running example in this work, to the participants as a subject system. We then showed them three representations of a security violation caused by an Eclipse plugin executing an implementation similar to the one shown in Listing 1 in a startup action. The security violation to be inspected is an illegal access to the `get(String,String)` method of the Eclipse Secure Storage. One of the representations reflects the current state of practice, and the other two are representations automatically generated by UMLsecRT:

1. The first representation is the stack trace of a security exception, as it would be currently available when investigating a security violation.
2. The second representation was the generated deployment diagram (similar to Figure 8).
3. The third was the generated sequence diagram generated by UMLsecRT (similar to Figure 9).

In order not to bias the participants, we did not tell them that two of the representations were generated by UMLsecRT, but that we were investigating the usefulness of security violation representations. For all representations, we asked them to identify key aspects of the vulnerability, both to force them to examine the representations in detail and to serve as control questions (although the answer was sometimes not 100% correct, there were no completely implausible answers). Next, we asked participants to write down the advantages and disadvantages of all representations. Finally, participants were asked to rate the usefulness of each representation for investigating a vulnerability on a scale from one for not useful to five for very useful.

A total of 39 experienced software developers participated in this quantitative user study. Of these developers, 51% had more than 10 years of experience and another 26% had more than 5 years of experience. Three respondents had less than 3 years of experience. The majority of our respondents have an academic background (29 respondents), but we also had 9 respondents who were employed in industry. Two of these participants indicated that they were also students, and one participant identified only as a student.

6.4.2 Results

Both user studies show that UMLsecRT provides practical, useful, and applicable solutions for countermeasures and improving the inspection of security violations. In the following, we discuss in detail the results of our user studies, the lessons learned, and the potential for future research.

Countermeasures: For all proposed countermeasures, including logging only, the experts identified scenarios in which these countermeasures are appropriate. The usefulness of the countermeasures depends on the use case, particularly for the responses of logging only, security exception, terminating the system, and continuing execution with default values. While we expected most of these, we were surprised to find that the security exception is also considered quite problematic. For custom countermeasures, it is mainly the required effort that limits broad applicability.

Logging: More exhaustive logging was seen as an essential requirement to facilitate the investigation of security violations and to facilitate the improvement of the system. However, only some of the participants considered logging alone as a valid option in completely uncritical areas. The majority called for a combination with other countermeasures. One expert noted that while such advanced logging may be very helpful to developers, it may also raise privacy issues that need to be addressed.

Exception: In particular, one developer mentioned that injected code could also catch the security exceptions...
thrown by UMLsecRT, which would likely reduce its effectiveness. Also, attackers could learn the system’s security responses. The need to handle the security exceptions in their own application was one of the issues mentioned, and that it could be a challenge to implement this properly, especially since all developers would have to deal with this type of exception. In addition, it would be a challenge to test the exception handling that becomes part of the functional code.

**Shutdown:** Terminating the system was seen as a good solution for systems without strict availability requirements, but also for microservices where new instances are started on demand. For most systems, however, this should be a last resort. Since we have not yet introduced our active countermeasures, most participants asked for a way to make this decision context-specific.

**Default values:** Specifying defaults was mostly seen as a doable task that should be part of the system documentation anyway. It was also thought that the need for explicit specification might force developers to think more about such cases from the beginning. In the opposite direction, default values were seen as something that could easily be generated by tool support. However, a major concern was whether UMLsecRT would really be capable enough to prevent future damage from continuing execution, or whether an attacker would be able to find a backdoor. The ability to gain deeper insight into an attack was appreciated, but assumed to be limited to a few use cases. Finally, the fact that the control flow does not change, as opposed to having exceptions, was seen as an advantage from a testing perspective.

**Custom countermeasures:** All participants considered this variant to be the one that meets all their requirements. Furthermore, they all assumed that the overhead would still be manageable, especially if this variant was not used in all locations. However, the developers in particular were skeptical about the practical implementations, as they did not consider themselves sufficiently trained in security to implement the countermeasures. The security experts, on the other hand, were enthusiastic about this countermeasure because it allows them to separate security code from logical code and to implement sophisticated responses for critical parts of the system. To facilitate such tasks, it was asked whether one could invoke Helm charts, or change permissions in role-based access control (RABAC). Identifying such standard tasks and providing code generators was one of the main wishes for our future research.

In summary, the countermeasures of UMLsecRT are applicable in practice, and the standard scenario would be as follows: Participants would enable advanced logging, and for critical parts of the system, security experts would define sophisticated measures. They would include an assessment of the current state and, based on that, various customized responses ranging from doing nothing to changing system configurations to shutting down the system. What would be the most appropriate default response was not clear and was assumed to be application specific.

**Model adaptations:** In Figure 17, we show the results of the quantitative user study on the usefulness of the model adaptations. While the answers have a low variance for the usefulness rating of the known stack trace and especially for the generated sequence diagram, the answers for the proposed deployment diagram are more diverse. Both the stack trace and the sequence diagram were rated as useful for investigating the reported vulnerability with an average of 3.4 and 4.2 on a scale of 1 to 5, respectively.

The deployment diagram received an average rating of 2.8. While 23% of the participants rated the deployment diagram as useful for investigating a security vulnerability, 38% of the participants rated it as not useful or undecided. For the stack trace and sequence diagram, the majority of participants rated these representations as useful (59% for the stack trace and 82% for the sequence diagram). While the stack trace was mostly (18 votes) rated with a usefulness of 4, the sequence diagram received almost as many votes for this level of usefulness (13 votes), and with 19 votes even more votes for very useful (rating of 5).

The well-known stack trace is considered useful, but the votes for the sequence diagram are even more positive. Based on the votes alone, we can conclude that the participants in our study have a mixed impression of the shown deployment diagram, but still see some use in it, but perhaps only in special cases. To identify such cases, we next discuss the qualitative answers to the advantages and disadvantages.

The stack trace is often rated as a familiar structure associated with the source code, but does not provide detailed information about the vulnerability. Similarly, the deployment diagram does not provide detailed information, but is rated as an easy entry point suitable for non-technical stakeholders. The sequence diagram may also be appropriate for non-technical stakeholders. Many participants agreed that the sequence diagram provides a detailed description of the security violation, but at the expense of readability for larger violations. The models may require trained personnel for productive use. In summary, many participants commented that the integration of all representations would be best for them. Since we did not tell them that the models were from UMLsecRT and that this suggestion was indeed the case, we see this as further confirmation of the model adaptations supported.

In summary, the participants in our case study rated the sequence diagram, which we use to represent security violations in UMLsecRT, as the best representation for providing details about the detected security violation. However, for a practical application, the integration of all representations seems to be the favorite of the participants, which we already support for the two models. Since the sequence diagram uses methods as messages, it provides the same...
integration with the code as the stack trace, and integration with the stack trace is straightforward.

### 6.5 RQ5–Scalability of the Model Synchronization

For UMLsecRT to be usable at development time, it must integrate with development environments and be usable without interrupting the development process. Two important aspects are the time needed to perform tasks, in this case primarily the synchronization of security annotations between models and code, and whether the models facilitate annotation of the system. Especially for the latter, model size is an essential measure [82]. In this part of the evaluation, we investigate whether the synchronization approach can be applied to real Java projects of different sizes in a reasonable amount of time, and whether reverse engineering produces models with a manageable size for large programs.

#### 6.5.1 Setup

We investigate the scalability of TGG-based synchronization for the two applications discussed (reverse engineering and synchronization). First, we investigate how the reverse engineering of UML class diagrams from Java source code scales in terms of execution time and model size. Second, we investigate the propagation of UMLsec security annotations into the implementation.

Our selection of subject systems is based on previous experiments performed for related approaches [55], [83], [84], [85], [86], as well as on a standard catalog for analyzing the evolution of Java systems [87], to address the research question. We selected 19 open source Java programs from different application domains, including software systems for both software developers and end users. We also aimed to include a range of different program sizes.

First, we applied our proposed synchronization technique to all subject systems to reverse-engineer a UML class diagram (step 1 in Figure 1) while measuring execution times. We then simulated step 2 by randomly injecting UMLsec security annotations into the reverse-engineered UML class diagrams and propagating them into the Java source code (step 3). Again, we measured the execution times of the synchronizations.

#### 6.5.2 Results

Table 5 lists the Java programs used as subject systems along with statistics about their size. These statistics include the logical lines of code (LLOC) of the program’s source code, as well as the number of types, methods, and fields. Types are classes, interfaces and enumerations.

**Reverse engineering:** The last column of Table 5 shows the execution times of the model generation in seconds. Here we show the median of 5 runs. Figure 18 shows the detailed runtimes of the transformations. The total height of each bar is equal to the corresponding runtime in Table 5 and consists of the time spent for parsing the source code and the transformation with TGG. We can see that the transformation is able to extract UML class diagrams for small and medium projects within seconds and scales even for larger programs.

The UML models created are at the granularity of the implementation-level class structure, and additional more abstract models must be extracted manually. The only abstraction from the implementation is the reduction of details from the statement level of methods and fields to dependencies between classes. However, this abstraction significantly reduces the complexity in terms of the dependency types used and the number of dependencies considered. Size is an important aspect with respect to the manual handling of models [82]. For this reason, Figure 19 shows the size of UML models corresponding to software systems of different sizes. To relate these values to other models representing the same software systems, we also show the sizes of the corresponding MoDisco models and a program model (pm) of the GRAViTy tool, which has been designed to represent object-oriented programs at a high level of abstraction [55]. All models seem to grow more or less linearly with the number of lines of code. But while the program model has on average 28% of the number of nodes of the MoDisco model, this ratio is only 11% for the extracted UML models.

In summary, from a model size perspective, UML class diagrams reverse-engineered with our UML TGG are not as advantageous as manually created UML mod-

Table 5: Program statistics and execution times of the UML model creation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>LLOC</th>
<th>types</th>
<th>methods</th>
<th>fields</th>
<th>in s</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickUML [88], [89]</td>
<td>2.0.1</td>
<td>2,667</td>
<td>22</td>
<td>175</td>
<td>156</td>
<td>3.38</td>
</tr>
<tr>
<td>jScａ [90]</td>
<td>2.1.0</td>
<td>5,437</td>
<td>131</td>
<td>563</td>
<td>200</td>
<td>7.47</td>
</tr>
<tr>
<td>jUnit [91]</td>
<td>3.8.2</td>
<td>5,780</td>
<td>188</td>
<td>841</td>
<td>161</td>
<td>6.05</td>
</tr>
<tr>
<td>JSSE–Open[DK] [92], [93]</td>
<td>20,896</td>
<td>236</td>
<td>1,875</td>
<td>861</td>
<td>23.22</td>
<td></td>
</tr>
<tr>
<td>Girnt [94]</td>
<td>11.0.2</td>
<td>21,228</td>
<td>397</td>
<td>3,925</td>
<td>1,323</td>
<td>16.95</td>
</tr>
<tr>
<td>Nutch [95]</td>
<td>0.9</td>
<td>21,473</td>
<td>331</td>
<td>1,750</td>
<td>1,083</td>
<td>16.79</td>
</tr>
<tr>
<td>Lucene [96]</td>
<td>1.4</td>
<td>25,472</td>
<td>333</td>
<td>2,096</td>
<td>1,166</td>
<td>12.98</td>
</tr>
<tr>
<td>log4j [97]</td>
<td>1.2</td>
<td>30,662</td>
<td>459</td>
<td>3,190</td>
<td>1,226</td>
<td>18.83</td>
</tr>
<tr>
<td>HotDraw [98]</td>
<td>7.6</td>
<td>32,434</td>
<td>480</td>
<td>3,781</td>
<td>900</td>
<td>34.47</td>
</tr>
<tr>
<td>PMD [99]</td>
<td>3.9</td>
<td>43,063</td>
<td>620</td>
<td>4,084</td>
<td>1,582</td>
<td>32.17</td>
</tr>
<tr>
<td>Edit [100]</td>
<td>4.0</td>
<td>49,829</td>
<td>606</td>
<td>3,429</td>
<td>1,976</td>
<td>25.29</td>
</tr>
<tr>
<td>Transforms [101]</td>
<td>3.1</td>
<td>71,348</td>
<td>610</td>
<td>1,509</td>
<td>396</td>
<td>23.65</td>
</tr>
<tr>
<td>Trust [107]</td>
<td>21</td>
<td>77,501</td>
<td>946</td>
<td>6,166</td>
<td>3,074</td>
<td>38.68</td>
</tr>
<tr>
<td>JabRef [102]</td>
<td>2.7</td>
<td>77,813</td>
<td>1,371</td>
<td>5,702</td>
<td>3,699</td>
<td>49.96</td>
</tr>
<tr>
<td>Xerces [103]</td>
<td>2.70</td>
<td>102,279</td>
<td>865</td>
<td>8,267</td>
<td>4,676</td>
<td>47.76</td>
</tr>
<tr>
<td>ArgouML [104]</td>
<td>0.198</td>
<td>135,542</td>
<td>1,596</td>
<td>12,401</td>
<td>3,458</td>
<td>78.45</td>
</tr>
<tr>
<td>JFreechart [105]</td>
<td>1.019</td>
<td>144,338</td>
<td>1,093</td>
<td>11,861</td>
<td>3,258</td>
<td>70.74</td>
</tr>
<tr>
<td>Tomcat [81]</td>
<td>6.0.45</td>
<td>177,013</td>
<td>1,732</td>
<td>16,661</td>
<td>7,991</td>
<td>87.52</td>
</tr>
<tr>
<td>Azurues [106]</td>
<td>2.306</td>
<td>201,541</td>
<td>3,432</td>
<td>17,564</td>
<td>7,106</td>
<td>100.47</td>
</tr>
</tbody>
</table>

Fig. 18: Run times for the program model and UML TGGs.
Fig. 19: Relationship between implementation sizes and corresponding model sizes.

Fig. 20: Time needed to propagate a security annotation from the UML model into the implementation.

Synchronization: Figure 20 shows the time taken to propagate a security annotation from the UML models to the implementation as an average of 10 annotations per project. Similar to reverse engineering, the synchronization time increases with project size, but is still a fraction of reverse engineering. While for small to medium projects (< 72k LLOC) it is feasible to synchronize after every change, for larger projects it may be a good idea to propagate multiple annotations at once. For example, it takes 26 seconds to propagate one annotation on Azureus, but only 50 seconds to propagate 10 annotations, and 165 seconds to propagate 50 annotations in one run.

To answer O4, the results show that the time required for initial model extraction is reasonable even for larger programs. Since our implementation supports incremental model synchronization, initialization costs may be omitted later in the case of evolving programs. Synchronization also scales well for small to medium-sized programs. It is still applicable to larger programs, but should then be applied in batches of annotations.

In this evaluation, we have shown that UMLsecRT allows us to effectively monitor Java applications for compliance with design-time security properties and to mitigate security violations as considered in the CWE. Furthermore, we have shown that there is an initial overhead depending on the size of the program, which is relativized over time.

Accordingly, an efficient implementation of UMLsecRT is feasible for long-running programs. Furthermore, developers agree that the UML models generated by UMLsecRT can help to investigate security violations and the countermeasures allow for practically relevant mitigations. In addition, synchronizing security annotations with the implementation allows security experts to work on more abstract and smaller UML models, and even scales for large programs.

6.6 Threats to Validity

Threats to validity include construct, internal, external, and conclusion validity [108]. In this section, we discuss the threats to the validity of the UMLsecRT evaluation.

6.6.1 Construct Validity

In our evaluation of UMLsecRT's effectiveness (O1) in mitigating security violations, we constructed the test cases we used for the evaluation based on the examples from Juliet. Since this dataset contains only static analysis examples and no exploits, the way we constructed the examples may affect the validity of the experiment. However, for all negative test cases, a security violation occurs at runtime when the test case is executed without security monitoring, making them all cases that need to be mitigated.

When constructing the experiment to evaluate the applicability of UMLsecRT to real-world programs (O2), we decided to measure the overhead in security-compliant scenarios and that annotating the benchmarks is not necessary to gain insight into the applicability to real-world programs. First, both decisions do not affect the ability to identify general problems with the monitoring approach, as shown by the identified problems with the instrumentation technologies. Second, since security violations are exceptional cases that must be mitigated under all circumstances, the primary applicability in terms of overhead is mainly relevant to the security-compliant scenario we focused on. For reasons of practicability, we decided not to recreate the DaCapo benchmark with security annotations. This decision has almost no impact on the runtime, as the entire monitoring code is still injected and executed. The only difference is that in our experiment, the injected code is always executed with empty sets of security-critical signatures, which is the most common case even in annotated programs.

The design of our user studies (O4) did not allow participants to interactively apply our approach. The first study was based on generated views of a vulnerability that we selected and presented in a survey. In addition, we asked participants to consider the three views presented independently. In the second study, we presented the countermeasures only in an interactive interview, but allowed participants to ask questions. These design choices for the user may have influenced the participants' responses regarding the usefulness of the UMLsecRT for investigating vulnerabilities and implementing countermeasures.

To study the scalability of UMLsecRT's synchronization between models and code (O5), we had to construct meaningful deltas to propagate from the models to the source code. Here, we decided to consider a scenario where synchronization is performed after each change, which in the application of UMLsecRT is usually the addition of a
security annotation. This choice may affect the validity of the experiment. Nevertheless, we considered a relevant case and outlined the effect of considering multiple annotations at once on an example.

### 6.6.2 Internal Validity

To study the effectiveness of security monitoring approaches (**O1**), we had to implement UMLsec «secure dependency». A potential threat is that we were able to implement it better in UMLsecRT than in other monitoring approaches. To counter this threat, we discussed in detail the challenges of implementing «secure dependency».

To test our assumption that long-running applications can be monitored with low overhead (**O2**), we implemented an experiment in which we monitored the response times of iTrust. The choice of iTrust is based on our own experience and may compromise the validity of these experiments. Nevertheless, this example shows that there are applications that can be monitored with low overhead.

In our experiments on the monitoring overhead (**O3**) we selected a particular method pattern on which we compared the different monitors that might be biased towards UMLsecRT. However, evaluations of other security monitoring approaches such as BeepBeep use similar patterns [31].

To recruit participants for the two user studies (**O4**), we primarily advertised the studies within our own networks, asking them to participate and to forward the invitations. This approach may have influenced the composition of participants in our study. In conducting the user studies, we may have projected our own bias towards UMLsecRT onto the participants. To reduce this bias, we carefully designed the two studies by not mentioning UMLsecRT at all in the quantitative study and by having an open discussion about the industry’s needs in the interviews about countermeasures of UMLsecRT.

### 6.6.3 External Validity

When studying the effectiveness of UMLsecRT in detecting security violations (**O1—Effectiveness**), we may not cover all relevant cases. We used the Juliet test suite as a guide to systematically select relevant vulnerabilities that can be detected by UMLsecRT. While there may be other relevant vulnerabilities that could be detected and mitigated using UMLsecRT, we currently only consider the selected ones as possible. Whether UMLsecRT is suitable for detecting security violations due to other vulnerabilities will be the subject of future work.

Similar to the selected vulnerabilities, the benchmarks and iTrust used to evaluate applicability to real-world applications (**O2**) may not be representative of any software system. To mitigate this threat, we explicitly selected the well-established DaCapo benchmark instead of a custom selection of applications and workloads.

A threat to the validity of our user studies (**O4**) is the limited number of participants, which may limit generalization to other groups of participants. In addition, the composition of the participant groups may not be representative. In the quantitative study, academics are in the majority, but we did not observe differences in responses between academics and practitioners. In the qualitative study, the automotive domain may be overrepresented, which could affect the generalizability of the results. Finally, by participating in a study, participants may have been subject to the Hawthorne effect [109] and may have given answers that they would not have given in a non-study situation. Nevertheless, even assuming a weakened significance, the user studies indicated a good usefulness of the adapted system models for investigating security violations and the usefulness of the countermeasures. We will investigate the usefulness from a user perspective in more detail in future work.

As well the selected applications as the deltas used to evaluate the scalability of UMLsecRT (**O5**) might not be representative. To minimize this threat, based on previous works and the literature, we selected applications of different sizes and from different domains. The annotations themselves, do not allow for much variation and can be considered as relevant changes to be considered in the evaluation. We already investigated structural changes in previous works [18], [107].

### 6.6.4 Conclusion Validity

Regarding **O2—Applicability**, we have successfully demonstrated that it is possible to monitor for vulnerabilities and breaches in real-world Java programs. However, we did not perform the evaluation based on vulnerabilities that have been documented in the wild in real-world applications. Nevertheless, as part of the experiment that addresses **O1**, we have shown that UMLsecRT is capable of detecting real-world security violations in minimal examples. Furthermore, as part of our case studies, we show two exemplary violations that are mitigated on real-world systems.

Based on our two studies, we conclude that both the adaptations and the countermeasures are useful (**O4**). While we provide reasonable indications, it remains to prove these conclusions in statistically significant experiments.

Regarding the scalability of model synchronization (**O5**), we conclude from the measured numbers that synchronization can be practically integrated into real development workflows. While we show the principle suitability, it should be confirmed in an independent user study.

## 7 Case Studies

In the evaluation, we conducted controlled experiments to evaluate the individual contributions of UMLsecRT. Here, we investigate in a practice-oriented context whether UMLsecRT is in principle capable of supporting the development of secure software systems as intended, and identify possible pitfalls. In particular, we focus on UMLsecRT as a whole, from requirements to the execution. Since we are interested in qualitative real-world experiences, case studies provide a suitable means to investigate the objectives [110].

The first case study is the Eclipse Secure Storage of the Eclipse IDE, which has already been used as a running example in this work. The second case study is the electronics health management platform iTrust, which we have already used in the runtime overhead evaluation.

Since the developers of iTrust provide complete documentation and models are available in existing research [2], [35], [45], [111], we use iTrust to demonstrate the feasibility of using UMLsecRT to develop a new software system with security in mind. While Eclipse also provides good

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
documentation of the implementation, there are no requirements or models available. Therefore, we apply UMLsecRT to Eclipse Secure Storage to demonstrate the feasibility of using UMLsecRT for legacy projects. In both cases, we replay actions taken during the development of the two systems while applying UMLsecRT.

7.1 Eclipse Secure Storage

Our first case study focuses on the application of UMLsecRT to a security-critical part of the Eclipse IDE, the Eclipse Secure Storage [46], which is used as a running example in this paper. How exactly Eclipse Secure Storage works is described in the Eclipse help document [46]. However, this description is rather high-level and is complemented by the low-level API documentation. We consider Eclipse Secure Storage to be the perfect case study for investigating the migration of legacy projects to UMLsecRT due to its security-criticality, good documentation, and widespread use in practice.

7.1.1 Reverse Engineering of Models

Since no models exist for Eclipse Secure Storage, the first step was to reverse engineer models to which we can then apply the UMLsec annotations. We automatically reverse-engineered a detailed UML class diagram from the Eclipse Secure Storage source code using UMLsecRT, which took 9.3 seconds and therefore did not cause any major interruptions. This gave us the model from which we presented a critical excerpt in this paper in Figure 2.

One drawback of reverse-engineered models is their size. Initially, we tried to visualize the entire UML model in a single view in Papyrus, but this was not feasible. Since we were already familiar with the structure of the Eclipse Secure Storage from its documentation, and since pure architectural representations are beyond the scope of UMLsecRT, we skipped extracting meaningful views and proceeded directly with the security engineering.

7.1.2 Static Security Specification

One of the two main goals of applying UMLsecRT to legacy projects is to create artifacts that allow for easier specification of security properties compared to specification at the implementation level. To this end, we started by annotating the reverse-engineered model with UMLsec security properties and creating views of the model along the specified security layers. We were guided by CARiSMA’s static checks, which revealed classes with missing security annotations according to UMLsec secure dependency after classifying a security-critical property or operation, facilitating annotating the models and reasoning about security implications. However, since we assumed that the Eclipse Secure Storage was security compliant, we always decided to add the necessary annotations and not to delete the dependencies. We propagated the annotations to the source code whenever we reached a state that was consistent according to the UMLsec secure dependency and before identifying a new element to annotate in the documentation. The propagation was usually done before we identified the next element and therefore did not have a major impact on our workflow.

Technically, we have demonstrated the feasibility of the tools for annotating the models and, in particular, the synchronization mechanism of UMLsecRT for propagating the security requirements to the implementation. From a developer’s point of view, the main difficulties in annotating the models is in the UML editors used. Handling the relatively large UML class diagram is not as fluid as navigating through the Java source files. However, once a suitable view had been created, we found the graphical representation easier to follow than the source code files because of the explicitly visible class-level dependencies.

7.1.3 Runtime Monitoring

Since any installed plugin can access the password store in the Eclipse IDE, and it is not predictable which plugins a developer will install in her Eclipse IDE, we consider a malicious plugin like the one introduced in Section 2.4 as a reasonable example. To conduct this part of the case study, we implemented this malicious plugin that attempts to illegitimately access passwords stored in the Eclipse Secure Storage. We also extended the Eclipse Secure Storage implementation with countermeasures to actively prevent such illegal access. After these two enhancements, we monitored Eclipse with the UMLsecRT agent and executed the malicious plugin.

Since the security annotations had already been propagated to the source code by UMLsecRT, all that remained was to manually launch Eclipse with the UMLsecRT agent attached. To do this, we simply launched the new instance from the Eclipse instance containing our annotated Eclipse Secure Storage project by adding the agent to the Eclipse launch configuration. Upon launching this new instance, UMLsecRT successfully mitigated access to the security layer and generated sequence and deployment diagrams as shown in this paper. Since we used similar diagrams in our evaluation of the usability of the model adaptations, these observations are directly applicable to this case study.

7.2 iTrust Electronics Health Management System

The second case study consists of a Electronic Health Records system developed as a class project over more than ten years [45], [47]. The main documentation is provided in the form of requirements describing use cases for the iTrust system. The software system itself was implemented in Java using Java Server Pages (JSP). In addition, design-time models have been created as part of various research activities [2], [3], [18], [35], [59], [111], [112], [113]. Based on these artifacts, we simulate the implementation of the iTrust system using UMLsecRT from the very beginning, starting with requirements engineering. In all steps we reused the existing iTrust artifacts and simulate their creation while following the UMLsecRT development approach.

7.2.1 Requirements Engineering

Typically, the development of a software system begins with domain analysis as part of requirements engineering [114]. The knowledge of entities and relationships within the system domain is captured in a domain model, which allows the identification of basic security requirements in the domain [115], [116], such as what is sensitive data, such as personal data or medical records. Specifying the intended functionality of the software system based on the domain model is one of the first steps in requirements engineering.
For this purpose, UML provides the notation of use case diagrams, and iTrust uses a publicly available use case-based specification of its requirements [117].

To simulate this phase, we started with a domain model showing basic concepts in a hospital [118], such as doctors treating patients, and annotated it with basic security requirements using UMLsec secure dependency. Then, to simulate requirements engineering, we manually recreated iTrust’s use case diagram based on iTrust’s requirements. We took the domain model as given and refined it by specifying the use case diagram. Whenever there was a refinement relationship between the use case diagram and the domain model, we explicitly modeled that relationship. Since this step does not differ significantly from general domain and requirements modeling, we see it mainly as a preparatory step for the specific tasks of applying UMLsecRT and for getting an impression of the reusability of early security specifications. In the next step, we further refined the domain model and use case diagrams to specify an architecture that allows the implementation of the specified use cases.

### 7.2.2 Software Architecture and Security Modeling

After requirements engineering, the architecture of the software system is specified based on the requirements models and textual requirements. Following the principle of security by design, we must explicitly consider security requirements in this step [8]. Accordingly, in this section we discuss the simulation of the architecture specification for the iTrust system. In doing so, we focus on the effort required for security specification using UMLsec at the level of detail required for effective application of UMLsecRT security monitoring, and how the reuse of early security specifications supports this effort.

Starting from the models developed in requirements engineering, we iteratively refine these models until we arrive at a detailed specification of the iTrust system as it was reverse engineered in previous work [2, 3]. More specifically, we simulated three evolutionary steps:

1. We defined classes to technically represent the roles and actors from the domain model and use case diagram.
2. We added data classes for storing medical information about patients, contact information, appointments, etc.
3. We added classes and operations to implement the functionality, use case by use case.

After each extension step involving the addition of a coherent set of model elements, a security engineering step takes place in which we propagate the security annotations from the requirements models into the architecture and ensure compliance with the UMLsec secure dependency. In doing so, we were provided with a list of security violations detected by the UMLsec checks implemented in CARISMA, which are executed each time new security annotations are specified. While many of the required security annotations could be easily specified in advance, there were cases that we did not immediately recognize, but were reported to us by CARISMA. Here, as intended by the check, we considered whether a security level should be extended to a new class or whether we should reconsider the dependency. However, since the given design of iTrust is required for the subsequent steps of the case study, we fixed all reported security issues by adding the required UMLsec stereotypes.

Overall, there was quite some effort in specifying the architecture and reasoning about all security implications, but the effort did not seem to differ from applications of UMLsec in the industry [36], [37], [38], [39], [40], [41]. Reuse of early security annotations could be facilitated by more tool support, but this is beyond the scope of UMLsecRT.

### 7.2.3 Implementation

After specifying the architecture, we need to implement the software system. Using the synchronization mechanism of UMLsecRT, we generated an early class layout from the architecture. Then we manually filled this layout with functionality. During this step, UMLsecRT kept the model synchronized with the manual implementation changes. Since this results in adding new elements to the model, additional security refinements are required. We performed this manual extension by copying and pasting implementation fragments from iTrust into the generated class layout.

A technical problem we encountered was that the MoDisco parser used is not incremental and creates a completely new source code model each time. This was treated by our synchronization as if all UML elements were discarded and new ones were created reflecting the new MoDisco model. Since the security annotations are also part of the implementation, there is no loss of information except for the refinement relations when using design models of multiple abstractions. To preserve the refinement relations, we simulated changes by manually copying the corresponding changes directly into the MoDisco model. Since incremental parsers exist [119], [120], we see this as a minor and easily solvable implementation challenge.

In summary, we were able to generate an initial code skeleton that is connected to the design-time models through UMLsecRT’s correspondence model. From a user perspective, there was no difference from code generation using other modeling tools such as Enterprise Architect or Rational Architect. In addition, we were able to continuously synchronize the growing source code with the design-time models, with only minor technical issues related to multiple models with different abstractions. As we detailed the implementation based on UMLsec secure dependency, we were notified of methods that were not annotated as expected and therefore needed to undergo a security review to determine if the security requirements were met or if changes were needed. We see this as a systematic way to raise awareness and guide security reviews, as also indicated in the expert interviews.

### 7.2.4 Runtime Monitoring

With the annotated source code, no additional effort is required to run iTrust with security monitoring. To simulate an attack, we assume that the Apache Commons Codec library has been replaced during deployment with a malicious version that attempts to gain access to sensitive information such as billing information. To do this, we implemented a malicious version of this library that takes advantage of being called by a sensitive method deep in the iTrust system, and exploits a missing authentication in the constructor of the SearchUserAction class that we found.

When we started iTrust, we immediately observed a mitigated attack when interacting with the web UI, which
surprised us because no part of the code using the malicious library should have been executed. In the adapted models, we saw that the violation did not come from the malicious library, but from the Java JSP interpreter accessing annotated methods of iTrust. Using the adapted models, we were able to trace this violation to code in the JSP implementation that directly interacts with annotated Java methods. Since this is expected and compliant behavior, but we are unable to annotate code in the JSP files, we decided to extract such code into a method in the Java implementation that serves as secure entry points. In most cases, these extracted code fragments even included an implementation of authenticity before executing a critical method. After this change, iTrust behaved as expected and our malicious library was executed as intended, but access to the billing information was prevented by UMLsecRT.

In summary, we had to investigate an unplanned security violation that we were not aware of beforehand. The model adaptations helped us to identify the cause of the violation. Furthermore, we can say that this was indeed a true positive violation. Both this and the planned violation were successfully mitigated, demonstrating the practical feasibility of UMLsecRT to enforce design-time security requirements for secrecy and integrity in practical scenarios.

In the two case studies, we were able to show that the results of the individual steps described in the previous sections work as expected when applied as a whole to real projects, and thus provide the developer with much-needed support for monitoring compliance with design-time security specifications without the need to explicitly specify monitoring policies. Furthermore, our case studies show that the use of UMLsecRT facilitates systematic security reviews of software systems, thereby relativizing the effort required for security annotations, since these reviews must be performed for security-critical systems anyway.

8 Discussion

In this section, we discuss the assumptions, limitations, and implications of UMLsecRT.

Our main assumption is that systems are developed using model-driven security engineering. The need for design models as a prerequisite for using UMLsecRT may lead to limitations in its applicability. In this regard, we consider two factors that might limit its practical applicability.

First, agile development may not allow for the required model-driven development. However, since Rumpe shows how to apply agile development to model-driven development [121], this should be a minor limitation. Furthermore, our synchronization allows for iterative development of models and code. However, in many safety-critical domains, standards such as ISO/EC 62304 for medical device software development [122] require the development and maintenance of the necessary artifacts. Pure security monitoring is even possible by working directly on the source code, but without many of the benefits of UMLsecRT.

Second, UMLsecRT may not be able to repay the cost of the additional effort required to produce detailed design models. However, these artifacts are likely to be required by standards to which software systems in many security-critical domains must conform. In this case, there is no additional cost to using UMLsecRT. For all other systems, using UMLsecRT may result in additional effort to produce these artifacts. In this case, UMLsecRT’s automated reverse engineering of UML models can be a cost-effective solution. In any case, if developers want to use our approach, they should adopt model-driven development practices.

It may seem that annotating the entire code base would be a huge overhead and might threaten usability. However, large amounts of annotation are widely used in the industry, e.g., in the Spring framework [123] or Jackson [124]. Consistent with these observations, participants in our quantitative user study did not see problems in specifying the information needed in annotations or the number of annotations needed. Furthermore, most of the data we need has already been collected during threat modeling and can not only be reused at low cost, but can even be improved by our approach. The suitability and usefulness of UMLsec for specifying this information has been evaluated in different contexts. For an extension of UMLsec with variability, we conducted a user study that found good applicability even though it was a complex extension of UMLsec [125]. Also, in a public report of the EU project VisiOn [126], the pilots write that they feel able to analyze complex aspects of privacy and security [40]. In a comparison of privacy models by Pierre Dewitte et al., the CARiSMA tool we use scored highest for tool support, indicating good applicability [127].

While our assumptions may limit the applicability of UMLsecRT, when using UMLsecRT, developers implicitly follow other best practices for secure system development. Using UMLsecRT leads developers to implement the principle of security by design and allows them to systematically identify security problems early.

Regarding the performance of our implementation, we measured a monitoring overhead of 4.1x. While we have shown that the overhead for instrumenting the classes becomes less relevant for long-running applications, there is room for improvement in the performance of the checks, which threatens the applicability to real applications. The overhead is mainly the retrieval of the UMLsecRT stack for the current thread, which currently happens twice for each method call, at entry and exit. A possible solution could be to introduce a field in each class that holds the stack, which is easy for single-threaded applications, but complicated when objects are shared between multiple threads. The relevance of the slowdown could be reduced by monitoring only the critical core parts of an application, similar to Bodden et al. [128]. Again, the models used in UMLsecRT could be used to identify these parts.

Another way to implement UMLsecRT is to extend the existing Java annotations to be used with aspects [129]. A disadvantage of this approach is that the monitoring is part of the target program. Also, it may not be possible to control a monitored application in a sophisticated way, since the aspects always run at the application level. Moreover, there is an inherent security risk that an attacker might learn that UMLsecRT aspects are part of the program and use reflection to disable or, worse, corrupt them.

The size of the reverse-engineered UML models may limit the applicability of UMLsecRT. However, using appropriate views on the extracted models, they can be effectively annotated with UMLsec security requirements, as
demonstrated in our case studies. The UML supports the concept of views, which allow the visualization of selected elements of a UML model [61]. A single UML element can be part of multiple views, allowing developers to create views of manually manageable size that focus on specific aspects of the software system, such as a security-critical dependency or a classified class member. Such views can be extracted automatically, e.g., using model slicing [130], [131] or clustering [132], [133]. Given appropriate slicing rules or coupling criteria, both approaches can be used to automatically retrieve all elements relevant to a developer when inspecting a particular UML model element. Based on these elements, an appropriate view can be created.

9 Related Work

In this section, we discuss works that address similar issues as UMLsecRT or provide alternative solutions that could be used by UMLsecRT. We relate UMLsecRT to these works, highlight differences, and focus in particular on the expressiveness of the security properties considered and the type of monitoring used. First, we discuss security policy languages that could be used in UMLsecRT as an alternative to UMLsec, second, general monitoring solutions upon which UMLsecRT’s security monitor could be built, and finally, other security monitoring approaches.

9.1 Security Policies Languages

Similar to UMLsec, other model-based languages such as SecDFD [17] or SecBPMN [16] support the specification and validation of security requirements at design time. For both languages, we have already demonstrated integration with UMLsec [43], [107], in particular with UMLsec’s secure dependency, making it accessible to UMLsecRT as well.

In addition to model-based languages, several textual security policy languages have been proposed. The Security Policy Language (SPL) of Ribeiro et al. [29], allows to define complex constraints for access policies. The SPL allows to specify hierarchies of entity types and to define logical constraints based on a tri-value algebra (“allow”, “deny” and “not apply”). Similarly, ConSpec [30], a formal language for policy specification, allows to specify constraints on states before and after events, e.g., on the values of variables. While it is not possible to express UMLsec secure dependency using ConSpec, it may be possible using SPL, but it is likely to be too coarse-grained. Furthermore, such policies are likely to grow in complexity, making them difficult to manage. To address this issue, Bauer et al. propose a composition mechanism for runtime security policies [134].

Siveroni et al. [135] researched the support of design and verification of secure software systems, emphasizing the early stages of development such as requirements elicitation. The proposed approach realizes static verification of properties and allows reasoning about temporal and general properties of a UML subset, e.g., UML state machines. Formal verification is performed using the SPIN model checker. The approach focuses only on the early stages of software design and thus only on statically verifiable properties.

9.2 Java Monitoring

In addition to the JVM Tool Interface (JVM TI), which is used by Java agents such as UMLsecRT, the Java Platform Debugger Architecture (JPD) provides the Java Debugger Interface (JDI) with a higher level of abstraction [136]. One of the main advantages of this interface is that it allows a clear separation between the application code and the monitoring code by eliminating the need for instrumentation that dynamically writes the monitoring code into the application code. However, execution of the smallest DaCapo case, which took 3106 ms without security monitoring, took about 8 h with security monitoring by a prototype of UMLsecRT based on the JDI, making this interface infeasible.

Another widely used technology for implementing approaches comparable to UMLsecRT is Aspect-Oriented Programming [129]. For Java, frameworks such as AspectJ [137] or Spring AOP [138] could be used for this purpose. However, as described in the discussion section, the security aspects would be part of the implementation, which could compromise the security of the system.

BeepBeep3 [139] allows to model the monitoring using a block diagram like syntax, so called BeepBeep processors and the older BeepBeep uses AspectJ to capture the program trace and to verify them using linear temporal logic. Among others, BeepBeep3 has been used for monitoring security properties [31]. In our evaluation, we have shown in detail why it is not possible or infeasible to express UMLsec Secure Dependency using any of the two BeepBeep versions.

Similar to BeepBeep, Larva [76], [140] is an event-based runtime monitoring approach that captures a sequence of events and analyzes them. A domain-specific language (DSL) allows you to specify which events to capture and what conditions to place on variables and transitions. In our evaluation, we have shown that UMLsec Secure Dependency can be checked using Larva when the security annotations are provided in a different way. However, Larva misses many unknown cases as all methods to check have to be explicitly specified upfront and is vulnerable to CWE486. Unlike other Java monitors, Larva allows the implementation of countermeasures comparable to UMLsecRT.

In general, besides BeepBeep and Larva, monitoring approaches based on languages for specifying what to monitor, such as LOLA [141] or event based monitoring using a temporal logic called HAWK [142], do not seem to be able to handle additional input that provides information about security requirements for specific source code elements. The authors of BeepBeep compare it with the runtime monitor for ConSpec [30], Java-MOP [143], and Java-MaC [144], which provide similar expressiveness. They show a slowdown for Larva and Java-MOP around a factor of 6 and 3.5 for the other two monitors for the simplest case, but are not applicable to the other cases. Since the idea of UMLsecRT is to avoid developers having to specify security monitoring policies by simply following the security-by-design approach of UMLsec, there is no need for a security monitoring approach that provides a DSL for specifying custom monitoring policies. Given the huge overhead of the monitoring tools discussed, this argues for implementing and optimizing the security monitor at a lower level, as we have done for UMLsecRT.
To address the inherent problem of high monitoring overhead and to reduce the time required for runtime verification of large programs, Bodden et al. distribute parts of the runtime verification among a large number of users [128]. Instead of instrumenting the entire product, only a part of the program is instrumented at a time. Regular expressions are used to specify traces of unwanted behavior. The authors implemented two variants, but still found a generally high instrumentation overhead.

9.3 Security Monitoring

A widely studied category of security verification is taint checking. Typically, data entering the system is considered as tainted and is not allowed to flow into sensitive sinks [145]. Since taint analysis is a statically undecidable problem, several approaches for runtime taint checking have been proposed [146], [147], [148]. As part of the programming language, it is declared what is tainted data, and the underlying data structure is extended with a flag that can be traced at runtime. Overall, the property considered in taint checking is very close to UMLsec secure dependency, but secure dependency considers control flow dependencies and focuses on explicitly declaring security properties of data. Since runtime taint checking requires significant changes to programming languages and the execution environment, we are not aware of any approach that is widely used in practice. Also, the problem of declaring what is tainted data and what is sensitive data remains. In contrast, UMLsecRT builds on established technologies and provides automated configuration of the runtime monitor based on established design-time security engineering approaches.

Lee et al. focused on inter-app communication in Android, which can allow an attacker to inject arbitrary activities [149]. Finally, user interaction can be hijacked to break the Android sandbox mechanism. Therefore, they propose a static analysis tool that uses the operational semantics of the activity lifecycle to reveal potential vulnerabilities. In contrast, UMLsecRT aims to provide the developer with a lightweight model extension to cover security risks in early design phases, coupled with source code and runtime. Our focus is on making security requirements available at an abstract level such as the system model, as well as at the implementation level to promote general awareness of specific security requirements.

Ion et al. [150] examined the security policy architecture of J2ME (Java for mobile devices), which, unlike Java Standard Edition, does not provide an extensible security architecture. They modified the J2ME VM to handle custom security policies at runtime without significant overhead. For security policy specification, they use the SPL [29] discussed above. In contrast, UMLsecRT uses a Java agent and therefore does not require any changes to the VM. By incorporating model-based design, we help developers gain additional knowledge about the behavior of runtime code.

Costa et al. present a more fine-grained and flexible policy-based security mechanism for J2ME and implement it in two variants [151]. First, similar to [150], by adapting the J2ME VM, facing the problem that keywords in policies are limited to methods that can be intercepted at fixed enforcement points. Second, based on byte code manipulation before and after each call to the J2ME API. They found a performance overhead of less than 5, while we achieve a similar or even lower overhead in the long run, supporting full Java and monitoring all accesses.

Hiet et al. propose to secure Java web applications by monitoring information flows [152]. They extend Blare, an OS-level intrusion detection tool, to implement policy-based intrusion detection by tracing inter-method flows in Java applications, supported by the JRE calling the Security Manager before each I/O access. They encountered a slowdown by factor 12 for loading and factor 4 for execution. Blare requires a modified Linux kernel to run, while JBlare requires a modified JRE, which are severe assumptions against the target environment. Responding to or preventing violations, as well as round-trip engineering, is not discussed.

Staicu et al. conducted a large-scale study of 235,850 Node.js applications and identified two APIs that give direct system access [153]. They address this problem by first building templates for all values passed to these APIs, and then synthesizing a runtime policy to support monitoring, which is integrated into the code through code rewriting. They also support design time through static checks.

Ognawala et al. propose a mixture of concrete and symbolic execution to detect non-trivial vulnerabilities [154]. They allow users to interactively examine calls and evaluate possible vulnerabilities on a graphical representation. While they focus on different types of vulnerabilities than we do, they also conclude that an interactive, graphical vulnerability report helps developers prioritize remediation activities.

10 Conclusion and Outlook

In this paper, we introduce UMLsecRT for propagating model-based security planning to the code level, reducing the effort required to annotate the code base and supporting round-trip engineering by providing feedback on runtime observations back to the models.

UMLsecRT supports reverse engineering of models from source code and synchronization of security annotations in models and source code. Response to detected security issues is supported passively through call trace logging or actively by providing modified return values to protect sensitive application data. Round-trip engineering is supported by feeding additional elements monitored during execution back into the model and automatically generating attack sequence diagrams to help developers investigate attacks with graphical support and relationships to design-time models. This also addresses system evolution detection.

We introduced UMLsecRT by implementing support for dynamic checking of secure call dependencies with respect to UMLsec secure dependency, which previously could only be checked statically (and thus partially). UMLsecRT is supported by a prototypical implementation that realizes support at the source code level by providing Java security annotations, runtime monitoring is provided by the UMLsecRT Java agent, while model and code synchronization is realized using triple graph grammars.

We have successfully applied our approach to the Eclipse secure repository and iTrust. We evaluated UMLsecRT in terms of effectiveness and applicability against real CWEs and the DaCapo benchmark, and in terms of usability in
two user studies. The results show that UMLsecRT can be used in realistic application scenarios.

Future work will primarily focus on a more efficient implementation to reduce the current monitoring overhead and thus increase the applicability in real-world environments. In addition, we aim to evaluate the ability of including additional security properties and evaluating off-the-shelf applications with real-world security issues. We will also investigate the applicability of UMLsecRT to other domains, such as safety or real-time processing guarantees.

ACKNOWLEDGEMENTS

The work presented in this article is part of the Ph.D. theses of Sven Peldszus [107] and Jens Bürger [3]. This work has been supported by the German Federal Ministry of Education and Research (BMBF) in the project AI-NET-PROTECT, and the German Research Foundation (DFG) in the project TraceSEC (project number 500462081).

REFERENCES


Sven Peldszus received a Ph.D. from the University of Koblenz-Landau for his dissertation on Security Compliance in Model-Driven Development of Software Systems in the Presence of Long-Term Evolution and Variants and was awarded the CAST/GI Dissertation Prize for the best German IT security dissertation. His research interests include continuous tracing and verification of non-functional requirements and model-based quality analysis for critical domains such as autonomous vehicles. He is currently a postdoctoral researcher at the Ruhr University Bochum.

Jens Bürger wrote his dissertation on the topic of recovering security in model-based software engineering by context-driven co-evolution. He received a Ph.D. degree from the University of Koblenz-Landau. Currently, he is working as an IT Consultant at the Conciso GmbH in Dortmund, Germany.

Jan Jürjens is a professor for software engineering at the University of Koblenz and director research projects at the Fraunhofer Institute for Software and Systems Engineering (ISST) in Dortmund. He is the author of the book “Secure Systems Development with UML” (Springer 2005, Chinese translation in 2009).