# A Continuous Consistency-Aware Process for the Analysis of Timing Channels

Fatma Chebbi[1], Martin Armbruster[1], Christopher Gerking[1] and Anne Koziolek[1]

[1]Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany

**Abstract**

The protection of sensitive information in software systems has always been of considerable relevance, especially with the emergence of regulations such as the GDPR[1]. However, the overhead engendered by security verification steps can lead to neglecting them to avoid longer releases and higher costs. Agile development processes that are required to respond to changes and prioritize the provision of working software can lead to a reduced application of quality assurance steps. In this vision paper, we concretize a continuous consistency-aware quality management proposal within a security subarea. We focus on a quantitative verification of timing channels. Accordingly, our goal is to support developers in assessing the system's security throughout the software system lifecycle and making necessary trade-offs while limiting the additional overhead. Although parts of the process were easier to design, some points remained open to discussion and will need to be resolved in future work.

**Keywords**

Continuous Software Development, Model Consistency Preservation, Quantitative Timing Channel Analysis

## 1. Introduction

Different security mechanisms protect the confidentiality of sensitive data within software systems. Nevertheless, confidential information can still be deduced through side channels [1]. Taking into account the implicit nature of side channels, they can easily be overlooked by developers, especially when only focusing on the explicit architecture specification. In this way, the system behavior deducible from timing channels can be exploited by outside observers, especially those who have physical access to the software.

To mitigate that threat, different properties have been defined in the research. For confidentiality preservation, the property of *non-interference* describes executions by different users which do not affect one another so as to ensure that information flows only where permitted [2]. However, the fulfillment of a property like *non-interference* can be unrealistic in certain use cases [3]. Hence, in such cases, developers may gain the needed flexibility to take design decisions if they use the quantification of information leakage resulting from a given side channel. Unfortunately, current approaches to leakage quantification are mainly theoretical and so cannot be integrated into agile processes.

In this vision paper, we provide a description of a timing channel analysis-aware incremental process which allows the semi-automated consistency preservation of artifacts. To support incremental changes and consistency preservation in agile development processes, we adapt a model-driven quantitative information flow analysis process [4][1]. Our work is a concretization *for security analysis* of the outline for quality attribute analysis during incremental processes presented in Armbruster *et al.* [5]. The advantage of the Armbruster *et al.* [5] outline is that it considers all quality criteria and so is, by design,

[1]https://gdpr-info.eu/, accessed on 2026-02-13

*Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29-July 3, 2026*

fatma.chebbi@kit.edu (F. Chebbi); martin.armbruster@kit.edu (M. Armbruster); christopher.gerking@kit.edu (C. Gerking); koziolek@kit.edu (A. Koziolek)

https://mcse.kastel.kit.edu/staff_fatma_chebbi.php (F. Chebbi); https://mcse.kastel.kit.edu/staff_martin_armbruster.php (M. Armbruster); https://dsis.kastel.kit.edu/staff_christopher_gerking.php (C. Gerking); https://mcse.kastel.kit.edu/staff_anne_koziolek.php (A. Koziolek)

0009-0008-7177-5213 (F. Chebbi); 0000-0002-2554-4501 (M. Armbruster); 0000-0001-5531-9607 (C. Gerking); 0000-0002-1593-3394 (A. Koziolek)

© 2026 Copyright © 2026 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

[1]The paper has already been accepted, and the publication process is in progress.

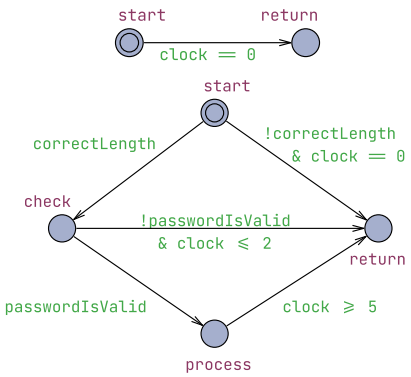
CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-26

CEUR-WS.org/Vol-4338/AgileMDE-paper2.pdf

```java
protected boolean hasCorrectLength() {
    return this.length == 5;}
```

```java
protected void check() {
    if (!password.hasCorrectLength()) {
        return;
    }
    if (password.isValid()) {
        process();
    }
    return;}
```

(a) Simplified code snippet




(b) Corresponding timed automaton

**Figure 1:** Example artifacts for password verification

also applicable to security criteria, as well. In fact, Armbruster *et al.* [5] have already been concretized successfully for performance analysis by the Continuous Integration of architectural Performance Models (CIPM) [6] approach. Moreover, this CIPM comprises implementations of consistency preserving transformations that can be partially reused in our work.

We provide a description for timing channel analysis which allows decomposing the timing channel analysis artifacts so as to enable in each increment a smaller transformation overhead for consistency preservation. We also address the composability of timing channel quantification results so that the analysis can be conducted exclusively for the system parts that are affected by the change.

## 2. Running Example: Timing Channel During Password Verification

The example we consider in this paper is a simplified scenario for password verification with a timing channel caused by early returns. Its code is the *check* method shown in Figure 1a. With knowledge of the code, an attacker can use the execution duration to first guess the length of the password, also shown in Figure 1a, and then the password itself. While the timed behavior is mainly implicit in the code, an explicit representation can be more suitable to verify whether the developed system presents such channels. A possible way to do so is, as shown in Figure 1b, to use timed automata [7], which we represent for simplicity as state machines with real-valued variables. These variables are also known as clocks and model the time elapsed during the execution of the program by means of guards in the transitions or invariants in the states. While the upper automaton in Figure 1b represents the behavior of the *hasCorrectLength* method, the lower one represents the behavior of the *check* method.

With such a behavioral model, the developers can be supported in two ways. On one hand, they can validate implications of their changes on the timing channels by checking the corresponding timed automaton before the release. On the other hand, in case of non-tolerable timing channels, developers can insert possible solutions in the timed automaton, e.g., by adding delays, so that the timing channels are no longer present and then change the code accordingly. However, in an agile development process, keeping all these artifacts up-to-date and consistent among each other, especially in shorter lifecycles, can be challenging. Hence, automating the consistency preservation can be beneficial in such a scenario.

## 3. Background

In this section, we introduce fundamental concepts for our approach.

### 3.1. Quantitative Information Flow Analysis

Confidentiality is one of the main protection goals expressing that sensitive data is only disclosed to authorized parties. To this end, mechanisms such as encryption and access control were implemented.

However, such measures can be bypassed in the presence of side channels, such as timing channels resulting in information leakage. Thus, in the context of information flow analysis, system properties, such as *non-interference*, were defined to ensure the independence of observable behavior of the system from the secrets hidden in the system [1].

Despite works on the fulfillment of the *non-interference* property, researchers are aware of the occasional impossibility of a perfect adherence to it. A more flexible possibility is to quantify the resulting leakage if *non-interference* happens to be violated. Smith [3] has introduced a set of information theory-based metrics, among others, *information leakage*, expressing the reduction of the attacker's uncertainty about sensitive information after observing the system's behavior. This metric can be calculated on the basis of different types of *entropies*. The suitable type of the latter can be selected depending on a specific attack scenario.

### 3.2. A Process for Timing Channel Analysis in Software Systems

In a previous doctoral symposium paper [4], we presented a process to bridge the gap between the theoretical quantification of information flow [3] and an existing practical approach for *non-interference* verification [8]. Similar to our work, Gerking *et al.* [8] focus on timing channels as a specific category of side channels. Accordingly, they use timed automata to model the system's behavior. Since this work presents a qualitative analysis of the *non-interference* property, we intend to extend it in a way that the security violations detected through the analysis can be quantified.

The approach we suggested previously [4] consists of a batch process, composed of two subsequent phases: A *construction phase* and an *analytical phase*. In the first phase, behavioral models are automatically derived from existing artifacts of the software system through model transformations. An example of these artifacts is the existing source code. To accurately capture the system's behavior in a timed automaton, we need to ascertain the implicit information about execution duration of the different parts of the program. To obtain this information, we extract a Palladio Component Model (PCM) [9], an architectural performance model, and use the results of its performance simulation. To achieve the PCM extraction, we reuse the CIPM approach by Mazkatli *et al.* [6].

In the construction phase, a further preprocessing step of the models prepares them for the analysis: We construct a network of timed automata whose interaction is necessary to detect possible *non-interference* violations. This part is also conducted through model transformations.

When all models are present, we move to the analytical part, where the defined security property (e.g., *non-interference*) is analyzed based on the provided models. Thus, we define different model properties, which should be verified through model checking. Moreover, metrics should be defined and compared to common information theory-based ones, such as *information leakage* introduced in Section 3.1. This should enable developers to assess the impact of existing leakages on the system's security.

### 3.3. A Process for Continuous Consistency-Aware Quality Management

The CIPM approach [6] targets performance and defines a DevOps pipeline to enable accurate architecture-based performance predictions. In its current realization, CIPM builds upon the extended Java Model Parser and Printer (JaMoPP) [10] as intermediate Java model and the PCM [9].

The CIPM approach also serves as foundation for a general process for continuous consistency-aware quality management was proposed which integrates into iterative software development paper [5]. This general process aims to maintain the consistency between artifacts in a version control repository and models for architecture-based quality predictions, enabling continuous analyses of the quality attributes of a software system. *The process requires a specialization for concrete attributes* because it is defined independently of concrete quality attributes. The idea is that (based on the changes in commits within a version control repository) a Continuous Integration pipeline transforms software artifacts to

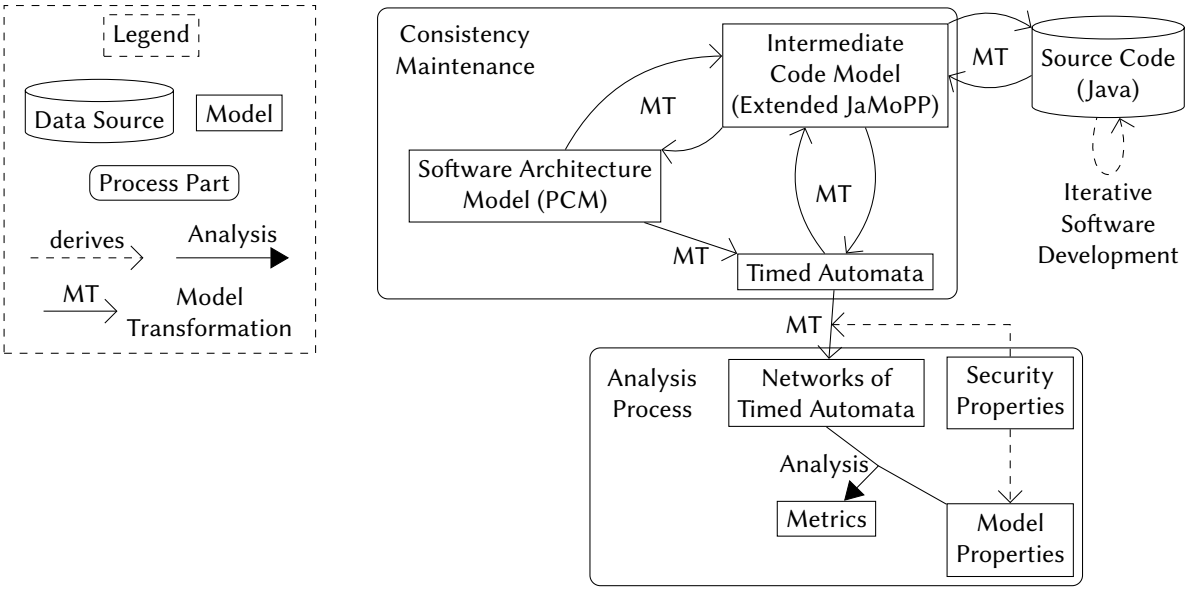



**Figure 2:** Our proposed incremental process for the timing channel analysis.

intermediate models, which are kept consistent to a software architecture model and related models for analyses through change-driven, incremental model transformations, corresponding to reverse engineering. The process also allows forward engineering: the software architecture and related models can be changed (e.g., to incorporate analysis results), triggering consistency maintenance to the artifacts in the version control repository.

# 4. Process Model

In Section 4.1, we provide a brief overview of the process we propose. Then, we focus on two important aspects of the process. In Section 4.2, we present our thoughts for the integration of incremental steps allowing consistency preservation in shorter change cycles. Subsequently, as they can be more challenging than other types of consistency preservation, we discuss the propagation of changes from architectural or behavioral artifacts to the code in Section 4.3.

## 4.1. Incremental Process

The incremental process we propose is depicted in Figure 2. Similarly to the former version [4], our process comprises different models, each serving a specific purpose. In what follows, we present a preliminary list of these models, which can be extended during process improvement steps.

- *Source Code:* We consider the executable code that can, for instance, be retrieved from a repository to be also a model of the software system under analysis. Even though we chose Java as a concrete programming language, other languages should be similarly supported as described in this paper.
- *Intermediate Code Model:* To enable the automated transformations between the different models, we parse the Java code to an intermediate model using the extended JaMoPP [10], which also includes a definition of a Java metamodel.
- *Palladio Component Model:* In our process [4], the PCM serves as a means to estimate the implicit timed semantics in the code. An example of this information is the fact that the second return in our running example described in Figure 1 is reached after a maximum of 5 time units.
- *Timed Behavioral Model:* In order to conduct the timing channel analysis, an explicit description of the timed behavior is needed. A possible formalism to describe this behavior is timed automata. These automata can be derived by means of a model transformation relying on the timed semantics

found in the code similar to the work of Liva *et al.* [11] enhanced with the implicit semantics obtained from the PCM simulations. In addition, for the timing channel analysis, these automata should be transformed into a network of timed automata containing auxiliary automata that are necessary for the verification process similarly to the work of Gerking *et al.* [8]. Subsequently, properties of the network of timed automata will be verified and corresponding metrics will be derived through statistical model checking [12].

Ideally, every change to any of these models should lead to an update of all other related models through a consistency preservation mechanism. To this end, we apply the Vitruvius approach [13] ensuring a consistent state between different views, i.e., models. For the transformations between the code, the intermediate model, and the PCM, the CIPM approach [6] already provides implementations that also reuse the concepts of the Vitruvius approach. For example, the transformation from code to the intermediate model, as well as the transformation from the intermediate model to the PCM model are already completely implemented. Transformations from these models to the code were partially implemented [14]. We reuse their provided implementation for our process and extend it as needed. Similar to the CIPM implementation, we intend to implement the consistency preservation steps between the timed automata, the PCM, and the intermediate model.

## 4.2. Definition of Increments

In an agile process, changes to software systems are frequent. If consistency preservation consists of transforming the entire model, performance issues can arise. To remedy this problem, similar to the CIPM approach, we aim to use delta-based consistency preservation mechanisms. As mentioned previously, the generation of the PCM model is almost fully implemented in the CIPM approach and can be integrated into our process. However, as described in our previous work [4], the security analysis part, i.e., the construction of the behavioral models, did not consider such increments. The way the process was conceived in that work, small changes in the *hasCorrectLength()* method shown in Figure 1a will result in reconstructing each timed automaton describing a related or unrelated part of the system, e.g., the automaton corresponding to the methods *isValid()* and *process()*, as well as the reconstruction of their Network of Timed Automata (NTA)s that are necessary for the security analysis. The decision of triggering all the transformations by each change was the result of a trade-off we had to make between transformation code complexity and transformation overhead. In fact, in large programs with a low change rate, using state-based transformations can be easier to implement than delta-based ones as not every change operation type needs to be considered.

This choice was reasonable, as the initial work [4] was not based on the assumption of recurring cycles. As a result, the overhead caused per iteration was not as relevant as in the current incremental process. Thus, we implemented a prototype constructing timed automata from Java code using the simpler state-based transformation language QVTo[2]. For our current work, the number of executed transformations per iteration should be reduced. This can be achieved by only reconstructing the parts of the models, e.g., automata, that were affected by the change. For illustration, in Figure 1 we can see the methods *hasCorrectLength()* and *check()* as two different blocks that are analyzed independently. Changes to *hasCorrectLength()* would trigger a reconstruction of the corresponding automaton, the one corresponding to the *check()* method and their NTAs. Yet, changes to the *check()* method should not trigger changes to any automaton corresponding to *hasCorrectLength()*. This dependence between methods can be determined on the code model using change impact analysis techniques.

## 4.3. Model-to-Code Transformation

In addition to supporting the propagation of changes from code to model, it is also crucial to consider the different roles present in the development process (e.g., developers, architects, and security experts). To avoid overburdening developers in the development process, we intend to allow change propagation

[2] https://wiki.eclipse.org/QVTo/, accessed on 2026-02-10

from architectural artifacts (PCM and timed automata) to the code. Hence, we allow architects to give their input without causing high manual overhead to adjust the system. For example, by adding a delay in the upper left transition in the timed automaton corresponding to the *check()* method (see Figure 1b), the consistency preservation mechanism we use can lead to adding a *wait()* call before line 4 of the method (see Figure 1a), so that the timed behavior of the code is consistent with the model. Another alternative would be to add the *wait()* method call within the *hasCorrectLength()* method. Furthermore, to obtain the new timed behavior, adding the *wait()* method is not the only possibility. Being on a higher abstraction level than the code, changes to the models may lead to multiple implementation possibilities and make propagation to the code less straightforward. To address this, we intend to involve LLMs in the code generation part. To avoid hallucination or unsuitable additions, the developer needs to confirm the changes suggested by the tool before applying them permanently. We discuss this point further in Section 5.2.

#### 4.3.1. Changes to the Palladio Component Model

The propagation of changes from the PCM model to the code model and subsequent code generation is a work in progress in the CIPM approach [14]. Thus, once this part is fully realized, we intend to integrate it into our process. After the intermediate model is adjusted to the PCM changes, a verification should be conducted to check whether the structural changes to the intermediate model affect the timed automata of the software. In this case, they need to be propagated from the intermediate model to the timed automaton. Moreover, since PCM serves as an input for the timed automaton, changes in its artifacts should additionally trigger a new simulation and measurement of runtime information that will be forwarded to the timed automata, which will need to be adjusted accordingly. Subsequently, the analysis steps will be conducted again for the affected automata.

#### 4.3.2. Changes to the Timed Automata

By enabling the propagation of changes from timed automata to the other artifacts, security experts can evaluate different behavioral alternatives modeled with timed automata with respect to security and integrate them into the software system.

For this, the workflow can be divided into two phases:

**Evaluation Phase:** In this phase, the security expert can modify an existing timed automaton and trigger the transformations to construct the NTA for the analysis. For each variant, the metrics can be compared with each other, and then the expert can decide which variant to apply to the system.

**Selection Phase:** Once the security expert has decided which alternative to choose for the system, they save the updates to the timed automaton permanently. The corresponding network of timed automata being already constructed for the evaluation will also be saved. Furthermore, the rest of the artifacts should be updated with the least possible involvement of developers. Similarly to the case of PCM, changes should first be propagated to the intermediate model. In this way, we can make use of the implementation provided by the CIPM approach to update the PCM once the intermediate model is updated.

### 4.4. Composability of Analysis Results

Here, we assume that a system is composed of a set of components linked through their interfaces. In order to support the scalability of the analysis, we intend to analyze each component of the system independently and combine the results mathematically. Accordingly, the analysis does not need to be reconducted for the whole system in case a modification is limited to one part of it. To this end, we assume that a system is composed of $n$ components from the components set $\mathscr{C}$, with $\mathscr{C} = \{C_0, ..., C_n\}$, and a set of interfaces $I_i = \{I_i^0, I_i^1, ...\}$ is defined for each component $C_i, i \in \mathbb{N}$. For each pair of interfaces

$(I_i, I_j), i, j \in \mathbb{N}$, we define a relation $\mathscr{R} : I \times I \rightarrow \mathrm{R}$, with R denoting the set of relationship types between two interfaces. In this work, we define three types of possible relationships $\mathrm{R} = \{\nmid, \|, |\}$. Here, $\nmid$ corresponds to interfaces having methods that are executed in sequential order also known as cascade [15], i.e., the output of one method is a subset of the input of the other, $\|$ corresponds to interfaces that are executed independently from each other, and $|$ to interfaces with methods where the order of execution is not dependent on each other as in the case of $\|$ but having the same input. The last two types of relations are inspired by the definition presented in the work of Kawamoto *et al.* [16]. Finally, assuming the existence of a set of functions $F$ combining the analysis results of two components, where function $F \ni f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, we define an application $G : \mathrm{R} \times (\mathscr{C} \times \mathscr{C}) \rightarrow F$ defining the suitable function for composing the analysis results of the two input components based on the type of relationship between each type of interface of the components. The function should be oriented on theoretical findings such as the work of Kawamoto *et al.* [16].

## 5. Discussion

During the conception of the process, a number of technical and conceptual points to be discussed emerged. These are as follows.

### 5.1. The Incremental Transformation for Timed Automata

As seen in Section 4.2, delta-based transformations seem to be the intuitive way to implement transformations in an incremental process. However, in some cases, it can also be reasonable to use a state-based transformation language such as QVTo. This type of transformation language allows for simpler code artifacts. For smaller models, where changes do not need to be tracked and updated frequently, delta-based transformations do not present a bigger advantage. Accordingly, in the case of the construction of timed automata, the better alternative cannot be easily selected. We thus plan to investigate a middle ground in order to profit from the advantages of both types of transformation language. This will consist of trying to alleviate the resulting code complexity by implementing a hybrid transformation, consisting of a delta-based transformation similar to the CIPM approach, which would then trigger state-based transformations in complex parts of the system. As a fallback solution, we consider implementing the timed automata transformation based on a delta-based language, e.g., Reactions employed in Vitruvius [13].

### 5.2. Propagation of Changes from the Model to the Code

Model transformations from the code to the intermediate model and then to the PCM and timed automata tend to abstract certain details of the code. For instance, to keep the complexity of timed automata as manageable as possible for the further analysis steps, code blocks that do not present control flow variations can be merged in one block that is represented as a unit. This simplification leads to the impossibility of an exact recovery of the code parts that were represented in the model. As a result, the propagation of the change to the code will be presented as a suggestion for the developers that will need to be examined and validated manually by them. A next step would consist of the integration of Large Language Models into the generation of code lines, based on the available code and architectural artifacts. Sadik *et al.* [17] investigated the generation of deployable artifacts from UML-based models enriched with OCL constraints using GPT-4. Later, Ahmed *et al.* [18] have even introduced VeriGen: an approach for an LLM-based generation of artifacts in the different stages of software development enhanced with formal verification. However, the generated code represents only a skeleton and thus human involvement is also needed with their approach. Similarly, for our work, we still need to define to what extent human actions need to be involved in code generation and to what extent we can rely on LLMs.

### 5.3. Scalability of Analysis

As shown in the running example (see Figure 1), the models used for the analysis capture the system's behavior even at the statement level. Within bigger systems, the behavioral models can become arbitrarily complex, and thus performance issues can emerge. To counteract this issue, we intend to consider the system as a composition of different modules on two levels: the method level and the component level.

The method level was discussed in Section 4.2; we intend to allow incremental changes, where modifications applied within a method only lead to the modification of the corresponding automaton, those of methods depending on it, and their respective NTAs. Subsequently, these results should only lead to conducting the analysis on the modified automata. The values of the different methods can then be aggregated to derive interface-related metrics.

On the component level, we assume that a component can use an arbitrary number of interfaces. To compose the analysis results, we associate metrics with each interface, e.g., by extracting the maximal quantification from its methods, and combine them as suggested in Section 4.4. Accordingly, we improve the scalability of our process.

The set of possible relationship types discussed in Section 4.4 is preliminary and should be extended in the future with additional possibilities. Furthermore, the corresponding combination functions are yet to be studied and developed. To this end, we intend to define the combining functions $F$ (see Section 4.4) based on existing theoretical works [16]. Moreover, as an attempt to improve the scalability of our approach, we intend to test different granularities of the representation of the behavioral model. For instance, the influence of omitting certain types of statements from the timed automata.

## 6. Related Works

In addition to information flow quantification, techniques such as declassification [19] were introduced to relax information flow policies by releasing sensitive information flow when needed for the system's functionality [19]. However, while declassification is limited to the specified dimensions, the quantification results can be mapped to a continuous interval of leakage values. As a result, the former can be more restrictive than quantifying leakages.

Combining model-driven techniques with security assurance and verification has interested several researchers. For instance, Ben Said *et al.* [20] conducted a composable qualitative automatic verification of *non-interference*. Accordingly, they allowed the assessment of their system's security in the early stages of software development. In a similar work, Stenzel *et al.* [21] presented IFlow, a work using model-driven techniques in which formal models are generated from UML models allowing the verification of information flow properties for mobile apps and web services. Similarly, Sun *et al.* [22] suggested an approach to verify information flow properties by constructing interface automata from structural and behavioral models. These works focus on security analysis in the design phase and without specifying its integration into the development process, which is one of the main issues that we consider in our paper.

Khakpour *et al.* [23] considered incremental security analysis for dynamic software architecture, e.g., self-adaptive systems. They performed a quantitative security analysis based on probabilistic model checking. To this end, they analyzed an attacker's model, possible attack scenarios, and a security-aware system model comprising information such as component vulnerabilities. In contrast to their work, we intend to cover vulnerabilities that are not known to architects.

## 7. Evaluation

To evaluate our approach, we need projects coded in Java and including PCM models and timed automata as artifacts. However, looking into available open source projects on GitHub, we could not find any projects containing both types of artifact. Accordingly, we intend to look for projects implemented

with Java and either containing PCM artifacts or Uppaal timed automata describing their behavior. This condition can be fulfilled by different examples such as CoronaWarnApp[3], an evaluation scenario comprising manually generated PCM models, TeaStore[4], or Teammates[5]. Both Teammates and TeaStore were used as evaluation cases within the CIPM approach and thus, corresponding PCM models can be found in their repository[6].

A remaining problem in this case is, however, the unavailability of corresponding Uppaal timed automata for these examples. Accordingly, we intend to formalize the transformation of Java model elements and PCM model elements to the timed automata. In both cases, we intend to formalize a state-based transformation ensuring the mapping of model elements to timed automata. For formalization, we intend to use a similar relationship definition, such as the consistency preservation rules defined by Klare *et al.* [13]. In the case of Java model elements, we proceed as we did for the implementation of the state-based prototype to generate timed automata from source code. For this reason, we followed a similar approach to the semantics defined by Liva *et al.* [11]. The implementation of the formalized state-based transformation can at a later point be validated by means of tests and smaller examples. Subsequently, we use these transformations to generate timed automata for the evaluation example. Then, through a series of modifications, the state of the projects should be altered and then, through the CIPM approach and the implemented state-based transformation, we extract the corresponding timed automata. The obtained artifacts can then be considered as the gold standard against which we want to evaluate our incremental approach. To check the correctness of our results, we split the series of changes into smaller increments, which we perform gradually along with the delta-based transformations implemented in our approach to ensure the consistency of artifacts. Once all iterations are accomplished, we can compare the results obtained with our reference model. It is noteworthy that the number of iterations can vary between 1 and an arbitrarily larger number as long as it is technically feasible.

Since one of the main goals of this paper is to optimize the performance of transformations for an agile process, we also intend to compare the effort needed for transformation and security analysis using the previous process to the one using the incremental process in long and short software lifecycles. One possible metric that we intend to use in this comparison is the duration of the transformation and the analysis on the same artifacts. Further, we also intend to evaluate the scalability of our approach by considering different change sizes (from atomic changes to complete model generation).

## 8. Conclusion

To facilitate the integration of security-related activities into an agile software development process, we (1) applied an incremental consistency-aware development process [5] and also (2) integrated a model-driven information-flow analysis approach, previously introduced as a batch process [4]. We are confident in the feasibility of the process, and for future work, we are already planning implementation as outlined here in this vision paper.

## Acknowledgments

This work was funded by the Topic Engineering Secure Systems of the Helmholtz Association (HGF) and supported by KASTEL Security Research Labs, Karlsruhe and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - SFB 1608 - 50179826. This research was edited by our textician, Daniel Shea.

[3] https://github.com/abunai-dev/EvaluationScenario-CoronaWarnApp, accessed on 2026-05-12
[4] https://github.com/DescartesResearch/TeaStore, accessed on 2026-05-12
[5] https://github.com/TEAMMATES/teammates, accessed on 2026-05-12
[6] https://github.com/CIPM-tools/CIPM, accessed on 2026-05-12

## Declaration on Generative AI

During the preparation of this work, the authors used Writefully, LanguageTool, and GitHub Copilot in order to: Grammar and spelling check. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

## References


[1] A. Sabelfeld, A. C. Myers, Language-based information-flow security, IEEE Journal on selected areas in communications 21 (2003) 5–19.

[2] J. A. Goguen, J. Meseguer, Security policies and security models, in: 1982 IEEE Symposium on Security and Privacy, IEEE, 1982, pp. 11–11.

[3] G. Smith, On the foundations of quantitative information flow, in: International Conference on Foundations of Software Science and Computational Structures, Springer, 2009, pp. 288–302.

[4] F. Chebbi, Information flow-based quantification of timing-based software systems, in: 2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-Companion '26), ACM, 2026. doi:10.1145/3774748.3787670.

[5] M. Armbruster, M. Mazkatli, A. Koziolek, A process for continuous consistency-aware quality management (2025).

[6] M. Mazkatli, D. Monschein, M. Armbruster, R. Heinrich, A. Koziolek, Continuous integration of architectural performance models with parametric dependencies–the cipm approach, Automated Software Engineering 32 (2025) 1–79.

[7] R. Alur, Timed automata, in: International Conference on Computer Aided Verification, Springer, 1999, pp. 8–22.

[8] C. Gerking, D. Schubert, E. Bodden, Model checking the information flow security of real-time systems, in: International Symposium on Engineering Secure Software and Systems, Springer, 2018, pp. 27–43.

[9] R. Reussner, S. Becker, E. Burger, J. Happe, M. Hauck, A. Koziolek, H. Koziolek, K. Krogmann, M. Kuperberg, The palladio component model (2011).

[10] M. Armbruster, Parsing and Printing Java 7-15 by Extending an Existing Metamodel, Technical Report, Tech. rep. July 28, 2022.

[11] G. Liva, M. T. Khan, M. Pinzger, Extracting timed automata from java methods, in: 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2017, pp. 91–100.

[12] A. Legay, B. Delahaye, S. Bensalem, Statistical model checking: An overview, in: H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, N. Tillmann (Eds.), Runtime Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 122–135.

[13] H. Klare, M. E. Kramer, M. Langhammer, D. Werle, E. Burger, R. Reussner, Enabling consistency in view-based system development—the vitruvius approach, Journal of Systems and Software 171 (2021) 110815.

[14] A. T. Genç, M. Armbruster, M. Mazkatli, A. Koziolek, Towards continuous round-trip engineering of architectural performance models, in: 2026 IEEE 23rd International Conference on Software Architecture Companion (ICSA-C), 2026.

[15] B. Espinoza Becerra, Foundations of quantitative information flow: Channels, cascades, and the information order (2014).

[16] Y. Kawamoto, K. Chatzikokolakis, C. Palamidessi, On the compositionality of quantitative information flow, Logical Methods in Computer Science 13 (2017).

[17] A. R. Sadik, S. Brulin, M. Olhofer, A. Ceravola, F. Joublin, Llm as a code generator in agile model driven development, in: F. José Domínguez Mayo, L. Ferreira Pires, E. Seidewitz (Eds.), Model-Based Software and Systems Engineering, Springer Nature Switzerland, Cham, 2026, pp. 198–212.

[18] M. A. Ahmed, M. W. Anwar, W. H. Butt, F. Azam, Verigen: An llm-augmented framework for

end-to-end automation of software development lifecycle-from requirements specifications to code generation, IEEE Access (2026).

[19] A. Sabelfeld, D. Sands, Declassification: Dimensions and principles, Journal of Computer Security 17 (2009) 517–548.

[20] N. Ben Said, T. Abdellatif, S. Bensalem, M. Bozga, Model-driven information flow security for component-based systems, in: From Programs to Systems. The Systems perspective in Computing: ETAPS Workshop, FPS 2014, in Honor of Joseph Sifakis, Grenoble, France, April 6, 2014. Proceedings, Springer, 2014, pp. 1–20.

[21] K. Stenzel, K. Katkalov, M. Borek, W. Reif, A model-driven approach to noninterference., J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl. 5 (2014) 30–43.

[22] C. Sun, N. Xi, J. Li, Q. Yao, J. Ma, Verifying secure interface composition for component-based system designs, in: 2014 21st Asia-Pacific Software Engineering Conference, volume 1, 2014, pp. 359–366. doi:10.1109/APSEC.2014.60.

[23] N. Khakpour, C. Skandylas, Compositional security analysis of dynamic component-based systems, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 1232–1244.