2019 IEEE International Conference on Software Architecture (ICSA)

Component-Based Refinement and Verification of Information-Flow Security Policies for Cyber-Physical Microservice Architectures

Christopher Gerking
Heinz Nixdorf Institute
Paderborn University
Paderborn, Germany
Email: christopher.gerking@upb.de

David Schubert
Software Engineering and IT Security
Fraunhofer IEM
Paderborn, Germany
Email: david.schubert@iem.fraunhofer.de

***Abstract*—Since cyber-physical systems are inherently vulnerable to information leaks, software architects need to reason about security policies to define desired and undesired information flow through a system. The microservice architectural style requires the architects to refine a macro-level security policy into micro-level policies for individual microservices. However, when policies are refined in an ill-formed way, information leaks can emerge on composition of microservices. Related approaches to prevent such leaks do not take into account characteristics of cyber-physical systems like real-time behavior or message passing communication. In this paper, we enable the refinement and verification of information-flow security policies for cyber-physical microservice architectures. We provide architects with a set of well-formedness rules for refining a macro-level policy in a way that enforces its security restrictions. Based on the resulting micro-level policies, we present a verification technique to check if the real-time message passing of microservices is secure. In combination, our contributions prevent information leaks from emerging on composition. We evaluate the accuracy of our approach using an extension of the CoCoME case study.**

***Keywords*-security, microservices, cyber-physical systems**

I. Introduction

Cyber-physical systems interact intensively with their environments. Thereby, systems are able to coordinate their behavior and provide highly collaborative functionality. However, this interaction comes at a price because it also increases the vulnerability of leaking information to untrusted third parties. It is therefore crucial for software architects to reason early about security properties of a cyber-physical system under development.

One approach to reason about security is the theory of *information flow* which promotes numerous formal security properties like, e.g., *noninterference* [1]. In terms of information flow, the behavior of a system is secure if it provides no way for *secret* information to influence *public* information that is observable by third parties. By distinguishing between secret and public, the approach classifies information into different levels of sensitivity, thereby representing a *security policy* that systems must satisfy. Dedicated formal verification techniques provide means to check if a system satisfies its security policy and thereby prevents information leaks.

Nevertheless, information flow security is not easily applicable if cyber-physical systems are designed using component-based principles [2]. In this case, a system is decomposed into multiple components, and requires architects to "break down" a global security policy into a set of local policies, restricting the information flow of particular components. Using these local policies to reason about the overall system security requires *compositionality* of the verification. Unfortunately, it is a well-known fact that information flow security is not generally compositional [3]. Thus, even if all local security policies are satisfied, composing them in an ill-formed way can introduce global information leaks. In applications such as the Internet of things, the relevance of this problem increases along with the ongoing trend towards *microservice* architectures [4] because those result in a higher degree of decomposition and finer-grained security policies that need to be composed securely. Thus, software architects require guidance to refine security policies in a well-formed way that enables local reasoning about information flow.

Related work on architectural security either lacks the formal rigor of information flow [5]–[8], or does not take a number of cyber-physical system characteristics into account [9]–[11]. Namely, these systems must satisfy real-time constraints imposed by their environment. The resulting time-dependent behavior must be considered during the verification to detect implicit information leaks that are exploitable by observing the system's response times. Also, existing approaches assume a synchronous handshake communication between components. In contrast, cyber-physical systems interact asynchronously by lightweight message passing. Finally, cyber-physical systems implement complex, application-level interaction protocols that are based on a two-way communication. This so-called *feedback* is known as the most restrictive form of composition that often prevents a compositional verification [12].

In this paper, we realize and evaluate our early ideas [13] and present an approach to the refinement and verification of security policies for cyber-physical microservice architectures. Our approach is based on MechatronicUML [14], a component-based design method for cyber-physical systems.

978-1-7281-0528-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSA.2019.00015

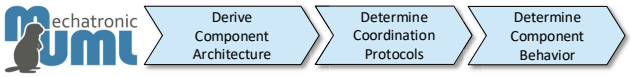

Fig. 1. MECHATRONICUML design process

Based on the MECHATRONICUML component model, we enable architects to specify security policies by classifying the interaction points of an architecture according to the sensitivity of the information they exchange. When architects decompose a coarse-grained architecture into finer-grained microservices, we provide them with a set of well-formedness rules for the refinement of security policies. These rules allow them to refine a global, macro-level security policy into a set of local, micro-level policies that enforce the global information flow restrictions. Finally, we present a compositional verification approach that allows to check the real-time message passing of a microservice against its micro-level security policy. Even on composition with feedback, our well-formedness rules ensure that macro-level security follows from the fact that all microservices satisfy their micro-level policies. Thus, the systematic refinement of policies enables local reasoning about the security of microservices, and allows architects to prevent information leaks that emerge from an ill-formed composition.

We illustrate and evaluate our approach using a security-oriented extension of the community case study *CoCoME* [15]. We take into account eight predefined security policies for different environmental parties that interact with the system. On this basis, we demonstrate the accuracy of the results provided by our approach when detecting insecure architectures.

In summary, our paper makes the following contributions:

- We provide a set of well-formedness rules to refine an information flow security policy during the decomposition of an architecture into microservices.
- We present a compositional verification approach to check microservices against their security policies, and preserve security on composition.
- We evaluate our approach using a security-oriented extension of the CoCoME case study.

Paper Organization: We provide background information on our underlying component model and the theory of information flow in Section II. Next, we give an overview on our approach in Section III, before describing the specification of security policies in Section IV. In Section V, we propose well-formedness rules for refined security policies, and describe our compositional verification approach in Section VI. Section VII presents the results of our case study. Finally, we discuss related work in Section VIII, before concluding in Section IX.

II. BACKGROUND

Initially, in Section II-A, we provide background information about the MECHATRONICUML component model. Subsequently, we introduce the theory of information flow in Section II-B.

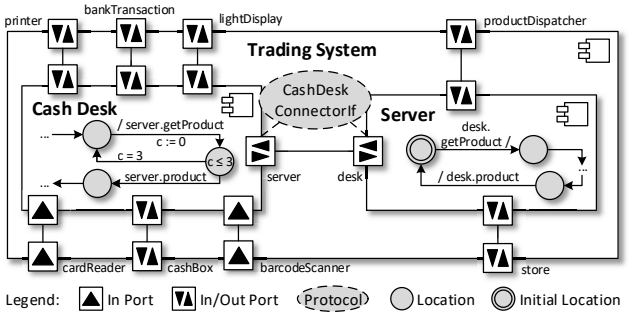

Fig. 2. MECHATRONICUML component architecture for CoCoME

A. MECHATRONICUML

MECHATRONICUML [14] is a component-based design method for cyber-physical systems, comprising a domain-specific modeling language and an underlying design process. In Fig. 1, we present a general view of this process.

1) Derive Component Architecture: The design starts with a stepwise derivation of a component architecture for the system under development. The starting point for the derivation is an initial top-level component that represents the entire system, including a set of interaction points for the external communication with the environment. The communication is realized by an asynchronous message passing over the interaction points, called *ports*. Thus, ports are used to send messages to environmental parties, or receive messages from them. A port that receives messages is referred to as *in* port, whereas an *out* port is one that sends messages. In contrast, a port that both sends and receives messages, and thereby enables a two-way communication, is referred to as *in/out* port.

As an excerpt of the community case study CoCoME [16], Fig. 2 shows a trading system in terms of a MECHATRONICUML component architecture. The top-level component Trading System comprises eight ports for the interaction with the environment. Among them, barcodeScanner and cardReader are *in* ports whereas all other ports are *in/out* ports.

MECHATRONICUML relies on a top-down approach to iteratively decompose the top-level component into *subcomponents*. Thus, by decomposing architectures from the macro-level to the micro-level, and due to the lightweight message passing, MECHATRONICUML advocates the microservice architectural style [4]. We therefore refer to the bottom-level components as microservices. A component with subcomponents is referred to as *composite* component. In our example, Trading System is a composite component that is decomposed into a Cash Desk and a Server. Internal communication within a composite component is enabled by *connectors* between ports. A connector between a composite component and one of its subcomponents is called *delegation* because it delegates the sending or receiving of messages to a subcomponent. In contrast, an *assembly* is a connector between two subcomponents. In Fig. 2, delegations exist for each of the eight external ports. Furthermore, an assembly enables the Cash Desk to communicate with the Server.

2) Determine Coordination Protocols: In the second step of the MECHATRONICUML process depicted in Fig. 1, software architects describe the interaction between components in terms of application-level communication protocols. These so-called *coordination protocols* define the messages that can be sent or received by two communicating ports within certain time frames. In our example, we use coordination protocols to represent the external and internal interfaces between the components of the CoCoME case study. Thus, in Fig. 2, we represent the interface CashDeskConnectorIf by means of a coordination protocol that defines the communication between the Cash Desk and Server components. Since server and desk are *in/out* ports representing a two-way communication, the components are said to be composed with *feedback* [12].

3) Determine Component Behavior: In the third step of the MECHATRONICUML process in Fig. 1, architects provide the bottom-level microservices inside the architecture with behavioral specifications, thereby implementing the predefined coordination protocols. Thus, each microservice is equipped with a stateful behavior that drives the message passing over its ports. To satisfy the real-time constraints imposed by coordination protocols, the behavior is based on the formalism of timed automata [17]. Thus, the microservice behavior is defined by a set of locations and edges between them. Firing an edge represents a switch from the source to the target location. The firing of edges, as well as the activity of locations, may be restricted to certain time frames. To that end, a microservice may use multiple real-valued *clocks* to measure the progress of time. The firing of an edge may also depend on the presence of a certain message that must have been received over a particular port. When an edge fires, it may send a message over a port as well, or reset the timing of a certain clock to zero. Whereas the behavior of microservices is defined on the basis of timed automata, the behavior of composite components emerges from the composition of microservices.

In Fig. 2, we depict an excerpt from the behavior of Cash Desk and Server. As part of the CashDeskConnectorIf protocol, the cash desk uses its server port to send a getProduct message. Thereby, it requests information about a certain product from the server. At the same time, it resets the clock c that acts as a timeout. After receiving the message, the server responds with a product message over its desk port, providing the cash desk with the requested information. However, the cash desk only processes the response within a time frame of $c \leq 3$. Otherwise, if the response has not been received at $c = 3$, it switches back to its original location in order to request the information again.

B. Information Flow Security

The theory of *information flow security* [1] allows to analyze the behavior of a system with respect to information leaks. Intuitively, a system is secure if secrets are never made public. Thus, secret information processed by a system must not be deducible through the public information that the system exchanges with untrusted third parties. To that end, the approach considers information at different levels of sensitivity:

- *secret* information must not be disclosed to the public. Therefore, the behavior of a system must not allow third parties to deduce which secret information is processed.
- *public* information is exchanged with third parties. Thus, the system behavior must ensure that observing this information does not enable any deductions about secrets.
- *neutral* information is neither secret nor public. In contrast to secret information, it may be disclosed to third parties. In contrast to public information, observing neutral information may enable deductions about secrets.

1) Perturbation & Correction: Information flow security requires the behavior of a system to prevent secret information from influencing public information. In order to define the meaning of influence more precisely, the approach relies on the notion of *perturbation* [3]. A perturbation is a change in the amount of the secret information that a system processes during an execution. Perturbing secret information must have no influence on the public information processed during that execution, otherwise an information flow from secret to public would be possible. Perturbations of secret information may, however, influence neutral information. This is an important factor as it allows for a *correction* [3] in response to a perturbation. By means of a correction, a system adjusts the amount of neutral information processed during an execution, thereby allowing the public information to remain uninfluenced. Thus, since neutral information is used for corrections, observing it may enable deductions about secrets. This side effect needs to be taken into account when composing a system from multiple subsystems. In this case, neutral information may leak secrets from one subsystem to another, and therefore must be treated with special care on composition.

2) Generalized Noninterference: Information flow security promotes numerous formal security properties [3], differing in (i) what sensitivity is assigned to certain information, (ii) what perturbations are taken into account, and (iii) which corrections are allowed. Our approach is based on *generalized noninterference* [18], a traditional information flow property that restricts perturbations to inputs. Hence, to perturb a system, secret inputs are both added to or deleted from the execution. In response to the perturbation, the system may add or delete neutral inputs or outputs to produce a corrected execution without influencing any public information. Generalized noninterference is known to be preserved on feedback composition, provided that the underlying communication between the composed subsystems is asynchronous [12]. Thus, we make use of this characteristic and use generalized noninterference as our underlying information flow property.

3) Simulation Preorder: Since public information needs to be uninfluenced by secrets, the public observations allowed by a system must subsume all the public observations allowed by a perturbed variant of the same system. In the scope of state transition systems, such a behavioral relation is expressible in terms of a *simulation* preorder [19]. Thus, the original system must be able to *simulate* all the public observations allowed by the perturbed variant. In this paper, we use simulations as our underlying verification technique.

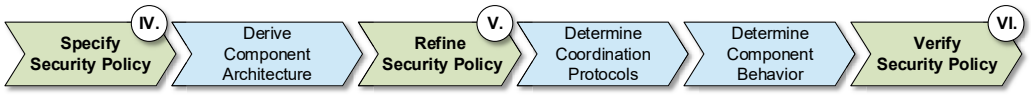

Fig. 3. Security-related extensions to the MECHATRONICUML design process as proposed in this paper

III. OVERVIEW

In the remainder of this paper, we extend the MECHATRONICUML process introduced in Section II-A with additional security-related steps. Fig. 3 illustrates our extensions including their dedicated sections within this paper. Initially, as a starting point for the derivation of the architecture, we enable architects to specify a macro-level security policy for the system under development. We describe the specification of such policies in Section IV. In response to the derivation of the architecture, we provide architects with a rule set for refining security policies from the macro-level to the micro-level. We present the corresponding well-formedness rules in Section V. Finally, after the predefined coordination protocols have been implemented, we propose to check the behavior of microservices against their micro-level security policies. The underlying verification approach is given in Section VI.

IV. SPECIFICATION OF SECURITY POLICIES

In the scope of our paper, a security policy is a definition of critical information flow that would violate a system's security requirements. Thus, a policy declares the sources and sinks of a critical flow through a system. According to our component-based approach, software architects specify security policies per component. Therefore, a security policy classifies the component's ports according to the sensitivity of the information they send or receive. The classification is based on the sensitivity levels introduced in Section II-B. In the following, we introduce the sensitivities of ports and their visual encoding for the remainder of this paper:

- *secret* ports represent sources of critical information flows. Accordingly, the information received through such a port is secret, and untrusted third parties must not be able to deduce these information.
- *public* ports are sinks of critical information flows. They represent the interaction with untrusted third parties. Thus, the information exchanged over public ports must not allow any deductions about secret information.
- *neutral* ports do not restrict the security policy. The information exchanged by a neutral port is not considered secret itself, but may enable deductions about other secret information.

In Fig. 4, we depict an example security policy for the Trading System introduced in Section II-A. The policy specifies that the information received from a bank over the bankTransaction port is secret. Furthermore, the productDispatcher port is classified as public, and therefore must not enable deductions about the information received from the bank. Since all other ports are neutral, their information flow is unrestricted.

A component satisfies its security policy if the inputs that it receives over secret ports do not influence the observations that third parties can make over public ports. Beside outputs, these observations also subsume the inputs received over public ports. Thereby, we ensure that the entire message passing between public ports is never influenced by secret information. Furthermore, our notion of information flow is time-dependent and therefore requires that the timing of public observations remains uninfluenced by secret inputs as well. Thereby, we seek to detect so-called *timing channels* [20] that would otherwise allow third parties to deduce secrets from the instant of time at which certain observations are made.

Please note that a security policy typically represents the confidentiality or integrity requirements of a single party. For example, the policy illustrated in Fig. 4 describes the requirements of a bank that interacts with the system. However, as indicated by the example of the Trading System, systems typically interact with multiple parties, resulting in different policies with alternating sensitivities for particular ports. Therefore, different security policies for the same component may co-exist, whereas the component must satisfy all of them at once to be secure.

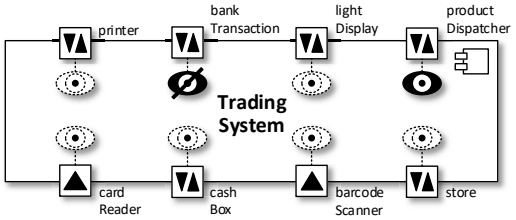

Fig. 4. Example security policy of the Trading System

V. WELL-FORMEDNESS OF REFINEMENTS

In this section, we guide software architects during the refinement of security policies on decomposition of a component into a composite component. In this situation, architects need to refine the component's global policy into a set of local policies, one for each resulting subcomponent. These local policies classify the ports of all subcomponents by means of the sensitivity levels introduced in Section IV. However, the refinement must be *well-formed*, i.e., every two connected ports must be classified consistently such that the composition of local policies enforces the information flow restrictions of the global policy. To that end, we provide software architects with a rule set that ensures well-formedness of refinements and prevents information leaks from emerging on composition of subcomponents. The proposed rules are based on the theoretical foundations of compositional information flow

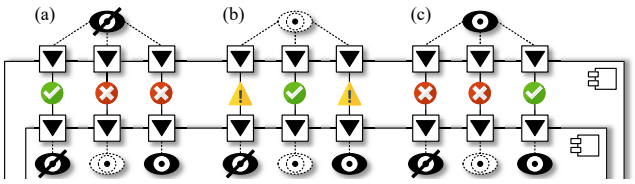

Fig. 5. Well-formedness rules for input delegations

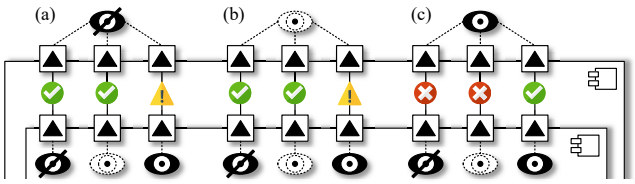

Fig. 6. Well-formedness rules for output delegations

security by Mantel [3]. In our paper, we apply these rules in an architectural context. Thus, our rules are applicable to individual connectors, and check whether a particular connection is consistent with respect to the sensitivity levels of the connected ports. Our rules refer to a unidirectional connection between distinct source and target ports. Therefore, a feedback composition using bidirectional *in/out* ports must comply with the rules for both directions. The application of a rule to a particular connection leads to one of three different results. In the following, we represent these results visually. First, ✅ denotes a well-formed refinement. Second, ❌ denotes an ill-formed refinement. Third, a refinement is denoted by ⚠ if it is not ill-formed, but refines the security policy in an overly restrictive way. In the upcoming Section V-A, we address the well-formedness of delegations from a composite component to one of its subcomponents. Subsequently, in Section V-B, we establish rules for assemblies between two subcomponents.

A. Delegation

A delegation requires the port of a subcomponent to be classified consistently with the delegating port of the composite component. Thus, the assumptions made by the composite component with respect to the secrecy of information must not be violated by a subcomponent. Furthermore, since the composite component guarantees that any public information will not be influenced by secrets, the subcomponent must provide this security guarantee. In the following, we distinguish between delegations for *in* ports and *out* ports.

1) In Ports: The respective well-formedness rules for *in* ports are depicted in Fig. 5, which may be interpreted as a decision tree: if the delegating port is classified as secret (a), a well-formed refinement requires the port of the subcomponent to be secret as well. Thus, secret information received by a composite component must also be considered secret by the subcomponent that the information is passed on to. Other refinements, classifying the port of the subcomponent as neutral or public, are both ill-formed. Thereby, we ensure that subcomponents will not violate the assumptions of the composite component by reducing the secrecy of information.

The refinement of neutral *in* ports (b) is less restrictive. Since a neutral *in* port does not restrict the security policy, it may delegate to arbitrarily classified ports of subcomponents. However, delegating to a secret or public port will make the local security policy more restrictive than requested by the global policy of the composite component. Therefore, only delegations to neutral ports are considered well-formed.

In case of public *in* ports (c), a well-formed refinement requires the connected port of the subcomponent to be public as well, whereas other refinements are all ill-formed. Thereby, we ensure that the port's input behavior does not depend on secret information, thereby providing the guarantee given by the security policy of the composite component.

2) Out Ports: In case of an *out* port, the well-formedness rules for secret and neutral information are equivalent. As depicted in Fig. 6, whenever a port of a composite component is classified as secret (a) or neutral (b), a well-formed refinement allows the associated port of the subcomponent to be either secret or neutral as well. The reason for this flexibility is that both secret and neutral outputs allow deductions about secrets. Thus, delegations may connect secret and neutral *out* ports interchangeably. Secret or neutral *out* ports of a composite component may also be connected to public ports of a subcomponent. However, this makes the local security policy more restrictive than requested because a public port of a subcomponent will never send any information that allows deductions about secrets.

In contrast, public *out* ports of a composite component (c) involve more restrictive rules, which are equivalent to the rules for public *in* ports. Accordingly, a well-formed refinement requires the associated port of a subcomponent to be public as well, whereas all other refinements are ill-formed. This rule ensures that the subcomponent provides the guarantee given by the composite component, which requires that the outputs of public ports must not depend on secret information.

Summary: The above rules are always complied if the ports of subcomponents are classified in the same way as their associated delegating ports. Intuitively, this regulation suggests that subcomponents should treat information with the same sensitivity as the composite component.

B. Assembly

In case of an assembly, we need to ensure that secrets are not leaked from one subcomponent to another in an uncontrolled way. Thus, as depicted in Fig. 7, a well-formed refinement requires that a secret (a) or neutral (b) *out* port of one subcomponent must be connected to a secret *in* port of another subcomponent. In contrast, if the connected *in* port is neutral or public, the security policy is refined in an ill-formed way. This rule is motivated by the fact that information sent over secret or neutral *out* ports will potentially allow deductions about secrets. Thus, it is mandatory that these information are received over secret *in* ports to ensure that they will not be leaked to the public.

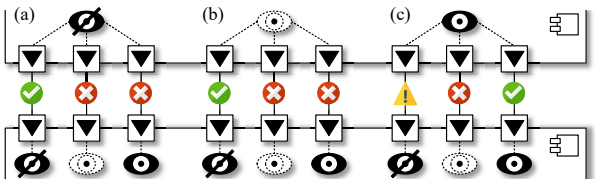

Fig. 7. Well-formedness rules for assemblies

If an *out* port is classified as public (c), the connected *in* port must be public as well in order to yield a well-formed refinement. Otherwise, if the connected *in* port is neutral, the way in which it processes received messages may depend on secrets. The refinement is therefore ill-formed because no guarantee is given that the communication over the public *out* port remains uninfluenced by secrets. In contrast, if the *in* port is secret, the security policy is more restrictive than necessary because the sending component already guarantees that the information does not enable any deductions about secrets.

Summary: Note that a neutral *in* port is rejected in any case, regardless of which sensitivity the connecting *out* port has. Therefore, neutral *in* and *in/out* ports must not be used in combination with assemblies.

VI. Compositional Verification

In this section, we propose a verification technique that allows to check if the behavior of a microservice satisfies its micro-level security policy. First, we give an overview on the general approach in Section VI-A. Afterwards, we describe the procedure of automating the verification in Section VI-B.

A. General Approach

The proposed verification technique is based on *generalized noninterference* [18] as the underlying information flow property (cf. Section II-B). We select this property because it is preserved on feedback composition of asynchronously communicating components, which is in accordance with the message passing communication of MechatronicUML (cf. Section II-A). By preserving security on composition, our technique enables a compositional verification that allows to reason about a macro-level security policy based on the verified micro-level policies.

To verify generalized noninterference, we need to check that perturbing the execution of a microservice by adding or deleting secret inputs does not change the public messages processed during that execution. Accordingly, the public observations that are allowed by the perturbed behavior need to be fully included in those public observations that the original behavior allows. However, according to the real-time behavior that underlies our approach, we also need to ensure that the timing of public messages remains unchanged by arbitrarily timed perturbations. Therefore, in this paper, we rely on the notion of *timed simulation* [19] as a preorder between the real-time behavior of a microservice and the corresponding perturbed behavior. Thereby, we check that every public message processed by the perturbed behavior in a certain time frame is properly *simulated*, i.e., may also be processed by the original behavior in that time frame. If so, the perturbed behavior does not enable any additional public observations that are not allowed by the original behavior. Thus, when third parties observe the timing of public messages during a particular execution, they can not distinguish the perturbed behavior from the original behavior. Accordingly, no information flow from secret to public is possible.

In Fig. 8, we illustrate this verification approach using the example of the Cash Desk microservice. On the left-hand side, we show the microservice in the original state, and represent the perturbed variant on the right-hand side. The depicted security policy excludes an information flow from the secret bankTransaction port to the public server port. Thus, on the right-hand side of Fig. 8, we depict in red the perturbation of secret inputs received over the bankTransaction port. In the center, we illustrate the concept of timed simulation in terms of a preorder. Accordingly, every public behavior exhibited by the perturbed variant within a certain time frame must also be exhibited by the unperturbed Cash Desk in that time frame.

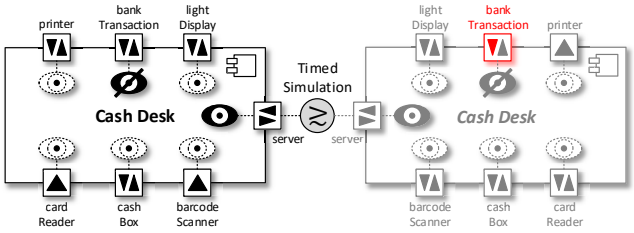

Fig. 8. Illustration of the general verification approach for the Cash Desk

B. Procedure

The goal of our approach is to automate the check for security using an off-the-shelf verification tool. To that end, we rely on our previous work [21] that reduces the check for information flow security of real-time systems to a so-called *refinement* check. A refinement check [19] provides means to verify different behavioral relations between real-time systems, i.e., it checks that the communication behavior of one system is properly refined by another one. Since timed simulation is among these relations, we apply the approach to verify that the real-time behavior of a microservice is properly refined by the perturbed behavior. In Fig. 9, we give an overview on our verification procedure.

To verify the information flow security of a microservice as depicted at the top of Fig. 9, the initial step ❶ of our verification procedure is to *conditionalize* the inputs received over secret ports. Essentially, the purpose of this step is to introduce perturbations to the behavior of the microservice. To this end, we enrich its behavioral specification with additional conditions that are added to every edge that processes a secret input. The additional conditions put the firing of these edges under the control of dedicated boolean variables, which are manipulated nondeterministically during execution in order to enable or disable the processing of a secret input. In Fig. 9, we

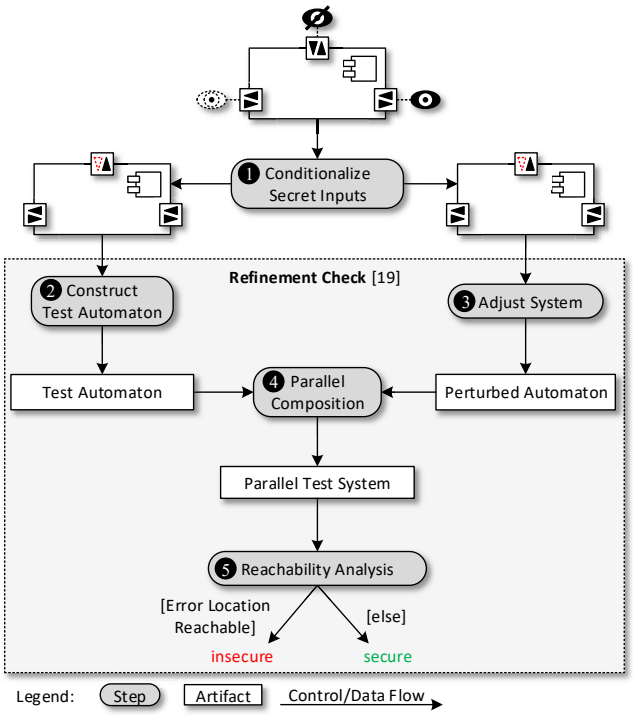

Fig. 9. Overview of the verification procedure

depict this approach in terms of a dashed red line around the secret inputs of our microservice. The disabling corresponds to a perturbation that deletes an input from the execution. In order to represent not only the deletion but also the addition of secret inputs, we create two copies of the resulting conditionalized behavior. Both copies are perturbed independently from each other. Thus, during an execution, they differ arbitrarily in the amount of secret inputs they process. In particular, from the perspective of one copy, the execution of the other copy differs nondeterministically in terms of both added and deleted inputs. Accordingly, in Fig. 9, we consider the right-hand microservice as the perturbed variant of the microservice on the left-hand side. Subsequently, we use both of them as the inputs to the refinement check. Essentially, our approach composes the behavior of a system with itself (cf. Fig. 9). Thus, using this approach as a verification technique for information flow security is also known as *self-composition* [22].

In step ❷, the refinement check transforms the behavior of the left-hand microservice into a *test automaton* [19] that represents the central concept of the verification approach. The test automaton introduces a dedicated *error* location that indicates a violation of the timed simulation whenever it becomes reachable during execution. Thus, reaching that location represents the detection of an information leak. In this paper, we omit the technical details on the construction of test automata. Instead, we refer the reader to [19] with respect to the construction of test automata for timed simulation, and [21] for the application of test automata in the context of information flow security.

Step ❸ adjusts the perturbed microservice on the right-hand side such that it synchronizes its behavior with the test automaton whenever both send or receive the same message. In the subsequent step ❹, both automata are composed in parallel to ensure their synchronized execution. Thereby, the approach enables the perturbed automaton to drive the execution of the test automaton. The resulting parallel test system gives rise to a reachability analysis in step ❺. Whenever the perturbed behavior processes a public message that is not allowed by the unperturbed behavior in that time frame, the test automaton switches to its *error* location. Thereby, the verification of information flow security reduces to checking the reachability of the *error* location during the execution of the parallel test system. The system is insecure if the *error* location is reachable during any execution, and is secure otherwise. The reachability analysis is carried out using the off-the-shelf model checker UPPAAL [23] that is capable of analyzing the real-time behavior of timed automata.

VII. CASE STUDY

In this section, we present the case study that we conducted to evaluate our approach. To report our study, we rely on the structure proposed by Runeson and Höst [24]. Accordingly, in Section VII-A, we describe the design of our study. Section VII-B presents the results we obtained. In Section VII-C, we discuss threats to the validity of our findings.

A. Design

The goal of our study is to answer the following research question: ***How accurate are the security results that architects obtain on verification of refined security policies?*** In particular, secure architectures should be accurately separated from insecure architectures which leak information.

1) Case Selection: The case that we select for our study is the community case study CoCoME [16] that deals with a trading system designed using component-based principles. In particular, we select CoCoME due to its predefined information flow security requirements [15] which serve as benchmark during our evaluation.

2) Data Collection: We recreate CoCoME's architecture in terms of a MECHATRONICUML component model as depicted in Fig. 10. Ports in our architecture correspond to the interfaces for communication between CoCoME's components. We represent required interfaces as *out* ports, whereas provided interfaces translate into *in* ports. Furthermore, an interface that comprises at least one return value corresponds to an *in/out* port. Based on the resulting component architecture, we specify a set of eight macro-level security policies that reflect the security requirements for CoCoME [15]. These requirements describe whether the information received through an interface is allowed to be disclosed to different environmental parties that interact with the trading system. Thus, each of our policies reflects the perspective of one external interface, and regards this interface and the corresponding port as secret. In Table I, each row represents one of the eight resulting macro-level policies, highlighting the secret port of each policy in grey.

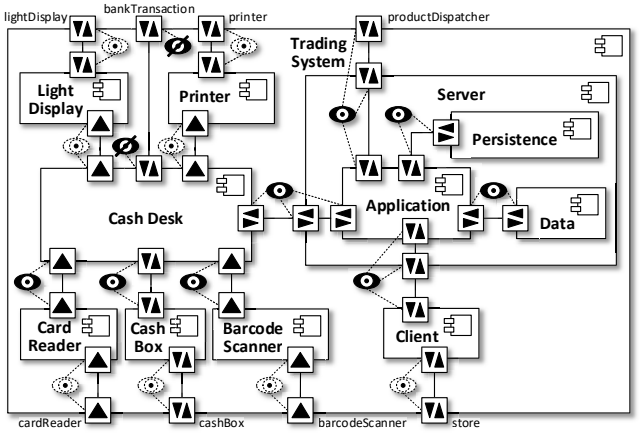

Fig. 10. Refined security policy for the bankTransaction port

To complete a macro-level security policy, the remaining external ports are classified according to whether their corresponding interfaces are allowed to disclose the parameters passed to the secret interface. If all these parameters may be disclosed to all the parties that interact over an external interface, then the port that corresponds to this interface is classified as neutral. In contrast, if any of the parameters must not be disclosed to at least one of the interacting parties, the port is classified as public. Subsequently, we refine each of the macro-level policies following the well-formedness rules given in Section V. Thereby, we create a micro-level policy for each microservice. For example, Fig. 10 depicts a refinement of the security policy for the secret bankTransaction port.

In the next step, we use UPPAAL timed automata to implement the behavior of microservices. Our implementations reflect the causal dependencies between processed messages, i.e., the resulting automata describe the possible sequences of inputs received or outputs sent by a microservice. Furthermore, we add real-time constraints that enable a microservice to react to timeouts during the communication with other microservices. Based on the implementations, we manually rate each macro-level policy according to whether it is satisfied by the behavior of the overall system. Thereby, as shown in Table I, we provide each policy with an expected result for the subsequent verification that is either *secure* or *insecure*.

Finally, to compare our approach against the specified expectations, we verify the refined micro-level security policies using the UPPAAL model checker as proposed in Section VI. Thereby, we end up with a verification result for each microservice against each security policy.

3) Analysis: To analyse the outcomes of our study, we compare the results obtained from the verification against the expected security results. Thereby, we check if our predefined expectations are satisfied. If the system is expected to be secure with respect to a policy, that expectation is satisfied if all the microservices are also verified as secure. In contrast, an insecure expectation is satisfied if at least one of the microservices is insecure according to the verification.

B. Results

On the right-hand side of Table I, we show the results of our case study. A microservice that has been verified as secure is marked with ✓, whereas ✗ denotes an insecure verification result. In Table I, we restrict our presentation to the verification results for the Light Display, Printer, Cash Desk, and Application as the only microservices that are effectively restricted by at least one of their micro-level security policies. The remaining microservices are never effectively restricted because their micro-level policies do not combine secret *and* public ports (cf. Fig. 10). Thus, these microservices are secure by definition and therefore omitted from Table I.

Concerning the first four security policies, all microservices have been verified as secure, thereby satisfying our expectation. The remaining four security policies produce insecure results for at least one of the microservices. However, in case of the Application, no results are available for two policies due to excess memory usage of UPPAAL. Nevertheless, since the Cash Desk produces an insecure result in both of these cases, our expectations for the last four policies are satisfied as well. In summary, no false positives or false negatives were detected, although the verification of one microservice terminated abnormally for two policies. Nevertheless, the result of our case study is that secure architectures could be accurately separated from insecure architectures.

C. Validity

In terms of *construct validity*, our study depends on expected results that have been rated manually. Thereby, we seek to overcome the lack of a *ground truth* that would allow us to validate the accuracy of results automatically. Furthermore, since the selected case [15] is based on an alternative definition of information flow security compared to our approach, the results of our verification might not be an appropriate indicator to assess the accuracy.

The *internal validity* of our study is threatened by the fact that we did not analyze the behavior of CoCoME as is. Instead, since UPPAAL does not support an object-oriented programming style, our implementations omit the explicit data handling of microservices. Thus, we abstract from the way that individual parameters are processed, stored, and propagated. Thereby, we might have masked both false positive and false negative errors.

In the context of *external validity*, the generalizability of our findings is affected because CoCoME is not a cyber-physical system in a narrower sense. Although the trading system involves hardware components, the case abstracts from their interaction at the physical level. In particular, no hard real-time constraints are imposed on the system by the physical environment. Therefore, our findings might not be generalizable to cases that actually show these characteristics.

We foster the *reliability* of our study by providing supplementary material [25] that includes the underlying implementations of microservices in terms of UPPAAL timed automata, as well as the corresponding verification models. Thereby, we seek to enable other researchers to reproduce our results.

TABLE I
SECURITY POLICIES AND VERIFICATION RESULTS

External Ports									Microservices			
light Display	**bank Transaction**	**printer**	**product Dispatcher**	**store**	**barcode Scanner**	**cash Box**	**card Reader**	**Expected Result**	**Light Display**	**Printer**	**Cash Desk**	**App.**
⌀	👁	👁	👁	👁	👁	👁	👁	secure	✓	✓	✓	✓
👁	⌀	👁	👁	👁	👁	👁	👁	secure	✓	✓	✓	✓
👁	👁	⌀	👁	👁	👁	👁	👁	secure	✓	✓	✓	✓
👁	👁	👁	⌀	👁	👁	👁	👁	secure	✓	✓	✓	✓
👁	👁	👁	👁	⌀	👁	👁	👁	insecure	✓	✓	✓	✗
👁	👁	👁	👁	👁	⌀	👁	👁	insecure	✓	✓	✗	N/A
👁	👁	👁	👁	👁	👁	⌀	👁	insecure	✓	✓	✗	N/A
👁	👁	👁	👁	👁	👁	👁	⌀	insecure	✓	✓	✗	✓

VIII. RELATED WORK

In the following Section VIII-A, we first give a general overview on the field of architectural security, including both analytical and constructive approaches. Next, in Section VIII-B, we discuss related work at the more specific level of compositional information flow security.

A. Architectural Security

Analytical approaches towards security at the architectural level differ widely in the concrete aspects being analyzed. For example, Sion et al. [26] address the elicitation of threats and the assessment of corresponding risks inside data flow diagrams. Grunske and Joyce [27] also enable the analysis of security risks, but take component-based architectures into account, as we do in this paper. Buyens et al. [28] analyze architectures with respect to the design principle of *least privilege*. Seifermann [29] lifts data flow analyses from the source code to the level of components, whereas Taspolatoglu and Heinrich [30] use the same component model to identify vulnerabilities arising from evolutionary changes. The more closely related works by Gunawan and Herrmann [31] as well as Copet and Sisto [32] both propose compositional verification approaches for security properties, whereas Mohammad and Alagar [5] as well as Borda et al. [6] even take cyber-physical systems into account. However, unlike our work, none of these compositional approaches provides the formal rigor of information flow security.

In the area of *constructive approaches*, Jasser et al. [33] use the results of a threat analysis to refactor data flow diagrams and thereby improve a system's security. Shin et al. [34] enrich component architectures with reusable connectors that encapsulate mechanisms for secure communication. In contrast, Saadatmand and Leveque [35] follow a generative approach that transforms a component architecture into a secured form that integrates such security mechanisms. Uzunov et al. [7] propose a process that guides the architectural decomposition of a system and enables the resulting architecture to mitigate security threats. Unlike our work, the authors do not consider information flow security and the refinement of corresponding policies. Zhou and Alves-Foss [8] provide software architects with dedicated architectural patterns to decompose, aggregate, or eliminate components in a way that preserves security policies. Whereas the general intent is similar to this paper, our refinement rules are not restricted to certain architectural patterns, and thereby leave architects free to decompose a system more flexibly. Furthermore, although the authors rely on security policies with multiple levels, their underlying security model is not based on information flow.

B. Compositional Information Flow

Since the seminal work on compositional information flow by McCullough [18], numerous practical approaches have been proposed. Among them, the approaches by Abdellatif et al. [9] and Greiner et al. [10] both address component-based architectures. The approach by Yurchenko et al. [36] generates verifiable proof obligations from component-based information flow policies. However, unlike our approach, none of these works addresses the systematic refinement of policies during the architectural decomposition. In contrast, Chong and van der Meyden [11] enable a top-down refinement of architectural security policies. They propose a compositional approach to infer security of a system from its global structure and local properties of single components. As opposed to our approach that uses model checking as a concrete verification technique, the authors provide a more general framework that leaves open how to prove the local properties of components. Furthermore, none of the above approaches takes crucial characteristics of cyber-physical systems into account.

In contrast, Li et al. [37] enable compositional information flow in the presence of message passing. Similarly, Rafnsson et al. [38] propose a set of *combinators* to preserve security on composition of time-dependant and asynchronously communicating systems, thereby matching the cyber-physical system characteristics addressed in this paper. However, compared to our work, both approaches analyze the security of processes at program level, and take neither the architectural design, nor the refinement of security policies into account. Furthermore, compared to our work in the context of real-time systems, the work by Rafnsson et al. [38] is limited to discrete time.

IX. Conclusion

In this paper, we enriched the architectural decomposition of cyber-physical microservice architectures with a rule set that ensures a well-formed refinement of information-flow security policies. Furthermore, we used the off-the-shelf tool UPPAAL to check the real-time behavior of microservices against their refined policies. Our underlying security property preserves the verification results on composition of microservices. In our evaluation, we successfully demonstrated the accuracy of these results in the context of the CoCoME case study.

Our well-formedness rules enable software architects to systematically refine security policies on decomposition of component-based systems, thereby "breaking down" a global policy to enable local reasoning about security. The security property that underlies our verification prevents information leaks from emerging on composition of verified microservices, and thereby enables the construction of secure architectures.

In future work, we intend to re-prove the compositionality of generalized noninterference in the presence of real-time behavior to formally underpin our approach. We also plan to apply our rule set in a search-based approach to explore the architectural design space and identify potential refinements of security policies automatically. Finally, we intend to address variability and self-adaptation, thereby enabling architectures to adjust to diverse security situations.

Acknowledgment

This work was partially developed in the ITEA 3 project "APPSTACLE" funded by the German Federal Ministry of Education and Research (no. 01IS16047I).

References

[1] H. Mantel, "Information flow and noninterference," in *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 605–607.

[2] I. Crnkovic, I. Malavolta, H. Muccini, and M. Sharaf, "On the use of component-based principles and practices for architecting cyber-physical systems," in *CBSE 2016*. IEEE Computer Society, 2016, pp. 23–32.

[3] H. Mantel, "On the composition of secure systems," in *S&P 2002*. IEEE Computer Society, 2002, pp. 88–101.

[4] P. D. Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *ICSA 2017*. IEEE Computer Society, 2017, pp. 21–30.

[5] M. Mohammad and V. S. Alagar, "A formal approach for the specification and verification of trustworthy component-based systems," *Journal of Systems and Software*, vol. 84, no. 1, pp. 77–104, 2011.

[6] A. Borda, L. Pasquale, V. Koutavas, and B. Nuseibeh, "Compositional verification of self-adaptive cyber-physical systems," in *SEAMS 2018*. ACM, 2018, pp. 1–11.

[7] A. V. Uzunov, K. Falkner, and E. B. Fernández, "Decomposing distributed software architectures for the determination and incorporation of security and other non-functional requirements," in *ASWEC 2013*. IEEE Computer Society, 2013, pp. 30–39.

[8] J. Zhou and J. Alves-Foss, "Security policy refinement and enforcement for the design of multi-level secure systems," *Journal of Computer Security*, vol. 16, no. 2, pp. 107–131, 2008.

[9] T. Abdellatif, L. Sfaxi, R. Robbana, and Y. Lakhnech, "Automating information flow control in component-based distributed systems," in *CBSE 2011*. ACM, 2011, pp. 73–82.

[10] S. Greiner, M. Mohr, and B. Beckert, "Modular verification of information flow security in component-based systems," in *SEFM 2017*. Springer, 2017, pp. 300–315.

[11] S. Chong and R. van der Meyden, "Using architecture to reason about information security," *ACM Transactions on Information and System Security*, vol. 18, no. 2, pp. 8:1–8:30, 2015.

[12] A. Zakinthinos and E. S. Lee, "How and why feedback composition fails," in *CSFW 1996*. IEEE Computer Society, 1996, pp. 95–101.

[13] C. Gerking and D. Schubert, "Towards preserving information flow security on architectural composition of cyber-physical systems," in *ECSA 2018*. Springer, 2018, pp. 147–155.

[14] S. Becker, S. Dziwok, C. Gerking, C. Heinzemann, W. Schäfer, M. Meyer, and U. Pohlmann, "The MECHATRONICUML method: Model-driven software engineering of self-adaptive mechatronic systems," in *ICSE Companion 2014*. ACM, 2014, pp. 614–615.

[15] S. Greiner and M. Herda, "CoCoME with security," Karlsruher Institut für Technologie, Tech. Rep. 2, 2017.

[16] S. Herold et al., "CoCoME," in *The Common Component Modeling Example*. Springer, 2008, pp. 16–53.

[17] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[18] D. McCullough, "Noninterference and the composability of security properties," in *S&P 1988*. IEEE Computer Society, 1988, pp. 177–186.

[19] C. Heinzemann, C. Brenner, S. Dziwok, and W. Schäfer, "Automata-based refinement checking for real-time systems," *Computer Science - R&D*, vol. 30, no. 3-4, pp. 255–283, 2015.

[20] A. K. Biswas, D. Ghosal, and S. Nagaraja, "A survey of timing channels and countermeasures," *ACM Computing Surveys*, vol. 50, no. 1, pp. 6:1–6:39, 2017.

[21] C. Gerking, D. Schubert, and E. Bodden, "Model checking the information flow security of real-time systems," in *ESSoS 2018*. Springer, 2018, pp. 27–43.

[22] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," *Mathematical Structures in Computer Science*, vol. 21, no. 6, pp. 1207–1252, 2011.

[23] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL – a tool suite for automatic verification of real-time systems," in *Hybrid Systems III*. Springer, 1996, pp. 232–243.

[24] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[25] C. Gerking, "Checking secure information flow of CoCoME using UPPAAL." [Online]. Available: https://doi.org/10.5281/zenodo.2559472

[26] L. Sion, D. V. Landuyt, K. Yskout, and W. Joosen, "SPARTA: security & privacy architecture through risk-driven threat assessment," in *ICSA Companion 2018*. IEEE Computer Society, 2018, pp. 89–92.

[27] L. Grunske and D. Joyce, "Quantitative risk-based security prediction for component-based systems with explicitly modeled attack profiles," *Journal of Systems and Software*, vol. 81, no. 8, pp. 1327–1345, 2008.

[28] K. Buyens, R. Scandariato, and W. Joosen, "Least privilege analysis in software architectures," *Software and System Modeling*, vol. 12, no. 2, pp. 331–348, 2013.

[29] S. Seifermann, "Architectural data flow analysis," in *WICSA 2016*. IEEE Computer Society, 2016, pp. 270–271.

[30] E. Taspolatoglu and R. Heinrich, "Context-based architectural security analysis," in *WICSA 2016*. IEEE Computer Society, 2016, pp. 281–282.

[31] L. A. Gunawan and P. Herrmann, "Compositional verification of application-level security properties," in *ESSoS 2013*. Springer, 2013, pp. 75–90.

[32] P. B. Copet and R. Sisto, "Automated formal verification of application-specific security properties," in *ESSoS 2014*. Springer, 2014, pp. 45–59.

[33] S. Jasser, K. Tuma, R. Scandariato, and M. Riebisch, "Back to the drawing board," in *ICISSP 2018*. SciTePress, 2018, pp. 438–446.

[34] M. E. Shin, H. Gomaa, D. Pathirage, C. Baker, and B. Malhotra, "Design of secure software architectures with secure connectors," *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, no. 5, pp. 769–806, 2016.

[35] M. Saadatmand and T. Leveque, "Modeling security aspects in distributed real-time component-based embedded systems," in *ITNG 2012*. IEEE Computer Society, 2012, pp. 437–444.

[36] K. Yurchenko, M. Behr, H. Klare, M. E. Kramer, and R. H. Reussner, "Architecture-driven reduction of specification overhead for verifying confidentiality in component-based software systems," in *ModComp 2017*, 2017, pp. 321–323.

[37] X. Li, H. Mantel, and M. Tasch, "Taming message-passing communication in compositional reasoning about confidentiality," in *APLAS 2017*. Springer, 2017, pp. 45–66.

[38] W. Rafnsson, L. Jia, and L. Bauer, "Timing-sensitive noninterference through composition," in *POST 2017*. Springer, 2017, pp. 3–25.