Compositional Security Analysis of Dynamic Component-based Systems

Narges Khakpour
narges.khakpour@newcastle.ac.uk
Newcastle University
Newcastle, United Kingdom
Linnaeus University
Växjö, Sweden

Charilaos Skandylas
charilaos.skandylas@liu.se
Linköping University
Linköping, Sweden
Linnaeus University
Växjö, Sweden

ABSTRACT

To reason about and enforce security in dynamic software systems, automated analysis and verification approaches are required. However, such approaches often encounter scalability issues, particularly when employed for runtime analysis, which is necessary in software systems with dynamically changing architectures, such as self-adaptive systems. In this work, we propose an automated formal approach for security analysis of component-based systems with dynamic architectures. This approach leverages formal abstraction and incremental analysis techniques to reduce the complexity of runtime analysis. We have implemented and evaluated our approach against ZNN, a widely known self-adaptive system exemplar. Our experimental results demonstrate the effectiveness of our approach in addressing scalability issues.

CCS CONCEPTS

• **Security and privacy** → **Formal security models**; **Software security engineering**; • **Software and its engineering** → **Model checking**; • **Computer systems organization** → *Self-organizing autonomic computing*.

KEYWORDS

Security Analysis, Model Checking, Runtime Security

ACM Reference Format:
Narges Khakpour and Charilaos Skandylas. 2024. Compositional Security Analysis of Dynamic Component-based Systems. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695499

1 INTRODUCTION

Today's software landscape moves towards heterogeneous and dynamic systems to cope with a frequently changing environment and user requirements. Due to the adapting nature and various sources of uncertainty in such systems, new security threats that are unknown at design time can arise. Such threats can lead to exposing a larger attack surface throughout a system's lifetime compared to systems with a static architecture and might introduce new opportunities for vulnerability exploitation. Further, cyber-attacks grow in scale, sophistication and frequency, as new vulnerabilities are introduced and novel exploitation techniques and strategies are discovered [2, 17, 45]. Conventional offline security analysis approaches might be of limited effectiveness, as they may fail to consider security threats introduced as the system's behavior and attack surface evolve at runtime.

Key Challenges. Changes in self-adaptive systems can happen at the architectural level or in the components' behavior. In systems where the system architecture and its vulnerabilities are dynamic and unknown a priori at design-time, *automated* security analysis and enforcement approaches are required to continuously analyse security at runtime and react to security-affecting changes to protect the system (C1). Online methods to analyze security [5, 6, 34, 54] often rely on runtime monitoring [5, 6, 54], where an observer monitors the executions of a software system to verify if its behavior adheres to a set of formal specifications [39]. These approaches only check the current execution to detect security violations which makes the analysis incomplete and thus are unsuitable for security threat analysis, since not all possible execution paths are verified (C2). Skandylas et al. [53] proposed a formal approach using model checking for runtime security analysis of component-based self-adaptive systems that considers the probabilistic attacker behaviour and runtime architectural changes but it suffers from scalability issues. Abstraction is a technique to reduce the analysis complexity by abstracting away details. One way to abstract away details is to perform the analysis at the architectural level where the internal behaviour of the components is omitted [32]. Even though architecture-level security analysis can reduce complexity, the scalability of formal techniques such as model checking is limited when analysing component-based systems due to the state space explosion problem [42] (C3). Modular verification is a common method to tackle complexity that aims to reduce the verification problem into verifying smaller subproblems. Modular analysis approaches have been used to analyse component-based systems in the past, e.g., incremental [9, 32, 38], on-the-fly [25, 51] and compositional verification [10], and assume-guarantee reasoning [46].

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10.
https://doi.org/10.1145/3691620.3695499

To address challenges C1-C3, there is a need for automated security analysis approaches that (i) consider uncertainties in the attacker behaviour, (ii) use modular analysis to scale up the reasoning to realistic systems, and (iii) perform the security analysis at runtime to cope with architectural changes or newly discovered avenues of attack.

The method proposed in [53] designs self-protecting systems using runtime threat analysis and addresses C1-C2, however, it suffers from scalability issues. To capture the uncertainty involved in the attacker behaviour, it employs a *probabilistic* modelling approach (C1), and given the system architecture and its vulnerabilities, it automatically builds and analyzes the security model(C2). This paper builds upon [53] and proposes a new *automated* approach for security threat analysis of component-based self-adaptive systems that utilizes a combination of techniques to tackle complexity, including *formal abstraction*, and *incremental security model construction and verification*.

Contributions. We propose a two-layer probabilistic security model. The high-level layer is an abstraction over the lower-level layer (See Figure 2). The lower layer formalizes a probabilistic attack scenario and the attacker's interactions with the component-based system. We propose a sound algorithm to build the abstract layer that automatically partitions the low-level layer into *fragments*, verifies probabilistic properties on them, and uses the results to build the probabilistic abstract layer. Verification is then performed on the high-level layer, to reduce the analysis complexity and improve scalability. We use the PRISM [29] model checker for verification, and PCTL (Probabilistic Computation Temporal Logic) [28] to specify properties. Our approach is sound, in the sense that model checking a subset of PCTL formulas against the high-level model and the low-level model leads to identical results.

To improve the scalability of our approach for runtime analysis, we employ incremental model update and verification. When a component-based system changes at runtime, often only a subset of its components are altered. We identify the components and fragments affected by a change, reconstruct and reverify the modified fragments of the low-level model, and update the corresponding parts of the high-level layer automatically. We have automated the whole approach and conducted experiments to study the effects of modularization and incremental update on the analysis complexity. The contributions of this paper include:

- [*Scalability, C3*] We exploit formal abstraction and modular analysis to automatically build a two-level formal security model. This model enables us to probabilistically reason about the security threats at an abstract level, either statically or at runtime, to tackle the analysis complexity. We employ a *probabilistic* approach to model and reason about uncertainty.
- [*Automation, C1*] The security model is automatically built from the system architecture described in an Architecture Description Language (ADL) [24] and the formal specification of its vulnerabilities. The security model is incrementally updated at runtime to reflect the changes, e.g., when new architectural changes or new avenues of attack are discovered.
- [*Soundness, C2*] We use model checking at runtime to analyze threats against the system. and formally prove that our approach is sound.
- [*Experiments*] We implement and apply our approach on a case study, and conduct experiments to study its performance. The results show that it reduces the complexity of the security analysis and offers an improvement in scalability.

An earlier version of this work, with detailed examples, algorithms, and an additional case evaluation is available at [52].

The rest of the paper is structured as follows. Section 2 presents a running example that we use for demonstrative purposes throughout this paper. Section 3 discusses the required preliminaries. Section 4 formally introduces our two-level security model. Section 5 details the modular generation, incremental update and verification of our model. Section 6 provides a proof of soundness for our abstraction. Section 7 discusses the evaluation of our approach. Section 8 discusses related work, and Section 9 concludes the paper.

2 RUNNING EXAMPLE

A component-based system is composed of a set of components where a component provides interfaces to be invoked by other components. Interfaces might contain security vulnerabilities, i.e., weaknesses that can be exploited to attack the system. Figure 1 shows an example system called InsecureStore that we use in this paper as a running example.The user can add, remove and lookup documents in this document handling system. InsecureStore includes five initial components: Frontend, Backend, UserMgr, FileMgr, and Database.

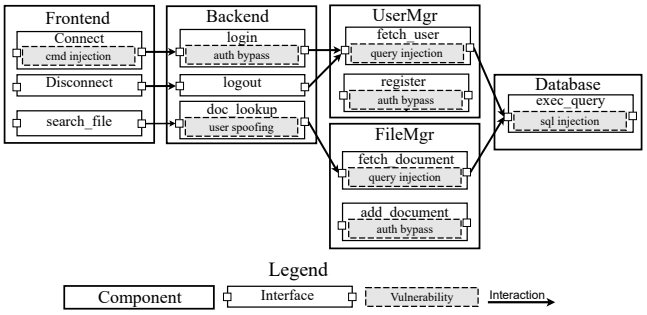
InsecureStore is capable of altering its architecture at runtime to better meet its goals. An example architectural change is shown in Figure 1(b) where a FileServer component is added to the system to improve its efficiency when dealing with a large number of document retrieval queries. Instead of retrieving documents from the Database component, the FileMgr component now requests a document from the newly added file server. The security characteristics of these two architectures are different. An attack exploiting sql_injection that was available in the architecture of Figure 1(a) by invoking the exec_query interface of the Database from the FileMgr component is no longer possible. However, a new path of exploitation has been uncovered in Figure 1(b), i.e., the get_document interface of the FileServer is callable and vulnerable to path injection.

3 PRELIMINARIES

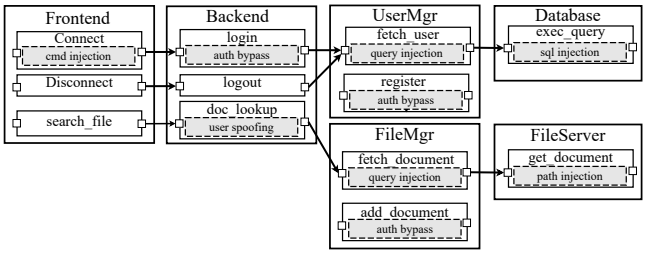
This section introduces probabilistic model checking and attack modeling for component-based systems.

3.1 Probabilistic model checking

The goal of verification is to check whether a system model $\mathcal{T}$ satisfies a specification ψ. Probabilistic model checking



(a) Initial InsecureStore architecture



(b) InsecureStore architecture after adaptation

Figure 1: Two possible InsecureStore architectures

is a technique to verify quantitative properties of a system that exhibits stochastic behavior [36]. We specify the system behaviour using *Probabilistic Transition Systems* (PTS).

Definition 1 (PTS)**.** A probabilistic transition system is a tuple $\mathcal{T} = (V, V_0, \mathbb{T})$ where V is a set variables, V_0 shows the initialization of the variables, and $\mathbb{T}$ is a set of probabilistic transitions. A transition is a tuple $(\phi, \mathbb{D}_U)$ where ϕ is the transition guard, a Boolean expression over V, and $\mathbb{D}_U$ is a discrete probability distribution function over the set of updates U.

Initially, $\mathcal{T}$ is in its initial state. A transition is fired if its guard is satisfied, and when fired the variables are updated according to its update functions. The semantics of PTS is defined as a *Discrete-Time Markov Chain* (DTMC).

Definition 2 (DTMC)**.** A DTMC is defined as a tuple $\mathcal{D} = (S, V, \Sigma, s_0, \mathbf{P}, \mathcal{L})$. S is the set of states defined over the set of variables V. Σ is a set of actions including the internal action τ. $s_0 \in S$ is the initial state. $\mathbf{P} : S \times S \to [0,1]$ is the transitions probability matrix where $\sum_{s' \in S} \mathbf{P}(s, s') = 1$ for all $s \in S$, and $\mathcal{L}$ is a labelling function that assigns to each state a set of atomic propositions over V that hold in that state.

PCTL [28] is a logic for probabilistic reasoning about temporal properties of a system with stochastic behaviour, for example: "Is the probability that the attacker compromises the network less than 0.5?"? This logic contains state formulas ϕ and path formulas ψ, the syntax of which is given by the following grammar:

$$\begin{aligned} \phi &::= \top \mid \alpha \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{P}_{\bowtie p}[\psi] \\ \psi &::= \mathsf{X}\,\phi \mid \phi_1 \mathsf{U} \phi_2 \mid \mathsf{F}\phi \end{aligned}$$

where $\top$ denotes "true", α is a proposition, $\bowtie \in \{<, \leq, >, \geq\}$, and $p \in [0,1]$. The operators X ("next"), U ("until") and F ("eventually") are the standard temporal operators. Informally, $\mathsf{P}_{\bowtie p}[\psi]$ states that the path formula ψ will be satisfied with a probability meeting the bound $\bowtie p$. $\mathsf{P}_{=?}[\psi]$ denotes computing the probability of satisfying the property ψ. We write $s \vDash \psi$ to indicate that state s of $\mathcal{D}$ satisfies the formula ψ, and write $\mathcal{D} \vDash \psi$ if $s_0 \vDash \psi$. A property is satisfied by a PTS, if and only if it is satisfied by its semantics described as a DTMC.

3.2 Component-based System Attack Modeling

We define a software architecture as as a collection of components and interactions between them. Each component satisfies parts of the system's functionality which is exposed to the rest of the system via a number of interfaces that can be invoked to facilitate communication between the components.

A security-aware architecture [53] is a software architecture enhanced with security-related information including the vulnerabilities associated with each component, the interfaces that can be invoked to exploit those vulnerabilities and how vulnerabilities can be combined to form more complex attacks that allow the attacker to compromise the system further. Figure 1 depicts two possible security-aware architectures of InsecureStore. A security-aware architecture $A = \langle C, R \rangle$ is a pair where a component $c \in C$ is defined as $(I, \mathcal{V})$, I is the set of c's interfaces used for communication with other components, and $\mathcal{V} : I \to 2^V$ is a function mapping each interface $i \in I$ to a possibly empty set of vulnerabilities in V, and V is the set of all vulnerabilities. Further, R is a set of component interactions. An interaction r is a triple (c', i, c) where $c, c' \in C$ are interacting components and $i \in I$ is an interface provided by c. This means that the component c' invokes the interface i of the component c.

We rely on an architectural model of the system that is maintained at runtime to support incremental updating of our security model. The architectural model contains the concrete security-aware architecture, component types, vulnerabilities, attack rules to chain vulnerabilities to create attacks and their effects. Vulnerabilities, and the rules for complex attacks and security impact are modeled as opaque architectural properties, and include the required attack probabilities. When the system architecture changes, the corresponding changes are reflected to the architectural model of the system automatically.

A logical attack graph is used to illustrate and analyze the exploitation of security weaknesses and vulnerabilities to compromise a system. Figure 4(a) shows an example attack graph. A logical attack graph shows different strategies to reach a *final goal*, and comprises three types of nodes: *primitive nodes* (rectangles) that state the initial known facts about the attack, *derived nodes* (diamonds) which correspond to derived knowledge, and *rule nodes* (ellipses) that are rules used to derive new derived nodes via resolution [43]. When the preconditions of a rule node hold (i.e., its predecessors),

the rule is enabled and the attacker can gain the rule's consequence (i.e., its successor), which is a derived node. To generate the logical attack graph, we use a set of rules, contained in the architectural model, to specify vulnerability exploitation, the security-aware architecture and the attack goal, which are then fed to a tool called MulVAL [44] to generate a logical attack graph.

4 SECURITY MODEL

4.1 Scope and Overview

Scope. Our approach targets component-based systems with dynamic software architectures and can be used by designers and operators of such systems to analyze their security. This approach can be deployed alongside the system at runtime to continuously monitor and evaluate its security. Although in this paper we primarily focus on self-adaptive component-based systems, our approach is also applicable to other types of software systems, the architectures of which can be modeled as a graph of components and connectors [18], whose vulnerabilities can be identified. Examples of such architectures include microservice and container-based systems. We consider threats that are modeled as exploitable vulnerabilities in the components' interfaces. New threats to the system can arise from architectural changes that introduce or remove attack paths through component invocations or from discovering new vulnerabilities. Further, our approach offers a viable method for analyzing security in systems with known vulnerabilities that have not been remediated. Note that recent research results [48, 57] and industry security reports [12, 56] show that systems are often deployed with known vulnerabilities for various reasons.

We employ probabilities to capture uncertainties in the attacker and system behavior. Sources of uncertainty include, among others: (i) the attacker's skill level and budget to carry out attacks, (ii) the attacker's initial knowledge of the system and exploitation capabilities, (iii) the complexity of the exploitation, (iv) the effectiveness of defense mechanisms in detecting or preventing attacks, and (v) interactions between exploits or system behaviors that depend on the runtime state, e.g., they are timing/memory order dependent.

Security Model Overview. We employ a two-tier model shown in Figure 2 to describe and analyse the security of a system. The bottom layer, called the *low-level layer*, describes the attacker behaviour and her interactions with the system components, i.e., actions by the attacker or by the system that contribute to attacks and the consequences of those actions on the system or attacker behaviour. The top layer is the *high-level* layer that abstracts away the details of the low-level layer. This layer is constructed based on the verification results of the low-level layer that describe under which conditions security violations can occur and the consequences of such violations. Security properties are then verified against the high-level abstract model, instead of the large low-level model, to tackle analysis complexity. We construct an initial model compositionally, and incrementally update it at runtime to reflect changes in its architecture and exploitability, allowing us to modularly reason about security as the system evolves.

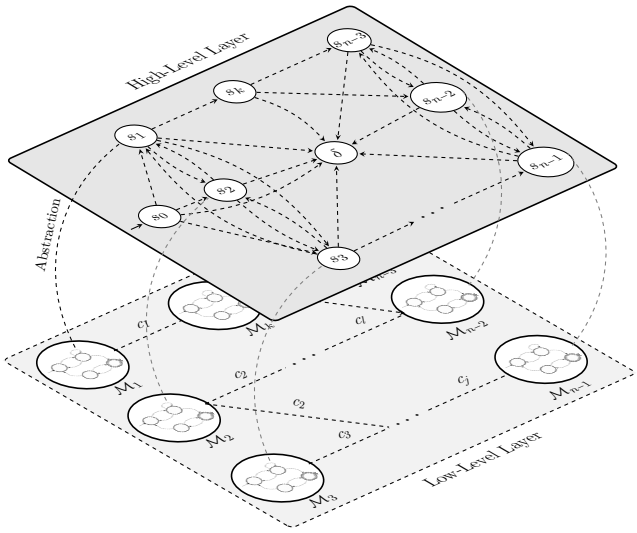
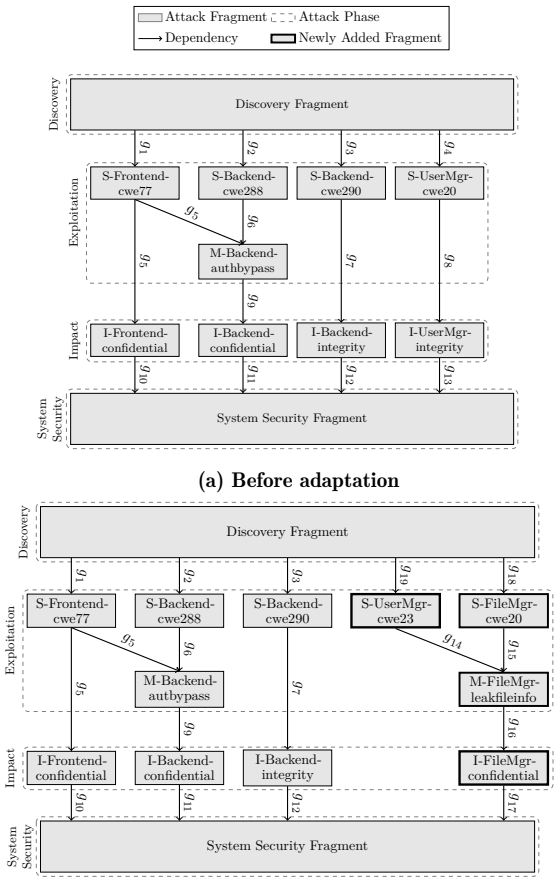


Figure 2: The Two-Level Security Model where $\mathcal{M}_i$ is a fragment model, δ is a deadlock state showing that the attacker gave up on the attack

4.2 Low-level Layer

An attack scenario describes different strategies an attacker may follow to reach a specific goal. We decompose an attack scenario into four *attack phases*: *discovery*, *exploitation*, *impact* and *system security*. Each attack phase is further decomposed into a number of *attack fragments*. An attack fragment is a set of actions taken by the attacker to reach a possibly intermediate goal. Figure 3(a) and Figure 3(b) show two attack scenarios where the boxes correspond to attack fragments, the dotted areas correspond to phases, and the arrows between the boxes show the dependencies between attack fragments. In the discovery phase, the attacker identifies the interactions among the system components to gain information about the active components, their exposed interfaces and their interactions. The exploitation phase describes the process by which the attacker interacts with the system under attack. It comprises three steps: (i) vulnerability discovery, i.e., identifying the vulnerable interfaces and available exploits of the active components, (ii) vulnerability exploitation, i.e., exploiting a single vulnerable interface which we call a *single-exploit* attack, and (iii) combining multiple exploits to perform more complex attacks, which we call *multi-exploit* attacks. The impact phase models the impact of exploitation on each component based on which fundamental security property among confidentiality, integrity and availability is violated. The system security phase, describes the conditions upon which the whole system is rendered non-functional.

We employ a capability-based approach to model the attacker behavior. An attacker capability is the ability to perform a certain set of actions, and is obtained by succeeding in



(a) Before adaptation

(b) After adaptation

Figure 3: Attack phases and the corresponding attack fragments before and after the adaptation where an element with the prefix 'S-', 'M-', 'I-' respectively shows a single-exploit, multi-exploit and security impact fragment. Goal capabilities, that serve as prerequisites to fragments are prefixed with g.

performing the actions in the associated attack fragment. For example, in order for the attacker to gain unauthorized entry to the system in our running example, she needs a capability to exploit the Backend component via an authentication bypass vulnerability and once successful in bypassing authentication, she can make use of Backend as a staging point to gain further capabilities. When an attacker gains a capability, i.e., she succeeds in achieving a sub-goal, her exploitation capabilities grow. Capabilities show the extent to which the attacker has succeeded in taking over the system and enable her future actions by defining her available exploitation options.

An attack fragment f comprises a set of related activities that are carried out to gain a specific capability called the *fragment goal*. The capabilities are of three types: *prerequisite capabilities* denoted by $\mathcal{C}_P$, *internal capabilities* denoted by $\mathcal{C}_I$, and a *goal capability* denoted by g_f. Internal capabilities are local to the fragment, while a goal capability can be a prerequisite capability of another fragment. Prerequisite capabilities serve as the requirements for gaining internal and/or goal capabilities, while a goal capability is the result of achieving the goal of the attack fragment. An attack fragment might have multiple prerequisite capabilities but will only have a single goal capability. For example, the attack fragment S-Frontend-cwe77 in Fig. 3 is used to gain a capability g_5 that enables the attacker to exploit the connect interface on the Frontend component to perform command injection on it. Command injection in turn allows the attacker to perform a multi exploit attack to achieve authentication bypass on the Backend component, i.e., to gain the capability g_9 in M-Backend-authbypass, in addition to affecting the *confidentiality* of the Frontend component. A fragment name is composed of its fragment type, the affected component and its goal name where the goal name shows either the CWE (Common Weakness Enumeration) [41] identifier where applicable, or the capability gained by the attacker otherwise.

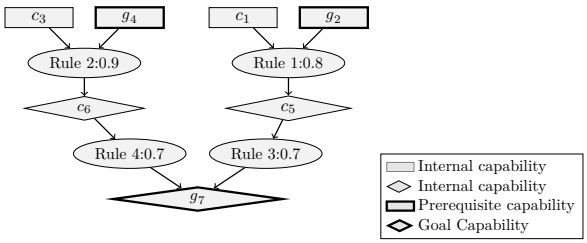
An attack fragment is specified as a PTS that models how the attacker actions affect the security of the system components. We refer to that PTS as a *fragment model*.

Definition 3 (Fragment model)**.** An attack fragment f is modeled using a PTS $\mathcal{M} = \langle V, V_0, \mathbb{T} \rangle$ where:

- $V = \mathcal{C}_P \cup \mathcal{C}_I \cup \{g, \delta\}$ where $\mathcal{C}_P$ and $\mathcal{C}_I$ are sets of Boolean variables used to show the prerequisites and the internal capabilities, g is the fragment goal capability, and δ is a boolean variable to indicate whether the attacker quit without achieving the goal g,
- each internal/goal capability c is associated with a probabilistic transition $(\phi_c, \mathbb{D}_U)$ in $\mathbb{T}$ where ϕ_c is the prerequisites to gain c, and $U = \{\{c := \top\}, \varnothing, \{\delta := \top\}\}$ is the set of possible updates by this transition. The transition states that when the prerequisites to gain a capability c hold (formula ϕ_c), one of the three updates in U is applied with a probability: either (i) the attacker succeeds and c is gained (i.e., the update $\{c := \top\}$), or (ii) the attacker fails and can retry (i.e., the update $\varnothing$), or (iii) the attacker gives up on trying the attack in future (i.e., the update $\{\delta := \top\}$.

Only transitions that lead to the attacker achieving the goal have a probability of giving up greater than zero. That is based on the assumption that the attacker gives up on the goal altogether rather than on performing a specific attack action. The internal actions of a fragment are local to it, i.e., they do not affect the rest of the fragments and are not considered in the high-level model. Thus, the precise internal capability that the attacker failed to gain and gave up is not relevant, as it does not affect the behavior of the high-level model or of any other fragments.

Figure 4(b) shows the transitions of a fragment model. A transition t is written as $[t]\ \phi_c \rightarrow \mathrm{p}_1 : \{c := \top\} + \mathrm{p}_2 : \{\delta := \top\} + \mathrm{p}_3 : \varnothing$, which states when the condition ϕ_c holds, then with probability p_1 the transition a will be taken to set c to $\top$, otherwise δ will be set to $\top$ with probability p_2, or no effect will occur with probability p_3 (i.e., the attack failed). Note that each fragment has a unique goal and the internal capabilities of two attack fragments are disjoint, i.e., $g \neq g'$, and $\mathcal{C}_I \cap \mathcal{C}'_I = \varnothing$ where $f \neq f'$. However, the goal capability



(a) Fragment graph

[Rule1] c_1 & g_2 & not c_5 → 0.8 : $(c_5 = \top)$ + 0.2: $\varnothing$
[Rule2] c_3 & g_4 & not c_6 → 0.9 : $(c_6 = \top)$ + 0.1: $\varnothing$
[Rule3] not δ_f & c_5 & not g_7 → 0.7 : $(g_7 = \top)$
+ 0.15 : $(\delta_f = \top)$
+ 0.15: $\varnothing$
[Rule4] not δ_f & c_6 & not g_7 → 0.7 : $(g_7 = \top)$
+ 0.15 : $(\delta_f = \top)$
+ 0.15: $\varnothing$

(b) Fragment model ($\mathcal{M}_f$)

[] not δ_f & not g_7 & $(g_2$ or $g_4)$ → 0.8235:$(g_7 = \top)$ + 0.1765:$(\delta_f = \top)$

(c) High-level layer $\mathcal{H}$ (Partial)

Figure 4: Transformation of a fragment graph into a fragment model, and its high-level abstraction

of a fragment can be a prerequisite capability of another fragment which establishes a dependency between the two fragments. *Further, we assume the attacker only attempts to gain a capability if she does not already hold it.*

Definition 4 (Low-level Layer)**.** Let $F = \{f_0, \ldots, f_n\}$ be the set of attack fragments of an attack scenario. The low-level layer model $\mathcal{M}$ is defined as the parallel composition of its fragment models, i.e., $\mathcal{M}_0 \parallel \cdots \parallel \mathcal{M}_n$ where $\mathcal{M}_i, 0 \leq i \leq n$ is the fragment model of f_i.

4.3 High-level Layer

The high-level layer abstracts away the internal behavior of the attack fragments, i.e., the internal capabilities of the attack fragments in the low-level layer, and the model only captures the goal capabilities and prerequisite capabilities.

Definition 5 (High-Level Model)**.** Let $\{f_0, \ldots, f_n\}$ be the set of attack fragments in the low-level layer, and $\mathcal{M}_i = (V_i, V_{0i}, \mathbb{T}_i)$ be the fragment model of $f_i, 0 \leq i \leq n$ where $V_i = \mathcal{C}_{P,i} \cup \mathcal{C}_{I,i} \cup \{g_i, \delta_i\}$. The high-level model is defined as a PTS $\mathcal{H} = (V, V_0^h, \rightarrow)$ where:

- $V = \bigcup_{0 \leq i \leq n} V_i \backslash \mathcal{C}_{I,i}$ is the union of the variables of all fragments except for their internal capabilities $\mathcal{C}_{I,i}$,
- each goal capability g_i belonging to a fragment f_i is associated with a probabilistic transition $(\phi_{g_i}, \mathbb{D}_U)$ where ϕ_{g_i} is the prerequisite to gain the fragment goal g_i, and $U = \{\{g_i := \top\}, \{\delta_i := \top\}\}$. This transition states when the prerequisites to gain the capability g_i hold (formula ϕ_{g_i}), then either (i) the attacker succeeds to reach the goal g_i, or (ii) the attacker gives up at some stage of the attack. $\mathbb{D}_U(\{g_i := \top\})$ is the probability of reaching g_i in its corresponding fragment model $\mathcal{M}_i$, and $\mathbb{D}_U(\{\delta_i := \top\}) = 1 - \mathbb{D}_U(\{g_i := \top\})$.

5 AUTOMATED SECURITY MODEL CONSTRUCTION AND INCREMENTAL UPDATE

In this section, we discuss how we automatically partition an attack scenario into fragments, modularly construct the model and incrementally update it.

5.1 Automated Modular Model Construction

5.1.1 Low-level Layer Partitioning and Construction. To construct the low-level layer *automatically*, we first generate a logical attack graph [44] to describe each attack fragment, called a *fragment graph*, and then transform this graph into a fragment model $\mathcal{M} = (V, V_0, \mathbb{T})$ where V is the set of derived nodes and primitive nodes in the fragment graph that represent the attacker capabilities. All primitive capabilities are initialized to true in V_0 while the derived capabilities have the initial values of false. This is due to the fact that the attacker already holds the primitive capabilities, and follows the attack strategy to gain the derived capabilities. According to the semantics of rule nodes, when the attacker holds all the predecessors of a rule node, she can apply the rule and gain its successor derived node. Thus, we translate each rule node r with the successor node c to a transition $\langle \phi_c, \mathbb{D}_U \rangle$ where ϕ_c is the conjunction of the capabilities associated with r's predecessors, and $\mathbb{D}_U$ is defined according to Definition 3 where the capability c is the rule successor.

Example 1. Figure 4(a) shows a fragment graph, and Figure 4(b) is its corresponding model. The label of a rule node shows its name separated by ":" from its success probability. The goal capability of this fragment is the final node g_7. For instance, Rule 2 states that when the capabilities c_3 and g_4 (i.e., Rule 2's predecessors) are held by the attacker, and she does not hold the capability c_6 (i.e., Rule 2's successor), she can use this rule and may gain c_6 with probability 0.9 or fail.

The low-level layer generation is carried out modularly using Algorithm 1 which partitions an attack scenario into disjoint fragments and generates their models in parallel automatically. Given an architectural model A, the algorithm first generates a generic attack graph $\mathcal{G}$ that includes all possible attack patterns (line 2). The attack patterns correspond to chains of rule applications that link a fragment's prerequisite capabilities to its goal capability. An attack pattern is represented by a Horn clause that specifies an attack rule. The rules are parametric and if a rule's parameters can be valuated in a way to match the prerequisites of a fragment, we can call it a matched attack pattern and the instantiated attack pattern constitutes an attack fragment or a part of it.

To perform the instantiation, we employ an exhaustive matching algorithm that will try to match all possible generated goal capabilities of the previous phases with prerequisites of the fragments of the next phase. Then, we use the architectural model A and the generic attack graph $\mathcal{G}$ to identify the

Algorithm 1: Modular Fragment Model Construction

```
   Input: Architectural Model A
   Output: Set of fragment models X
   // The set of all generated fragment graphs
1  pFragGraphs = ∅, X = ∅
   // Create the generic attack graph
2  G=CreateGenericAttackGraph(A)
   // Phase-by-phase fragment generation
3  for (depth = 0; depth < maxDepth; depth++) do
       // find all depth-th level fragments given pFragGraphs
4      F=FindNewFragments(depth,G,pFragGraphs)
       // generate the fragment attack graphs in parallel
5      ParallelGraphGen(F,G,pFragGraphs)
6  end
7  for fragGraph ∈ pFragGraphs do in parallel
       // build the fragment model from the fragment graph
8      M_i = ConstructFragModel(fragGraph,A)
9      X = X ∪{M_i}
10 end
11 return X
12 Function ParallelGraphGen(F,G,pFrags) is
13     for f ∈ F do in parallel
           // generate the formal specification of the fragment
14         spec = GenerateFormalSpec(f,G)
           // generate the fragment attack graph
15         fragGraph=RunGraphGen(f,spec)
16         pFrags=pFrags ∪{fragGraph}
17     end
18 end
```

fragments of each phase (line 4) and generate their fragment graphs phase-by-phase (lines 3-6). The fragments of a phase are identified by traversing $\mathcal{G}$ to find matching fragments F whose prerequisites fragments have been generated, i.e., those fragments that match the attack pattern specified by the attack rules used to generate $\mathcal{G}$. If a matching component allocation is found, it means that the attack is possible in the security-aware architecture and hence a fragment must be created for it. The fragment graphs of F are then generated in parallel using the function ParallelGraphGen (lines 13-19). This function automatically generates the specification of each fragment based on the specification of the generic attack patterns specified in first order logic in a similar fashion to [53] (line 15), and then feeds the specification to a logical attack graph generator to generate its graph (line 16). Once all the fragment graphs are built, they are transformed to fragment models in parallel, as discussed earlier (lines 7-10).

5.1.2 Automatic High-Level Model Construction via Formal Abstraction. Algorithm 2 gives the steps to construct a high-level model from a set of fragment models. The algorithm receives as input the output of Algorithm 1, i.e., a set of fragment models. For each fragment model $\mathcal{M}_i$, we first retrieve its associated goal g_i (line 3), add the goal to the set of variables V (line 4), instantiate said variables (line 5), and model check the property, $\mathsf{P}_{=?}[F\ g_i]$ to determine the probability that the attacker succeeds in gaining the fragment goal (line 6). Informally, this is equivalent to calculating the probability ρ_{g_i} that the attacker will eventually succeed in gaining g_i. We generate a transition for each fragment at the high-level model that either leads to gaining the fragment goal with a probability, or giving up. We add the transition to the set of transitions $\mathbb{T}$ (line 9). The transition guard is a conjunction of the attacker not having previously succeeded or given up on the goal (i.e., $\neg g_i$ and $\neg\delta_i$), and the set of prerequisites that need to hold before the goal can be attempted returned by the function GetPrerequisites. The transition's probability to take the update where the attacker gains the goal ($g_i = \top$) is ρ_{g_i}, while with probability 1 - ρ_{g_i} the attacker gives up ($\delta_i = \top$). The function MakeTransition receives as input the transition guard, the fragment goal g_i and the probability that the attacker will succeed in gaining the goal ρ_{g_i} and generates a transition including the δ_i update and its corresponding probability.

Algorithm 2: High-Level Model Construction

```
   Input: A set of fragment models X
   Output: The high-level model H
1  T = ∅, V = ∅
   // For each fragment model
2  for M_i ∈ X do in parallel
       // Retrieve the fragment model's goal
3      g_i = GetFragmentGoal(M_i)
4      V = V ∪{g_i} ∪ {δ_i}
5      V_0 = V_0 ∪{g_i = ⊥} ∪ {δ_i = ⊥}
       // Compute the probability the goal will be met
6      ρ_{g_i}= ModelCheck(M_i,'P_{=?}[F g_i]')
       // Retrieve fragment graph for M_i
7      fragGraph = getFragmentGraph(M_i)
       // Retrieve the fragment goal's prerequisites
8      φ_{g_i} = ¬g_i ∧ ¬δ_i∧GetPrerequisites(g_i,fragGraph)
       // Add the goal transition to the high-level model
9      T = T ∪{MakeTransition(φ_{g_i}, g_i, ρ_{g_i})}
10 end
11 H = (V, V_0, T)
12 return H
```

Example 2. Figure 4 (c) shows the corresponding transition of the fragment shown in Figure 4(b) in the high-level layer. The fragment goal g_7 can be reached via two paths: (i) Rule 1 and Rule 3, or (ii) Rule 2 and Rule 4. The prerequisites for the first path are c_1 and g_2, and the prerequisites for the second path are c_3 and g_4. g_2 and g_4 are fragment goals of its predecessors fragments, while c_1 and c_2 are internal capabilities that always hold and can be ignored. Hence, the guard to gain g_7 is holding g_2 or g_4, and not holding g_7 or giving up. The probability 0.8235 is the model checking result of $\mathsf{P}_{=?}[F\ g_7]$ against the fragment model shown in Figure 4(b).

When an architectural change occurs, a subset of the components and interactions are often altered, added or removed. Thus, a part of the system model and consequently, a part of the attacker behavior will remain unaffected, while the behavior related to the modified part of the system architecture will evolve. Fig. 3(a) shows the state of the low-level layer that evolves to Fig. 3(b) after an adaptation that adds a File-manager component to the system. There are seven fragments in Fig. 3(a) that belong to the exploitation and impact phases fragment models for Frontend, and Backend. These fragments models have not been altered by the adaptation, rather an extra vulnerable component has been added which has introduced three new fragments (See Fig. 3(b)). Therefore, instead of re-generating the whole low-level model, we update the model by updating the affected fragments that correspond to the components affected by an adaptation. Since the high-level model is constructed by verifying properties on the

$[]\ \top \to p_5 : (g_5{=}\top) + (1{-}p_5) : (\delta_{g_5}{=}\top)$
$[]\ \top \to p_6 : (g_6{=}\top) + (1{-}p_6) : (\delta_{g_6}{=}\top)$
$[]\ \top \to p_7 : (g_7{=}\top) + (1{-}p_7) : (\delta_{g_7}{=}\top)$
$[]\ \top \to p_8 : (g_8{=}\top) + (1{-}p_8) : (\delta_{g_8}{=}\top)$
$[]\ g_5\ \&\ g_6 \to p_9 : (g_9{=}\top) + (1{-}p_9) : (\delta_{g_9}{=}\top)$
$[]\ g_5 \to p_{10} : (g_{10}{=}\top) + (1{-}p_{10}) : (\delta_{g_{10}}{=}\top)$
$[]\ g_9 \to p_{11} : (g_{11}{=}\top) + (1{-}p_{11}) : (\delta_{g_{11}}{=}\top)$
$[]\ g_7 \to p_{12} : (g_{12}{=}\top) + (1{-}p_{12}) : (\delta_{g_{12}}{=}\top)$
$[]\ g_8 \to p_{13} : (g_{13}{=}\top) + (1 - p_{13}) : (\delta_{g_{13}}{=}\top)$

(a) Before Adaptation

$[]\ \top \to p_5 : (g_5{=}\top) + (1{-}p_5) : (\delta_{g_5}{=}\top)$
$[]\ \top \to p_6 : (g_6{=}\top) + (1{-}p_6) : (\delta_{g_6}{=}\top)$
~~$[]\ \top \to p_7 : (g_7{=}\top) + (1{-}p_7) : (\delta_{g_7}{=}\top)$~~
$[]\ \top \to p_8 : (g_8{=}\top) + (1{-}p_8) : (\delta_{g_8} = \top)$
$[]\ g_5\ \&\ g_6 \to p_9 : (g_9{=}\top) + (1{-}p_9) : (\delta_{g_9}{=}\top)$
$[]\ g_5 \to p_{10} : (g_{10}{=}\top) + (1{-}p_{10}) : (\delta_{g_{10}}{=}\top)$
$[]\ g_9 \to p_{11} : (g_{11}{=}\top) + (1{-}p_{11}) : (\delta_{g_{11}}{=}\top)$
~~$[]\ g_7 \to p_{12} : (g_{12}{=}\top) + (1{-}p_{12}) : (\delta_{g_{12}}{=}\top)$~~
$[]\ g_8 \to p_{13} : (g_{13}{=}\top) + (1 - p_{13}) : (\delta_{g_{13}}{=}\top)$
$[]\ \top \to p_{14} : (g_{14}{=}\top) + (1{-}p_{14}) : (\delta_{g_{14}}{=}\top)$
$[]\ \top \to p_{15} : (g_{15}{=}\top) + (1{-}p_{15}) : (\delta_{g_{15}}{=}\top)$
$[]\ g_{14}\ \&\ g_{15} \to p_{16} : (g_{16}{=}\top) + (1{-}p_{16}) : (\delta_{g_{16}}{=}\top)$
$[]\ g_{16} \to p_{17} : (g_{17}{=}\top) + (1{-}p_{17}) : (\delta_{g_{17}}{=}\top)$

(b) After Adaptation

Figure 5: Before and after adaptation high-level models

low-level model, changes in the low-level model need to be propagated to the high-level model. See [52] for more details about the incremental model update process.

Fig. 5 shows the high-level model for the configurations in Fig. 3. The high-level model is updated with the four new capabilities while the old capabilities associated with the UserMgr component that are no longer present are removed. In Fig. 5(b), additions are highlighted in bold, and removals are shown in strikethrough. In the figure, all variables in the model are initialized to false and the probability associated with a goal capability g_i is represented by p_i. *For readability, we omitted the negation of give up from the transitions guards.* For instance, the last transition in Fig. 5(b) basically represents the following transition: $[]\ \text{not}\ \delta_{\mathbf{g_{17}}} \& \mathbf{g_{16}} \to \mathbf{p_{17}} : (\mathbf{g_{17}} = \top) + (\mathbf{1} - \mathbf{p_{17}}) : (\delta_{\mathbf{g_{17}}} = \top)$.

5.2 Verification

We use our security model to perform quantitative security threat analysis via probabilistic model checking. To tackle the analysis complexity, verification of security properties is performed against the high-level model. We express our security properties in $\text{PCTL}_{\setminus X}$ which is a subset of PCTL with no next operator. We define an atomic proposition per goal capability or give up variable (See Definition 5). Misusing notation, we represent the atomic proposition that matches a goal capability by the fragment from which the goal capability is derived. When a fragment is prefixed by 'Q' then we refer to the atomic proposition that matches the give_up variable of the fragment. For example, the atomic proposition ExploitedFrontendCWE77 represents the attacker gaining g_5 in InsecureStore. We also define systemDown, a Boolean expression over atomic propositions which is defined per system and represents the condition upon which the system can be considered fully compromised to the extend that it can no longer remain operational. We can express properties to assess different security risks, e.g., (i) to determine the success probability of different attacks to analyse their risks, (ii) the probabilistic temporal dependencies among different threats to identify the severity of exploiting vulnerabilities, (iii) the probability of the system as a whole going down, etc. A few examples of relevant properties on InsecureStore include:

(1) $\mathsf{P}_{=?}[\mathsf{F}\ \text{systemDown}]$ to determine the probability that the system will be compromised.
(2) $\mathsf{P}_{>0.8}[\mathsf{F}\ \text{ExploitedFrontendCWE77}]$ to check if the probability that a the attacker will finally gain g_5 by exploiting cwe77 on the Frontend component is greater than 0.8.
(3) $\mathsf{P}_{=?}[\neg\mathsf{F}\ \neg(\text{QAttackBackendAuthbypass} \Rightarrow \mathsf{F}\ \text{systemDown})]$ to find the probability that the system will be eventually be brought down even though the attacker gave up on gaining g_9.

6 SOUNDNESS

We prove that our approach is sound, i.e., the verification results of a $\text{PCTL}_{\setminus X}$ formula at the high-level model and low-level model are identical. Let $\mathcal{D}_\ell = (S_\ell, V_\ell, \Sigma, s_{0\ell}, \mathbf{P}_\ell, \mathcal{L}_\ell)$ be the semantics of the low-level model in terms of a DTMC, and $\mathcal{D}_h = (S_h, V_h, \Sigma, s_{0h}, \mathbf{P}_h, \mathcal{L}_h)$ be the semantics of the high-level layer where $V_h \subseteq V_\ell$. We will prove that for all $\text{PCTL}_{\setminus X}$ formulas ψ defined over V_h, $\mathcal{D}_\ell \vDash \psi$ if and only if $\mathcal{D}_h \vDash \psi$. To prove this, we first show that $\mathcal{D}_\ell$ and $\mathcal{D}_h$ are probabilistic weak bisimilar [3].

Definition 6 (Probabilistic Weak Bisimulation [3])**.** Let $\mathcal{Y}(s, \tau^* a \tau^*, t)$ be a function that shows the probability of reaching the state t from s using internal actions τ and the action a. A weak probabilistic bisimulation over the DTMC $\mathcal{D} = (S, V, \Sigma, s_0, \mathbf{P}, \mathcal{L})$ is an equivalence relation $R \subseteq S \times S$ such that for all $(s_1, s_2) \in R$, $\lambda \in \Sigma \setminus \{\tau\} \cup \{\varepsilon\}$, and all equivalence classes $C \in S_R$, $\mathcal{Y}(s_1, \tau^* \lambda \tau^*, C) = \mathcal{Y}(s_2, \tau^* \lambda \tau^*, C)$.

Note that ε is an empty trace, $\varepsilon\pi = \pi$, and S_R shows the set of equivalence classes defined by R. This relation informally states that two probabilistic weak bisimilar states have the same probability to reach an equivalence class. We say two DTMCs $\mathcal{D}_i = (S_i, V_i, \Sigma_i, s_{0i}, \mathbf{P}_i, \mathcal{L}_i)$, $i \in \{1, 2\}$ are probabilistic weak bisimilar, denoted by $\mathcal{D}_1 \approx_p \mathcal{D}_2$, if and only if there exists a probabilistic weak bisimulation relation R defined over their disjoint union such that $(s_{01}, s_{02}) \in R$. It has been proven that two probabilistic weak bisimilar DTMCs coincide on $\text{PCTL}_{\setminus X}$ [4]:

Theorem 1. Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two probabilistic weak bisimilar DTMCs. For any $\text{PCTL}_{\setminus X}$ formula ψ, $\mathcal{D}_1 \vDash \psi \Leftrightarrow \mathcal{D}_2 \vDash \psi$.

Lemma 1. Let $\mathcal{M} = \mathcal{M}_0 \parallel \cdots \parallel \mathcal{M}_n$ be a low-level model, $\mathcal{D}_i$ be the semantics of $\mathcal{M}_i$, $0 \le i \le n$, and $\mathcal{D}_\ell$ be the semantics of $\mathcal{M}$. The fragment $\mathcal{D}_i$ is probabilistic weak bisimilar to the low-level model $\mathcal{D}_\ell$, i.e., $\mathcal{D}_i \approx_p \mathcal{D}_\ell$.

Lemma 2. The fragment $\mathcal{D}_i = (S_i, V_i, s_{0i}, \mathbf{P}_i, \mathcal{L}_i)$ and the abstract layer $\mathcal{D}_h = (S_h, V_h, s_{0h}, \mathbf{P}_h, \mathcal{L}_h)$ are probabilist weak bisimilar, i.e., $\mathcal{D}_h \approx_p \mathcal{D}_i$.

Proof Sketch. To prove this theorem, we should find a relation $R \subseteq S_i \times S_h$ that is a witnessing probabilistic weak bisimulation relation. We will show that the following relation is a witnessing weak bisimulation relation and $(s_{0i}, s_{0h}) \in R$:

$$R = \{\langle s, s' \rangle \mid s =_{\{\delta_i, g_i\}} \ s'\}.$$

where δ_i and g_i are respectively the give up and goal capability of fragment $\mathcal{D}_i$. The proof is similar to that of Lemma 1, with the difference that we use the fact that the sum of the probabilities of all paths to a goal state s' where $g_i \in \mathcal{L}_i(s')$ is equal to $\mathsf{P}_{=?}[F\ g_i]$ according to the semantics of PCTL. □

Theorem 2. The low-level model $\mathcal{D}_\ell$ and the abstract layer $\mathcal{D}_h$ are probabilistic weak bisimilar, i.e., $\mathcal{D}_h \approx_p \mathcal{D}_\ell$.

Proof. Since probabilistic weak bisimulation is an equivalence relation, it is also transitive. Hence, we conclude $\mathcal{D}_h \approx_p \mathcal{D}_\ell$ from $\mathcal{D}_h \approx_p \mathcal{D}_i$ (Lemma 2) and $\mathcal{D}_i \approx_p \mathcal{D}_\ell$ (Lemma 1). □

Theorem 3 (Soundness)**.** For all $\text{PCTL}_{\setminus X}$ formulas ψ, $\mathcal{D}_\ell \models \psi$ iff $\mathcal{D}_h \models \psi$.

Proof. Since $\mathcal{D}_\ell \approx_p \mathcal{D}_h$ according to Theorem 2 and probabilistic weak bisimulation coincides with equivalence for $\text{PCTL}_{\setminus X}$ according to Theorem 1, we can conclude $\mathcal{D}_\ell \models \psi$ iff $\mathcal{D}_h \models \psi$ for any $\text{PCTL}_{\setminus X}$ formula ψ. □

Theorem 3 proves the soundness of our approach.

7 EVALUATION

To study the scalability of our approach, we have conducted a set of experiments on ZNN[1], a well known self-adaptive exemplar (See [52] for more experiments on an additional case study). It is an implementation of a news service implemented in a client-server architecture. ZNN's components include a set of servers managed by a load balancer that acts as a middle point to facilitate communication between the servers and clients. News are stored in a database and fetched periodically by the servers. The adaptation operations include enabling or disabling a server, and increasing or decreasing fidelity of one or multiple servers. For vulnerability scanning, we employed Progpilot4[2] and OpenVAS[3] to scan ZNN's attack surface which collectively discovered 54 vulnerabilities. ZNN is equipped with multiple adaptation strategies. We selected three among them that cover a varying number of changes to the system architecture to use in our experiments. ImproveOverallFidelity (**A1**) increases the system fidelity by increasing the fidelity of all servers whose fidelity is below a threshold. QuickReduceOverallCost (**A2**) will remove a server from the system if its cost is above a certain threshold. VariedReduceResponseTime (**A3**) adds a server to the server pool and then lowers fidelity to reduce response time.

Experiment Design. We consider three different types of security analysis. In the first type, called *sequential* analysis, there is no modularity, i.e., the analysis is performed on the low-level model. This base analysis is similar to the analysis performed in [53]. In the *modular* analysis case, we follow the modular model construction and analysis approach presented in Section 5. We do not, however, perform any incremental model update and verification. In the *incremental* analysis case, we perform modular model generation during the system initialization and perform incremental model update and verification upon an architectural change at runtime. We design experiments to answer the following questions:

RQ1: How do the three analyses compare in terms of scalability and performance overhead?
RQ2: How do the different steps of the approach compare in terms of their performance overhead?

To analyze scalability, we study how different analysis approaches perform under different initial system sizes and how they handle different numbers of architectural changes. We perform experiments with different numbers of initial components and different adaptations. We consider an initial size of 4, 6, 8, 10, 12 and 16 components. Further, we report the interesting part of our results for brevity without loss of generality, i.e., in some cases the data is partially reported in the tables. The reason for considering different adaptations is to study how the scope of changes affects performance. In the rest of this section, we will refer to a specific experiment by the number of its initial components and adaptation, for example, 12-A2 refers to the experiment in which we perform the adaptation actions associated with **A2** on ZNN for 12 components. The experiments are performed on an Intel(R) Core(TM) i5-8265U CPU processor with 16GB of RAM running Debian Linux. We use the model checker PRISM version 4.7. In the following results, an experiment corresponds to one run of our runtime analysis triggered due to an adaptation. Each experiment was repeated five times [4]. We verify the property $\mathsf{P}_{=?}[\mathsf{F}\ \mathsf{SystemDown}]$ unless stated otherwise. SystemDown expands to the load balancer or any of the servers being affected by an attack.

Model Statistics. In our experiments, there are 4 to 16 components and 24 to 151 fragments. Figure 6 shows the number of states and transitions of the high-level model and the low-level model vs the number of the components of the initial architecture. The y axis uses a logarithmic scale which entails that the growth rate of both the number of states and number of transitions is almost exponential. As the number of components grows, so do the number of states and transitions of the models. Further, the low-level model failed to be built due to state space explosion in the four architectures marked with an X in Figure 6. The state size of the high-level model is smaller, as it abstracts away the internal capabilities of the low-level model. The large number of states, even for

[1] https://github.com/cmu-able/znn
[2] https://github.com/designsecurity/progpilot
[3] http://www.openvas.org/

[4] The full set of results alongside a virtual machine containing our implementation and experiments is available at https://doi.org/10.5281/zenodo.11524518.

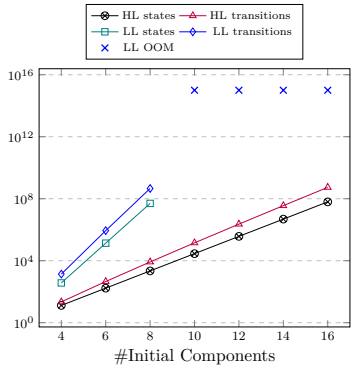


Figure 6: State space statistics

architectures with a small number of components, is due to the multiple vulnerabilities present in the components.

Table 1: Analysis time of the different approaches

#C	#A	#F	Security Model Construction (s)			Property Verification (s)			Total Time (s)		
			Seq	Mod	Inc	Seq	Mod	Inc	Seq	Mod	Inc
4	A1	24	1.36	3.83	2.37	0.01	0.003	0.003	1.37	3.83	2.37
	A2	24	0.54	3.60	0.59	0.01	0.003	0.003	0.55	3.60	0.59
	A3	31	0.54	4.28	3.13	0.01	0.003	0.003	0.55	4.28	3.13
12	A1	104	OOM	24.52	17.26	OOM	5.93	2.14	OOM	30.45	19.40
	A2	104	OOM	24.53	9.94	OOM	6.46	1.40	OOM	30.99	11.34
	A3	111	OOM	27.13	17.59	OOM	5.98	6.30	OOM	33.11	23.89
16	A1	144	OOM	40.07	33.61	OOM	1382.76	612.87	OOM	1422.83	646.48
	A2	144	OOM	39.01	18.86	OOM	1401.8	551.18	OOM	1441.81	570.04
	A3	151	OOM	38.80	33.66	OOM	1418.16	1374.43	OOM	1456.97	1408.09

Performance Analysis. Table 1 shows the analysis time for the sequential, modular and incremental analyses with respect to the number of initial components (#C), the adaptation scenario (#A) and the number of fragments after adaptation (#F). The security model construction column shows the time required to construct the whole security model, the next column gives the time for verifying the user property and the last column shows the total time. There are multiple cases where our model checker ran out of memory, which is marked with OOM in the table. *Note that the number of fragments being the same does not entail that no changes were performed or that the resulting architectures are identical.*

Table 2: Security model construction time per task

#C	#A	#F	Fragment Graphs Generation (s)			Fragment Models Construction (s)			High-level Model Construction (s)	
			Seq	Mod	Inc	Seq	Mod	Inc	Mod	Inc
4	A1	24	0.19	0.53	0.80	1.17	2.97	1.23	0.33	0.33
	A2	24	0.12	0.45	0.22	0.43	2.80	0.03	0.35	0.34
	A3	31	0.12	0.41	0.59	0.43	3.34	2.07	0.34	0.46
12	A1	104	0.15	6.15	4.22	OOM	17.52	12.01	0.82	0.74
	A2	104	0.14	3.25	3.65	OOM	20.46	5.55	0.81	0.73
	A3	111	0.15	5.15	5.61	OOM	21.12	11.00	0.84	0.83
16	A1	144	0.19	11.45	12.95	OOM	23.52	17.97	5.08	2.72
	A2	144	0.17	10.33	8.56	OOM	24.50	6.72	5.17	3.5
	A3	151	0.18	8.69	9.93	OOM	24.86	18.58	5.25	5.14

RQ1-1: Scalability. The first part of RQ1 which refers to the scalability of different analyses is addressed by Figure 6. Our results show that our approach can be used for small to medium-sized applications with up to a few tens of components. The results indicate that both the modular and the incremental analyses outperform the sequential one, noticeably reducing the model size to be verified, as the number of components grows from small-sized to medium-sized systems. For example, the number of states in the high-level model is reduced to 2×10^3 from 4.5×10^8 states in the case of 8 components. *Note that the scale in Figure 6 is logarithmic, hence, the number of states and the number of transitions almost grows exponentially. In other words, our modular and incremental analyses can handle systems with a significantly larger state space compared to the sequential approach.*

RQ1-2: Performance Overhead of different analyses. In terms of performance overhead of different approaches, i.e., the second part of RQ1, when the number of fragments is small, the sequential analysis outperforms the modular approach due to the overhead of building multiple fragment graphs in addition to building the high-level model by performing verification on the fragment models of the low-level model according to Table 1. The incremental analysis can be competitive in terms of performance with the sequential one when the number of changes to the system is small, as shown in 4-A2. As the number of fragments grows, the situation is reversed and the modular and incremental analyses outperform the sequential one. When the number of changes to the system is relatively small, the incremental analysis also outperforms the modular analysis. Conversely, when a large number of architectural changes occur between two analysis steps, in some cases, the bookkeeping and extra analysis steps required for the incremental analysis incur a slight performance overhead compared to the modular analysis.

RQ2: Performance Overhead of the Analysis Steps. As shown in Table 2, for sequential analysis, the majority of the analysis time is devoted to the low-level model construction and verification. All verification is done on the low-level model, the state space of which grows exponentially. For modular analysis, when the number of fragments is small, the construction and verification of the fragment models is the major contributing factor to the analysis time. As the number of fragments grows, the construction and verification of the high-level model affects the analysis time more. Two factors contribute to this outcome: (i) the states of the high-level model also grow exponentially, while the average number of states per fragment remain rather constant and (ii) the fragment graph generation and low-level model generation and verification are performed in parallel. Conversely, the high-level model is a single model the verification of which is not parallelized. The incremental analysis shares its performance overhead characteristics with the modular analysis, except, its low-level model construction and generation performance is improved due to the fact that it reuses past results.

Incremental vs Modular analyses. The modular analysis does not require to maintain the past runtime architectural models, in contrast to the incremental analysis that requires this model and the past analysis results. Hence, in case of small changes to the architecture, the incremental analysis will perform better than the modular one by largely reusing the old results. However, the extra bookeeping costs can incur a performance loss in case of large architectural changes.

Threats to Validity and Applicability. The system architecture, its existing vulnerabilities and the ruleset used to generate attacks and their effects impact the analysis results. Hence, alterations in the architectural style or vulnerability and ruleset density of the systems analyzed might yield different results. We believe that further experimentation with different architectures and rule inputs is required to better evaluate the effectiveness of our approach and its applicability to larger systems. The main threats to the applicability of our approach include: (i) reasoning only about known knowledge, i.e., we are unable to reason about unknown vulnerabilities or attacks, and (ii) proper setting of probabilities.

8 RELATED WORK

Modular information flow control [14] approaches ensure properties independently on each fragment of the program, protocol or software system. The enforced properties are then proven to be preserved at the whole program [47], protocol [1] or system level [58]. Information flow control approaches have also been proposed for component-based system security. In [49], the authors present a model-driven security framework to design and implement secure-by-construction component-based systems. Greiner et al. [26, 27], propose an approach for modular verification using dependency clusters, i.e., equivalence relations used to compositionally verify non-interference properties system-wide in a service-based system by composing simpler non-interfering services. The above approaches design secure-by-construction systems with a static architecture; they have not been designed with dynamic software architectures or evolving systems in mind, i.e., the compositionality of services and behaviors might not be maintained once the architecture changes.

Nejati et al. [42] provide a review of approaches to tackle state space explosion in component-based system verification. In [7], the authors use a contract-based design to specify correct interactions between modules and incrementally verify changes. In [32], the authors combine compositional verification and assume-guarantee reasoning with incremental verification to analyse component-based systems whose components and structure change dynamically at runtime. The authors in [60] present a modular and incremental approach to verify that adaptive programs adhere to requirements specified in A-LTL. The authors in [50] propose an approach based on assume-guarantee reasoning to verify adaptive embedded systems under environment constraints. The author in [33] proposes a modular symbolic method to synthesize a controller to support adaptations under partial observations and enforce safety properties. The above approaches aim for verification of functional or safety properties and do not concern themselves with security. Our work focuses on verification of probabilistic behaviors and is able to model and analyze the security of a dynamic systems at runtime.

In the context of component-based systems [11], runtime verification has been employed to check whether a component-based system adheres to its specification at runtime [19, 21]. In dynamically reconfigurable architectures and self-adaptive systems [15, 40, 55] verification has been employed to check whether reconfigurations or adaptations of the system architecture adhere to functional constraints. Filieri et al. [23] discuss and compare two parametric model checking approaches [13, 22] to improve the performance of verifying (R-)PTCL properties on DTMCs at runtime in self-adaptive systems. Parametric approaches require that the PCTL properties are known at design time so that precomputation is possible. In our approach, the security properties need not be known at design time and can be adjusted dynamically as the system evolves to accommodate new security requirements that might arise due to evolution. The above approaches aim to either check or ensure functional correctness and do not consider security. In contract to this paper, they verify the system behavior against given properties while we perform threat analysis by analyzing the attacker's behavior.

Runtime security analysis techniques often utilize runtime monitoring [20], to check if a specification is satisfied by the current execution or not [16, 31, 59], while we focus on threat analysis to compute security risks. Since these approaches check only one execution, their analysis results are incomplete. We perform full analysis at runtime, and consider all possible executions, which potentially leads to a higher performance overhead. Khakpour et al. [35] present a method to perform quantitative security risk assessment on adaptations considering dependencies of components vulnerabilities, which is informal in contrast to our approach. Skandylas et.al [53], model check security threats of a self-adaptive system during adaptation using UPPAAL [8]. The main goal of [53] was to analyze the security of all adaptation options in the system and select the most secure adaptation to reduce the exposed adaptation surface. They perform the verification on a model that corresponds to the low-level layer of our security model. However, this approach suffers from scalability issues.

9 CONCLUSION

In this paper, we proposed an automated approach for scalable security analysis of component-based systems with dynamic architectures. We exploited the power of formal abstraction and modular verification to build a two-tier probabilistic security model, which is maintained at runtime and incrementally updated. Security properties are then verified against the top abstract model for threat analysis. As future work, we plan to incorporate attack mitigation techniques into our approach by designing countermeasures that can be evaluated and applied at runtime. We also aim to investigate other techniques, e.g., assume-guarantee reasoning [37], hierarchical modeling and analysis [30] or improving our attack partitioning algorithm, to further improve the scalability of our approach to analyze larger systems.

Acknowledgment. The first author is supported by the UK Engineering and Physical Sciences Research Council (EPSRC) through the grant EP/X037274/1.

REFERENCES

[1] Suzana Andova, Cas Cremers, Kristian Gjøsteen, Sjouke Mauw, Stig Fr. Mjølsnes, and Sasa Radomirovic. A framework for compositional verification of security protocols. *Inf. Comput.*, 206(2-4):425–459, 2008.

[2] Kallol Bagchi and Godwin Udo. An analysis of the growth of computer and internet security breaches. *Communications of the Association for Information Systems*, 12:684–700, 2003.

[3] Christel Baier and Holger Hermanns. Weak bisimulation for fully probabilistic processes. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 119–130. Springer, 1997.

[4] Christel Baier, Joost-Pieter Katoen, Holger Hermanns, and Verena Wolf. Comparative branching-time semantics for markov chains. *Inf. Comput.*, 200(2):149–214, 2005.

[5] David A. Basin, Felix Klaedtke, and Samuel Müller. Monitoring security policies with metric first-order temporal logic. In James B. D. Joshi and Barbara Carminati, editors, *15th ACM Symposium on Access Control Models and Technologies, SACMAT 2010, Pittsburgh, Pennsylvania, USA, June 9-11, 2010, Proceedings*, pages 23–34. ACM, 2010.

[6] Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. Runtime verification meets android security. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, volume 7226 of *Lecture Notes in Computer Science*, pages 174–180. Springer, 2012.

[7] Yosab Bebawy, Houssem Guissouma, Sebastian Vander Maelen, Janis Kröger, Georg Hake, Ingo Stierand, Martin Fränzle, Eric Sax, and Axel Hahn. Incremental contract-based verification of software updates for safety-critical cyber-physical systems. In *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1708–1714, 2020.

[8] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets, Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned]*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[9] Saddek Bensalem, Marius Bozga, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. Component-based verification using incremental design and invariants. *Softw. Syst. Model.*, 15(2):427–451, 2016.

[10] Sergey Berezin, Sérgio Vale Aguiar Campos, and Edmund M. Clarke. Compositional reasoning in model checking. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures*, volume 1536 of *Lecture Notes in Computer Science*, pages 81–102. Springer, 1997.

[11] Alan W. Brown and Kurt C. Wallnau. Engineering of component-based systems. In *2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96), 21-25 October 1996, Montreal, Canada*, pages 414–422. IEEE Computer Society, 1996.

[12] XM Cyber. The state of exposure management in 2023, 2023. https://info.xmcyber.com/hubfs/The%20State%20of%20Exposure%20Management%202023%20-%20XM%20Cyber%20-%20Research%20Report.pdf.

[13] Conrado Daws. Symbolic and parametric model checking of discrete-time markov chains. In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004*, pages 280–294, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[14] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[15] Julien Dormoy, Olga Kouchnarenko, and Arnaud Lanoix. Runtime verification of temporal patterns for dynamic reconfigurations of components. In Farhad Arbab and Peter Csaba Ölveczky, editors, *Formal Aspects of Component Software - 8th International Symposium, FACS 2011, Oslo, Norway, September 14-16, 2011, Revised Selected Papers*, volume 7253 of *Lecture Notes in Computer Science*, pages 115–132. Springer, 2011.

[16] Denis Efremov and Ilya V. Shchepetkov. Runtime verification of linux kernel security module. *CoRR*, abs/2001.01442, 2020.

[17] Simon Yusuf Enoch, Zhibin Huang, Chun Yong Moon, Donghwan Lee, Myung Kil Ahn, and Dong Seong Kim. Harmer: Cyber-attacks automation and evaluation. *IEEE Access*, 8:129397–129414, 2020.

[18] G. Fairbanks and David Garlan. *Just Enough Software Architecture: A Risk-Driven Approach*. Boulder, CO : Marshall & Brainerd, 01 2010.

[19] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In Saddek Bensalem and Doron A. Peled, editors, *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers*, volume 5779 of *Lecture Notes in Computer Science*, pages 40–59. Springer, 2009.

[20] Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In Manfred Broy, Doron A. Peled, and Georg Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 141–175. IOS Press, 2013.

[21] Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. *Softw. Syst. Model.*, 14(1):173–199, 2015.

[22] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. Run-time efficient probabilistic model checking. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 341–350, 2011.

[23] Antonio Filieri and Giordano Tamburrelli. *Probabilistic Verification at Runtime for Self-Adaptive Systems*, pages 30–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[24] David Garlan, Robert T Monroe, and David Wile. Acme: Architectural description of component-based systems. *Foundations of component-based systems*, 68:47–68, 2000.

[25] Jaco Geldenhuys and Antti Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2004.

[26] Simon Greiner and Daniel Grahl. Non-interference with what-declassification in component-based systems. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 253–267. IEEE Computer Society, 2016.

[27] Simon Greiner, Martin Mohr, and Bernhard Beckert. Modular verification of information flow security in component-based systems. In Alessandro Cimatti and Marjan Sirjani, editors, *Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings*, volume 10469 of *Lecture Notes in Computer Science*, pages 300–315. Springer, 2017.

[28] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects Comput.*, 6(5):512–535, 1994.

[29] Andrew Hinton, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: A tool for automatic verification of probabilistic systems. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer, 2006.

[30] Jin B. Hong and Dong Seong Kim. Scalable security analysis in hierarchical attack representation model using centrality measures. In *43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop, DSN Workshops 2013, Budapest, Hungary, June 24-27, 2013*, pages 1–8. IEEE Computer Society, 2013.

[31] Hejiao Huang and Hélène Kirchner. Formal specification and verification of modular security policy based on colored petri nets. *IEEE Trans. Dependable Secur. Comput.*, 8(6):852–865, 2011.

[32] Kenneth Johnson, Radu Calinescu, and Shinji Kikuchi. An incremental verification framework for component-based software

systems. In Philippe Kruchten, Dimitra Giannakopoulou, and Massimo Tivoli, editors, *CBSE'13, Proceedings of the 16th ACM SIGSOFT Symposium on Component Based Software Engineering, part of Comparch '13, Vancouver, BC, Canada, June 17-21, 2013*, pages 33–42. ACM, 2013.

[33] Narges Khakpour. Control of self-adaptation under partial observation: A modular approach. In Antónia Lopes and Rogério de Lemos, editors, *Software Architecture - 11th European Conference, ECSA 2017, Canterbury, UK, September 11-15, 2017, Proceedings*, volume 10475 of *Lecture Notes in Computer Science*, pages 112–119. Springer, 2017.

[34] Narges Khakpour. A field-sensitive security monitor for object-oriented programs. *Comput. Secur.*, 108:102349, 2021.

[35] Narges Khakpour, Charilaos Skandylas, Goran Saman Nariman, and Danny Weyns. Towards secure architecture-based adaptations. In Marin Litoiu, Siobhán Clarke, and Kenji Tei, editors, *Proceedings of the 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 114–125. ACM, 2019.

[36] Marta Kwiatkowska, Gethin Norman, and David Parker. *Probabilistic Model Checking: Advances and Applications*, pages 73–121. Springer International Publishing, Cham, 2018.

[37] Marta Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Assume-guarantee verification for probabilistic systems. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 23–37, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[38] Kung-Kiu Lau, Keng-Yap Ng, Tauseef Rana, and Cuong M. Tran. Incremental construction of component-based systems. In Vincenzo Grassi, Raffaela Mirandola, Nenad Medvidovic, and Magnus Larsson, editors, *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering, CBSE 2012, part of Comparch '12 Federated Events on Component-Based Software Engineering and Software Architecture, Bertinoro, Italy, June 25-28, 2012*, pages 41–50. ACM, 2012.

[39] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebraic Methods Program.*, 78(5):293–303, 2009.

[40] Diego Marmsoler and Ana Petrovska. Runtime verification for dynamic architectures. *J. Log. Algebraic Methods Program.*, 118:100618, 2021.

[41] Robert Martin and Sean Barnum. Software security knowledge: Cwe. knowing what could make software vulnerable to attack. In *Conference: 23rd Systems and Software Technology Conference (SSTC)*, pages 0–62, 05 2011.

[42] Faranak Nejati, Abdul Azim Abdul Ghani, Keng-Yap Ng, and Azmi Bin Jafaar. Handling state space explosion in component-based software verification: A review. *IEEE Access*, 9:77526–77544, 2021.

[43] Xinming Ou, Wayne F. Boyer, and Miles A. McQueen. A scalable approach to attack graph generation. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, pages 336–345. ACM, 2006.

[44] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. Mulval: A logic-based network security analyzer. In Patrick D. McDaniel, editor, *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*. USENIX Association, 2005.

[45] Anand Bhushan Pandey, Ashish Tripathi, and Prem Chand Vashist. *A Survey of Cyber Security Trends, Emerging Technologies and Threats*, pages 19–33. Springer Singapore, Singapore, 2022.

[46] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems - Conference proceedings, Colle-sur-Loup (near Nice), France, 8-19 October 1984*, volume 13 of *NATO ASI Series*, pages 123–144. Springer, 1984.

[47] Adi Prabawa, Mahmudul Faisal Al Ameen, Benedict Lee, and Wei-Ngan Chin. A logical system for modular information flow verification. In Isil Dillig and Jens Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*, volume 10747 of *Lecture Notes in Computer Science*, pages 430–451. Springer, 2018.

[48] Michael Riegler, Johannes Sametinger, Michael Vierhauser, and Manuel Wimmer. A model-based mode-switching framework based on security vulnerability scores. *Journal of Systems and Software*, 200:111633, 2023.

[49] Najah Ben Said, Takoua Abdellatif, Saddek Bensalem, and Marius Bozga. Model-driven information flow security for component-based systems. In Saddek Bensalem, Yassine Lakhnech, and Axel Legay, editors, *From Programs to Systems. The Systems perspective in Computing - ETAPS Workshop, FPS 2014, in Honor of Joseph Sifakis, Grenoble, France, April 6, 2014. Proceedings*, volume 8415 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2014.

[50] Ina Schaefer and Arnd Poetzsch-Heffter. Model-based verification of adaptive embedded systems under environment constraints. *SIGBED Rev.*, 6(3):9, 2009.

[51] Stefan Schwoon and Javier Esparza. A note on on-the-fly verification algorithms. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 174–190. Springer, 2005.

[52] Charilaos Skandylas. *Design and Analysis of Self-protection : Adaptive Security for Software Systems*. PhD thesis, Linnaeus University, Department of computer science and media technology (CM), 2023.

[53] Charilaos Skandylas and Narges Khakpour. Design and implementation of self-protecting systems: A formal approach. *Future Gener. Comput. Syst.*, 115:421–437, 2021.

[54] George Spanoudakis, Christos Kloukinas, and Kelly Androutsopoulos. Towards security monitoring patterns. In Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo, editors, *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007*, pages 1518–1525. ACM, 2007.

[55] Gabriel Tamura, Norha M. Villegas, Hausi A. Müller, João Pedro Sousa, Basil Becker, Gabor Karsai, Serge Mankovski, Mauro Pezzè, Wilhelm Schäfer, Ladan Tahvildari, and Kenny Wong. Towards practical runtime verification and validation of self-adaptive software systems. In Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, volume 7475 of *Lecture Notes in Computer Science*, pages 108–132. Springer, 2010.

[56] Veracode. Unveiling the state of software security in the public sector, 2023. https://www.veracode.com/sites/default/files/pdf/resources/reports/veracode-state-of-software-security-2023-public-sector.pdf.

[57] Brandon Wang, Xiaoye Li, Leandro P. de Aguiar, Daniel S. Menasche, and Zubair Shafiq. Characterizing and modeling patching practices of industrial control systems. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1), jun 2017.

[58] Mingdi Xu, Zhaoyang Jin, Fan Zhang, and Feng Cui. Information flow-based security construction for compositional interface automata. In *Trusted Computing and Information Security*, pages 31–43, Singapore, 2020. Springer Singapore.

[59] Fadi Yilmaz and Meera Sridhar. A survey of in-lined reference monitors: Policies, applications and challenges. In *16th IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2019, Abu Dhabi, UAE, November 3-7, 2019*, pages 1–8. IEEE Computer Society, 2019.

[60] Ji Zhang, Heather Goldsby, and Betty H. C. Cheng. Modular verification of dynamically adaptive systems. In Kevin J. Sullivan, Ana Moreira, Christa Schwanninger, and Jeff Gray, editors, *Proceedings of the 8th International Conference on Aspect-Oriented Software Development, AOSD 2009, Charlottesville, Virginia, USA, March 2-6, 2009*, pages 161–172. ACM, 2009.