Sarch-Checks: A Method for Checking Software Architecture Security Properties using a Knowledge Graph

Jeisson Vergara-Vargas*†, Salah Sadou†, Chouki Tibermacine‡, Felipe Restrepo-Calle*
*Universidad Nacional de Colombia, Bogotá, Colombia
{javergarav, ferestrepoca}@unal.edu.co
†IRISA & CNRS, Université Bretagne Sud, Vannes, France
salah.sadou@irisa.fr
‡LIRMM & CNRS, Univ Montpellier, Montpellier, France
chouki.tibermacine@lirmm.fr

***Abstract*—Checking the security properties of a software system during design is essential to enable the construction of a foundationally secure system. However, combining design tasks with security checks leads to a difficult and error-prone activity. This paper presents a checking method for security properties, called Sarch-Checks. This method allows analyzing the context of architectural elements in terms of an expected security property and identifying the presence of countermeasures and vulnerabilities. It uses an architectural description of the system to be analyzed, through the use of a modeling language. It also uses a knowledge graph, modeled and built from the elements of the software architecture, and cybersecurity elements taken from official information sources such as NIST and MITRE. This solution is an aide to the architect to design more secure architectures. Additionally, a validation process of the proposed method is presented through a case study based on a real report of a vulnerability in an open-source software system.**

***Index Terms*—Security property, software architecture description, checking method, knowledge graph.**

I. Introduction

The architectural design of a software system encompasses different approaches, which enable the software requirements to be satisfied [4]. On the one hand, the architectural design supports decisions about the different structures of the system, comprised of different types of architectural elements, their relationships and their properties [13]. On the other hand, the architectural design also supports those decisions related to non-functional properties of a system, such as security, scalability, reliability, among others. These properties are generally known as software quality attributes [4]. One of the most relevant quality attributes today is security. This attribute is described as the ability of a software system to protect system elements, including data and components, from situations that may affect their confidentiality, integrity and availability [4].

Given the importance of this quality attribute, ensuring that a software system is secure is a fairly difficult task, which must be addressed at different points within the software development life cycle. This is the basis of the Secure by Design approach. Indeed, Secure by Design approach seeks to make early design decisions that allow certain security properties to be guaranteed before implementation [6]. One of these early moments is at design time, where architectural design plays a fundamental role. In addition, the architect is not always aware of architectural vulnerabilities, as these are only defined once they have been identified and reported via CVE (Common Vulnerabilities and Exposures[1]). So, the architect has to simultaneously deal with the design of the system and check for possible existing vulnerabilities. The latter involves exploring a huge database and combining these two tasks is difficult and error-prone.

Based on the above, it is essential to have methods, models and/or tools that enable to verify that the system design is faithful to the security requirements, and therefore that a fundamentally secure system can be built on the basis of that design. In this way, there are several approaches that have been proposed that allow these verification tasks to be carried out [18]. Some of them at the detailed design level or the implementation of the system, that is with decisions around the coding process [12], [21], and others at the architectural design level, that is with decisions around the system structure [1], [2], [7].

Therefore, we propose an approach that combines the classical representation of an architecture, through an architecture description or modeling language, with a knowledge graph-based representation of security weaknesses and the architectural tactics that mitigate them. The latter allows us to use languages that are powerful enough to express queries concerning vulnerabilities at the architectural level. We have implemented this approach using Neo4j, which we used to define the knowledge graph along with queries specified using the Cypher[2] language. To validate our approach, we carried out a case study on a system (Apache Airflow) whose vulnerabilities have already been reported by the National Vulnerability Database (NVD), concerning the confidentiality security property.

[1]https://cve.mitre.org/
[2]Neo4j's graph query language.

This paper is organized as follows: Section II introduces the proposed approach for verifying security properties through the use of a knowledge graph and software architecture descriptions. Section III describes the implementation aspects of the checking method. Section IV presents the validation of the approach using a case study. Section V brings the related work. Finally, Section VI presents the conclusions and some directions for future work.

II. General Overview of the Checking Method

The proposed checking method, called Sarch-Checks, has a scheme based on three fundamental elements: the inputs, the checking process and the outputs. Sarch-Checks performs an analysis process on a particular architectural element and the security property to be verified, with the objective of checking whether the property is met or not. Checking criteria include: the identification of possible weaknesses associated with the architectural element, and the presence or absence of architectural design decisions, as countermeasures, to mitigate the weaknesses.

Figure 1 shows the main steps of the proposed checking method, which are described in detail below.

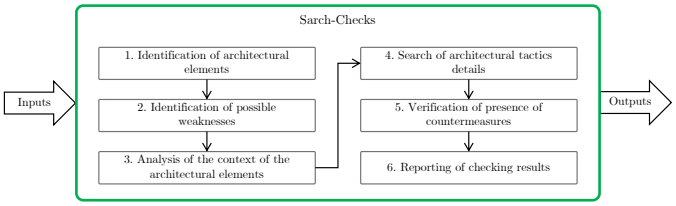

Fig. 1. General scheme of the proposed method.

A. Inputs

For Sarch-Checks, inputs refer to the elements that are required to carry out the checking process: a software architecture description and a security property to be verified.

1) *Description of software architecture:* following the ideals of Secure by Design, we seek to make decisions at design time that ensure security before implementation. The architecture description should include: architectural elements, architectural relationships and properties of both. In the case of Sarch-Checks, the component and connector structure is used, which contains those architectural elements that are present in the system at runtime.
2) *Security property to check:* Security is a fairly broad quality attribute, so it covers different properties [3]. In the case of Sarch-Checks, the three fundamental characteristics of security, known as the triad in the field of Cybersecurity, are taken as a reference: Confidentiality, Integrity and Availability (CIA). Confidentiality refers to the system's ability to protect its elements, primarily its data, from access or disclosure by unauthorized actors. Integrity refers to the system's ability to guarantee that its elements, especially its data, have not been altered in an unauthorized manner: guarantee of completeness and correctness. Finally, availability refers to the system's ability to guarantee that its elements will be available at the time they are required.

B. Checking Process

Sarch-Checks proposes a checking process based on the described inputs where it carries out a set of internal steps. Each step requires some inputs and produces some outputs. Below are the different steps:

1) *Identification of architectural elements*: it consists of the automatic analysis of the architectural description received as input, in order to identify each of the architectural elements to be analyzed. *Input:* description of the architecture, *output:* architectural elements.
2) *Identification of possible weaknesses*: for each architectural element identified in the previous stage, and with the security property to be verified, a search is performed in the knowledge graph, in order to find both the possible security weaknesses that the architectural element may have in relation to the specified security property, as well as possible architectural tactics that can be applied to mitigate these weaknesses. The knowledge graph is a complementary element to Sarch-Checks, which contains information related to software architecture and cybersecurity, modeled and related from different official sources in these areas. *Input:* architectural elements, security property, *output:* weaknesses and related architectural tactics.
3) *Analysis of the context of the architectural elements:* after identifying the possible weaknesses and the respective architectural tactics that can be used as countermeasures, an analysis of the context of the architectural elements in the global architecture of the system is carried out. This analysis consists of identifying the conditions in which this element is found in the architecture: interactions with other elements, interaction characteristics and internal properties. *Input:* weaknesses and related architectural tactics, *output:* context of the architectural elements.
4) *Search of architectural tactics details:* a new search is performed in the knowledge graph to retrieve the implementation details of the related architectural tactics. *Input:* related architectural tactics, *output:* architectural tactics details.
5) *Verification of presence of countermeasures:* an inspection process is executed on the context of the architectural element to identify whether or not there is evidence of the presence of the related tactics. *Input:* architectural tactics details, *output:* presence or not presence of architectural design decisions that correspond to the tactics.
6) *Reporting of checking results:* finally, after the inspection process, according to the result:
   - If there are design decisions (application of countermeasures): the security property is *guaranteed*.

- If there are no design decisions (absence or poor application of countermeasures): the security property is *not guaranteed*. In this case, the vulnerabilities are reported as instances of the related weaknesses.

III. Implementation

For the implementation of the proposed checking method, called Sarch-Checks, an information analysis process was carried out, as well as the use and construction of software methods and tools that allow interacting with each of the elements needed by our checking method. Figure 2 gives a general overview on the implementation of the checking method. Below we describe the elements of the method in detail.

A. Architecture Modeling Language

For the representation of the software system's architecture, an architecture modeling language called *Sarch* [5], [19] is used, which allows the design of software architectures emphasizing the different structures that make up a software system. Sarch has the ability to model the structure of components and connectors in a generic way, using only the elements and relationships formally defined in the domain.

1) *Components:* are those architectural elements that are present at runtime and constitute the fundamental units or building blocks of the architecture of a system. They are mainly associated with computational elements, such as components built with general-purpose programming languages that process data, and data stores, such as databases.
2) *Connectors:* are those architectural elements that serve as a communication bridge between two components. Generally, connectors have the function of transmitting the data that transits from one component to another, guaranteeing interaction flows within the system architecture.
3) *Ports:* they are secondary architectural elements, associated with the components. Ports refer to the interfaces that the components have, in order to interact with their surroundings, that is, with other components or systems.
4) *Roles:* they are secondary architectural elements associated with the connectors. The roles can be considered as the interfaces of the connectors, and their function is to determine the way in which the connector can be used by the components for their interaction.
5) *Attachments:* relations that allows communication between a components and connectors. The relationship can be denoted as the association between the port of the component and the role of the connector.

B. Knowledge Graph

Our method is based on a knowledge graph, called *Sarch-Knows*, which is modeled from the elements of two domains: software architecture and cybersecurity. This knowledge graph is implemented as a database in Neo4j and consists of nodes classified into two main categories: abstract elements and specific elements. Figure 3 shows a general representation of the knowledge graph structure.

The abstract elements refer to fundamental concepts of each of the two domains involved that are required for the proposed checking method:

- The *Software Architecture* field encompasses the following elements: system structure, architectural element, architectural relationship, component and connector structure, components, connectors, ports, roles, and design decisions. Design decisions are classified as: architectural tactics [4], which refer to those decisions that influence the achievement of a quality attribute response to some stimulus, and architectural patterns [8], which refer to those decisions that describe recurring problems in a particular design context, presenting an appropriate architectural solution to solve the problem.
- The *Cybersecurity* field encompasses the following elements: risk, threat, weakness, attack and countermeasure.

On the other hand, specific elements can be considered as instances of abstract elements, in the following way:

- The specific elements of the part of the graph associated with software architecture are categorized as SASE (Software Architecture Specific Elements). They are created from the literature in the field, emphasizing the architectural elements that are part of the structure of components and connectors and that are frequently used in the architectural design of different types of software systems.
- The specific elements of the part of the graph associated with cybersecurity are categorized as CCSK (Current Common Security Knowledge) which are created from different official sources of information in the field, mainly the NVD (National Vulnerability Database) and the following MITRE classifications: CWE (Common Weakness Enumeration) and CAPEC (Common Attack Pattern Enumeration and Classification).

The knowledge graph is defined as follows:

- Both abstract and specific elements are modeled as nodes.
- The nodes of each domain (software architecture and cybersecurity) are connected by logical links based on their conceptual relationships.
- There are also logical links that connect nodes of the two domains. These links are based on the relationship that exists between an architectural element, the associated security elements (weakness, threat, attack, risk) and possible countermeasures in terms of architectural tactics, implemented as patterns. This information was extracted from the description of CWEs associated with weaknesses in software architectures [15].

The graph receives queries using the Cypher language and, according to the parameters received, uses the attributes of the nodes to return a subgraph associated with a security scenario involving elements of the software architecture. Likewise, it is important to mention that the software architecture domain

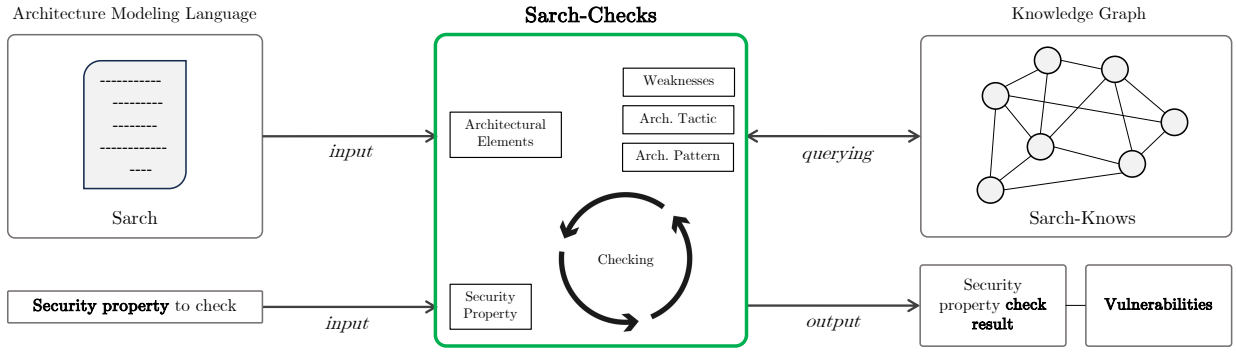

Fig. 2. Sarch-Checks: Overview of the checking method.

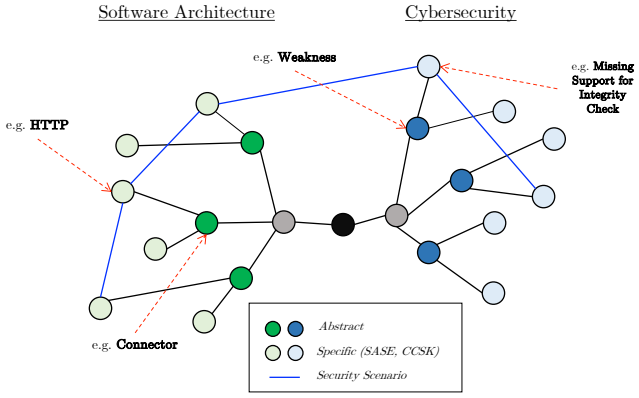

Fig. 3. Knowledge graph overview.

modeling in Sarch-Knows is consistent with the grammar of Sarch language.

C. Checker

Based on the above, the proposed checking method is implemented using both Sarch and the Sarch-Knows knowledge graph. Roughly speaking, Sarch-Checks will look at each node, representing a concrete architectural element, and search for any links to nodes representing weaknesses. The result is a list of architectural elements, each linked to a list of weaknesses. Knowing that in our knowledge graph we have links between weaknesses and architectural patterns, representing the countermeasure, the check for the existence of a vulnerability is as follows: for each weakness found, check whether the architectural element to which it is linked is part of a structure that conforms to the countermeasure pattern recommended for the weakness in question. If the pattern is not implemented in the architecture, we suspect that there is a vulnerability associated with that weakness.

The implementation details based on the steps of the method are described below:

- *Identification of the architectural elements:* The architectural description is read in Sarch, obtaining a list of architectural elements.
- *Identification of possible weaknesses:* For each architectural element identified, the following Cypher query is made, to the Sarch-Knows knowledge graph to find possible related weaknesses:

```
MATCH subgraph =
(a:specific {KEY: 'VALUE'})-
[:RELATIONSHIP*]-(b)
RETURN subgraph;
```

KEY: attribute to search (e.g., name), VALUE: architectural element (e.g., Database, REST), RELATIONSHIP: possible relationships (e.g., CanAffect, IsRelatedTo, CanBeMitigatedBy).*

As a result, a subgraph with the following data is obtained: architectural element, possible weaknesses, possible countermeasures, related attacks, associated security properties, related architectural tactics and related architectural patterns.

Figure 4 shows an example of the subgraph obtained from the query. This example concerns the REST architectural element. This element is associated with two weaknesses (Improper Authentication, Missing Authorisation). Each of the obtained weaknesses is related to some security properties (sp2 and sp3 in the figure) and, at the same time, with some countermeasures (c1 and c2). From the countermeasures we can deduce the needed architectural tactic (e.g. at2) and its example of implementation as a pattern (e.g. ap2).
- *Verification of presence of countermeasures:* Based on the weakness found, we have the associated tactic as a countermeasure and its various implementations as architectural patterns. Each pattern has a Cypher query that can be applied to the architecture to identify its presence. Thus, if the pattern is identified in the analysed architecture, and the architectural element linked to the weakness is part of the pattern, then we can consider that the architecture in question holds an architectural decision that implements the tactic countermeasuring the weakness found. Otherwise, if no pattern is found that implements the countermeasure tactic, then we can suspect the existence of a vulnerability linked to this weakness in the architecture.

Below is a general presentation of the structure of a Cypher generic query to obtain check the presence of

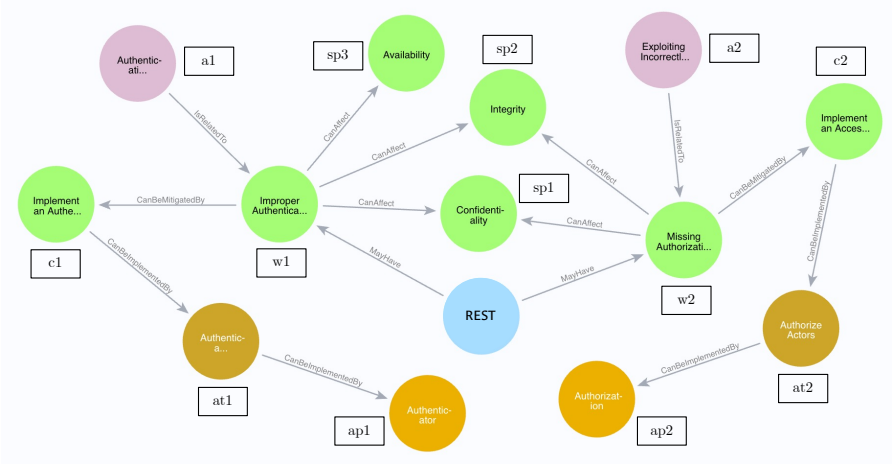

Fig. 4. Response subgraph associated with the REST architectural element.

an architectural pattern in the described software architecture.

```
//Part 1
WITH ['element_1', ..., 'element_n']
AS architecture_subgraph
CREATE (arch_element:ArchitecturalElement{})
-[:RELATIONSHIP*]->
(arch_element:ArchitecturalElement})
//Part 2
MATCH (n:entity_pattern)
-[:RELATIONSHIP*]->(m:entity_pattern)
WITH COLLECT(n) AS pattern_subgraph
//Part 3
FOREACH (entity_pattern
    IN COLLECT(pattern_subgraph)
    FOREACH (arch_element
    IN COLLECT(architecture_subgraph) |
          WHERE arch_element
                <> entity_pattern
          MATCH (n:arch_element)
          DELETE n
   )
)
RETURN CASE WHEN SIZE(nodosColeccion) = 0
THEN ' The graph is empty,
the pattern is applied.'
ELSE ' The graph is not empty, the pattern
is not applied.'
END AS result
```

Part 1 creates a new subgraph with the architecture elements. Part 2 obtains the subgraph containing the implementation details of the architectural pattern. And finally, part 3, compares each element (node) of the architecture (architecture_subgraph) with each element of the pattern (pattern_subgraph). If the elements are equivalent, the element of the architecture subgraph is removed, if not, the element is kept. In the end, if the architecture subgraph is empty, it means that the pattern is present in the architecture. Overall, this query extracts the structure of the pattern in question from the knowledge graph to try to find it in the architecture part of the graph.

- *Reporting of checking results:* The analysis report presents all the weaknesses in the architecture, with the architectural element concerned, specifying the security property not satisfied for each of them. In addition, it suggests the recommended tactic and examples of patterns implementing the tactic to achieve the countermeasure. All these elements are present in the knowledge graph.

IV. Case Study

In order to validate the proposed checking method, we conducted a case study using an open-source software system called Apache Airflow [3]. To carry out the execution of the case study, the steps presented by Wohlin et al. in [20] are taken as reference and described below. Applied to our case study this corresponds to the following steps: Design and planning, preparation and collection of data, data analysis and discussion on the results.

A. Design and Planning

The objective of this case study is to validate if the proposed method is effective in identifying the guarantee of a security property and the possible vulnerabilities at the software system architecture level.

To achieve this objective, we used an open-source software system, called Apache AirFlow, which has a report of vulnerabilities in CVE and NVD. This system is a platform created by the community to programmatically author, schedule and monitor workflows. Thus, after a reverse engineering process carried out on the software system from its public repository and its official documentation, we seek to abstract its architecture in order to apply the proposed checking method and compare the results. Thus, we applied the checking method based and our implementation and answer the following research question:

RQ: Is it possible to have evidence of a guarantee of a security property from the architectural description of a software system?

To concretely carry out this case study, the following steps were followed: 1) identification and selection of a reported

[3]https://airflow.apache.org/

vulnerability, 2) reverse engineering the software system to abstract the architecture, 3) execution of the checking process, and 4) analysis of the results.

B. Preparation and Collection of Data

The preparation and collection of data mainly covers the first three steps mentioned above.

1) Identification and Selection of a Reported Vulnerability: Apache Airflow is a web-based software system. Web-based software systems are among those with the largest attack surface and therefore the greatest security risks. For this reason, a reported real vulnerability of Apache Airflow is used: CVE-2020-13927 [4], which was reported in the Common Vulnerabilities and Exposures (CVE) program, whose function is to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities. This CVE report is supported by the CVE-2020-13927 Detail [5] entry in the National Vulnerability Database (NVD). This vulnerability is mapped from a weakness classified in the Common Weakness Enumeration (CWE) as CWE-287: Improper Authentication [6].

It should be noted that the description in CVE indicates several security properties. However, in the context of this case study we only consider the triad of security properties best known and accepted by the community, namely Confidentiality, Integrity and Availability (CIA).

The objective of this selection is to have a reference of a software system with a real reported vulnerability, which can be compared with the automatic checking by Sarch-Checks and in this way to be able to validate the checking method through the case study.

2) Reverse Engineering the Software System to Abstract the Architecture: Although the vulnerability has already been mitigated, the official documentation of Apache Airflow project makes it possible to identify and analyze the characteristics of the system at the time the vulnerability was found.

Based on the above, a reverse engineering process is performed, starting from the publicly available source code [7] and the official documentation [8] published on the Apache Airflow website. The aim of this reverse engineering is to abstract the architecture at the level of components and connectors.

3) Description of the Software System Architecture: At this point, having completed the process of identifying and selecting the vulnerability and abstracting the system architecture, we proceed to describe this architecture in Sarch.

Figure 5 represents the architecture obtained after the analysis, as well as its representation in Sarch language.

4) Execution of the Checking Process: The checking process is executed on the architecture without specifying a special security property. Therefore, it is expected to obtain as a result guarantees of the properties of confidentiality, integrity

[4]https://www.cve.org/CVERecord?id=CVE-2020-13927
[5]https://nvd.nist.gov/vuln/detail/CVE-2020-13927
[6]https://cwe.mitre.org/data/definitions/287.html
[7]https://github.com/apache/airflow/
[8]https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/overview.html

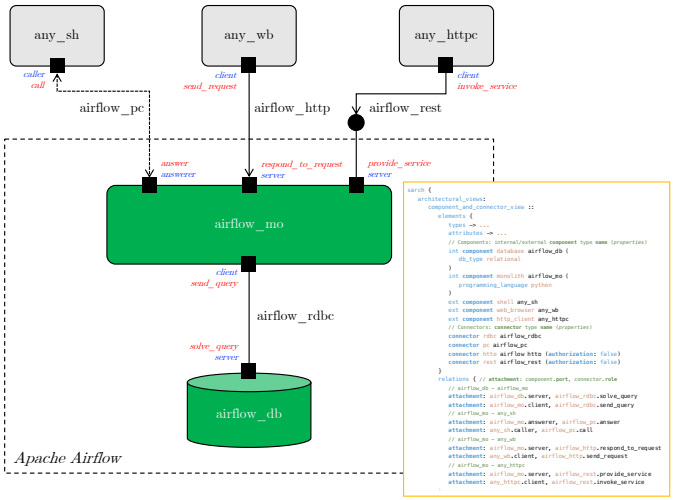

Fig. 5. Apache Airflow architecture (component ports, connector roles).

and availability, for all the architectural elements that are part of the system.

Consequently, as the main mechanism for data collection in this case study, different observations were made during the execution of the checking process.

According to the elements present in the architectural description that is taken as input, nine elements can be observed, of which five are components (airflow_db, airflow_mo, any_sh, any_wb and any_httpc,) and four are connectors (airflow_rdbc, airflow_sh, airflow_http and airflow_rest). In this way, after the identifying the architectural elements, Sarch-Checks is applied in order to carry out the respective checking process. According to the proposed method, its execution allows us to observe: 1) the possible weaknesses associated with the Apache AirFlow architecture, 2) the countermeasures associated with the identified weaknesses, 3) the related architectural tactics and patterns as countermeasures, 4) the presence or absence of related tactics and patterns in the Apache AirFlow architecture, and 5) in cases of non-presence of tactics and patterns in the architecture, the resulting vulnerabilities.

C. Data Analysis

After executing the verification process based on the Apache Airflow architecture, it is possible to observe the obtained findings summarized in Table I. Firstly, a set of possible weaknesses is identified, according to the architectural elements of Apache Airflows and the respective security properties that can be verified in Sarch-Checks. Likewise, a set of architectural tactics and patterns are also identified, associated as countermeasures to these weaknesses. And finally, a result that indicates whether the security properties are guaranteed or not. It is important to highlight again that the guarantee on security properties corresponds to the presence of the corresponding architectural patterns in the system architecture.

Elements from Table I having a value No (bold) correspond to suspected vulnerabilities. That is, they are associated with a weakness without being covered by the implementation

TABLE I
RESULTS OF THE CHECKING PROCESS OF SARCH-CHECKS

Architectural Element	Architectural Element Type	Weakness	Security Properties	Architectural Tactic	Architectural Pattern	Guarantee
airflow_http	HTTP Connector	CWE-287: Improper Authentication	Confidentiality Integrity Availability	Authenticate Actors	Authenticator	Yes
		CWE-862: Missing Authorization	Confidentiality Integrity	Authorize Actors	Authorization	Yes
		CWE-353: Missing Support for Integrity Check	Integrity	Verify Message Integrity	Transport Layer Securiy	Yes
		CWE-354: Improper Validation of Integrity Check Value	Integrity	Verify Message Integrity	Transport Layer Securiy	Yes
airflow_rest	**REST Connector**	**CWE-287: Improper Authentication**	**Confidentiality Integrity Availability**	**Authenticate Actors**	**Authenticator**	**No**
		CWE-862: Missing Authorization	Confidentiality Integrity	Authorize Actors	Authorization	**No**
		CWE-353: Missing Support for Integrity Check	Integrity	Verify Message Integrity	Transport Layer Securiy	Yes
		CWE-354: Improper Validation of Integrity Check Value	Integrity	Verify Message Integrity	Transport Layer Securiy	Yes
airflow_mo	Monolithic Component	CWE-250: Execution with Unnecessary Privileges	Confidentiality Integrity Availability	Limit Access	Secure Three-Tier Architecture	Yes
airflow_db	Database Component	CWE-250: Execution with Unnecessary Privileges	Confidentiality Integrity Availability	Limit Access	Secure Three-Tier Architecture	**No**

of a pattern recommended for the countermeasure in the architecture. We can be observed that one of them, involving the REST Connector element (line in bold), with weaknesses "CWE-287 Improper Authentication", corresponds to the vulnerability reported for Apache Airflow in CVE and NVD. The description of the vulnerability officially reported in CVE: *"The previous default setting for Airflow's Experimental API was to allow all API requests without authentication"*.

Precisely, the weakness reported by Sarch-Checks corresponds to a weakness in the REST connector (airflow_rest) that communicates the monolithic component (airflow_mo) of Apache Airflow with an external HTTP client (airflow_httpc). Specifically, the lack of authorization mechanisms when consuming the REST-API associated with the exposure of services by the monolith. The above allows us to validate that the proposed checking method manages to identify the same vulnerability officially reported for Apache Airflow.

However, the method also manages to identify additional vulnerabilities that are not officially reported in CVE:

- Element REST Connector with weakness CWE_282: This impacts confidentiality and integrity properties. However, as this element had already been declared in a CVE for confidentiality with the weakness CWE_287, we can think that the countermeasure will take into account the integrity property. Indeed, the property of confidentiality often implies that of integrity.
- Element Database Component with weakness CWE_250: The weakness means that it is possible to make some execution without need of privileges. In this case, the recommandation is to apply the 'Limit Access' architectural tactic. The value 'No' in the 'Guarantee' column means that Sarch-Checks has not found a pattern implementing this tactic in the architecture.

In the second case, to apply a countermeasure to this weakness it is suggested to implement the 'Secure Three-Tier Architecture' pattern [16]. We checked manually in the architecture of Apache Airflow and we did not find such a pattern. Thus, we suspect the presence of a vulnerability associated with this weakness in the architecture.

D. Discussion

From the observation made during the execution of the case study and the results obtained, the proposed research question RQ can be answered. It is possible to have evidence of a guarantee of a security property from the architectural description of a software system. The guarantee is based on the use or not of a pattern implementing an architectural tactic dedicated to the countermeasure. However, we speak of suspension because an architectural decision can cover a countermeasure without conforming to a given pattern. Thus, our approach can be considered as an aid to the architect to draw attention to parts of the architecture that may contain vulnerabilities.

The use of an architectural description that appropriately abstracts architectural elements, relationships, and properties can be used to perform a security properties checking process. This is possible if there is a knowledge base that abstracts different security scenarios, involving possible architectural elements and their possible weaknesses, as well as the countermeasures that can be taken to mitigate the risk, particularly using architectural tactics and patterns. The Sarch-Knows knowledge graph, used by Sarch-Checks, is a synthesis of all this. However, it is based on current knowledge of weaknesses and countermeasures. This may change over time. This is why we clearly distinguished in Sarch-Knows the stable part of security knowledge, implemented by abstract nodes, and the evolving part implemented by instance nodes of the abstract nodes. This structuring of Sarch-Knows will facilitate its evolution.

V. RELATED WORK

Several works have been proposed in the identification of vulnerabilities at the software design level, some of them focused on the identification and classification of vulnerabilities at the architectural design level [9], [11], [14]. In addition, Santos et al. in [16] propose a catalog of architectural security weaknesses, classified based on common architectural tactics. This classification generates its own view in CWE, called Architectural Concepts [9]. This view provides a classification and organization of weaknesses according to security architectural tactics and is a fundamental piece for the construction of Sarch-Knows, the knowledge graph that supports the proposed checking method.

In terms of verification, several methods have been proposed that enable the verification of security properties at the software design level. Among them, the use of security tests can be highlighted to guarantee compliance with the established functional security requirements [12]. Likewise, works focused on the verification of security properties through the application of different design patterns are also presented in [10], [21].

[9]https://cwe.mitre.org/data/definitions/1008.html

Likewise, among the proposed works at the architectural design level is an approach to support security analysis by using security scenarios and metrics [2]. This approach is based on formalizing attack scenarios and security metrics signature using the Object Constraint Language (OCL). Although OCL can be used to define certain architectural decisions linked to quality properties [17], it is not a powerful enough language to express security flaws. For instance, it is not easy to express a constraint on the data flow of an architecture. However, tracking the data flow is one of the ways to discover flaws in an architecture.

Finally, among other notable works is an architecture evaluation method using behavioral models with structural analysis for detecting of inconsistencies in security that were not perceived at the design phase [1] and an analysis approach to identify architectural design flaws using Design Rule Spaces [7]. However, these works contrast with our approach, which emphasizes the analysis and application of early security decisions based on a description of the architecture using a modeling language and a knowledge graph.

VI. Conclusions and Future Work

This paper presents a new method to check security properties at the architectural level of a software system. The proposed method makes use of a knowledge graph, built from fundamental elements and instances of software architecture and cybersecurity. This method enables the automatic checking of a security property based on an architectural description and the information found in the knowledge graph.

The main contribution of this work is to avoid theoretical modeling of weaknesses to search for them in the architecture. This way of doing things creates two biases: one is linked to the construction of the model of the weakness and the other is linked to the construction of the similarity metric to be used to find the model in the architecture. The combination of these two biases is often the source of false positives. To do this, our approach is based on validated knowledge, on security at the architectural level, accessible to everyone for verification.

The weakness of our approach is that it only covers already known weaknesses. It is therefore dependent on the community's contribution in the area of security.

Our future work aims to go beyond the limitation cited above. The objective is to characterize the architectural configurations that can lead to vulnerabilities. To do this, we need to collect a very large number of vulnerable architectural structures to be able to launch machine/deep learning allowing us to define clusters: architectural patterns producing vulnerabilities.

References

[1] Sarah Al-Azzani and Rami Bahsoon. Secarch: Architecture-level evaluation and testing for security. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pages 51–60, 2012.

[2] Mohamed Almorsy, John Grundy, and Amani S. Ibrahim. Automated software architecture security risk analysis using formalized signatures. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 662–671. IEEE Computer Society, 2013.

[3] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems, Third Edition*. John Wiley & Sons Inc, 2020.

[4] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 4th edition edition, 2021. OCLC: 1251808773.

[5] Eduardo Berrio-Charry, Jeisson Vergara-Vargas, and Henry Umaña-Acosta. A component-based evolution model for service-based software architectures. In *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*, pages 111–115, 2020.

[6] Daniel Sawano Dan Bergh Johnsson, Daniel Deogun. *Secure by Design*. Manning Publications, 2019.

[7] Qiong Feng, Rick Kazman, Yuanfang Cai, Ran Mo, and Lu Xiao. Towards an architecture-centric approach to security analysis. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 221–230, 2016.

[8] Eduardo Fernandez-Buglioni. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. John Wiley & Sons, June 2013. Google-Books-ID: 3vppszXPdr0C.

[9] Danielle Gonzalez, Fawaz Alhenaki, and Mehdi Mirakhorli. Architectural security weaknesses in industrial control systems (ics) an empirical study based on disclosed software vulnerabilities. In *2019 IEEE International Conference on Software Architecture (ICSA)*, pages 31–40, 2019.

[10] Eunsuk Kang. Robustness analysis for secure software design. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Security from Design to Deployment*, SEAD 2020, page 19–25, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Yves R. Kirschner, Maximilian Walter, Florian Bossert, Robert Heinrich, and Anne Koziolek. Automatic derivation of vulnerability models for software architectures. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, pages 276–283, 2023.

[12] Wissam Mallouli. Security testing as part of software quality assurance: Principles and challenges. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 29–29, 2022.

[13] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, oct 1992.

[14] Joanna C. S. Santos, Anthony Peruma, Mehdi Mirakhorli, Matthias Galstery, Jairo Veloz Vidal, and Adriana Sejfia. Understanding software vulnerabilities related to architectural security tactics: An empirical investigation of chromium, php and thunderbird. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 69–78, 2017.

[15] Joanna C. S. Santos, Katy Tarrit, and Mehdi Mirakhorli. A catalog of security architecture weaknesses. In *2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017*, pages 220–223. IEEE Computer Society, 2017.

[16] Joanna C. S. Santos, Katy Tarrit, and Mehdi Mirakhorli. A catalog of security architecture weaknesses. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 220–223, 2017.

[17] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. A family of languages for architecture constraint specification. *J. Syst. Softw.*, 83(5):815–831, 2010.

[18] Katja Tuma, Laurens Sion, Riccardo Scandariato, and Koen Yskout. Automating the early detection of security design flaws. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '20, page 332–342, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Jeisson Vergara-Vargas and Henry Umaña-Acosta. A model-driven deployment approach for scaling distributed software architectures on a cloud computing platform. In *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 99–103, 2017.

[20] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[21] Xiaoyu Zheng, Dongmei Liu, Hong Zhu, and Ian Bayley. Pattern-based approach to modelling and verifying system security. In *2020 IEEE International Conference on Service Oriented Systems Engineering (SOSE)*, pages 92–102, 2020.