Towards an Automated Approach for Detecting Architectural Weaknesses in Critical Systems

Joanna C. S. Santos, Selma Suloglu, Joanna Ye, and Mehdi Mirakhorli
Rochester Institute of Technology
Rochester, NY, USA
{jds5109,sxsvse,jxy1164,mxmvse}@rit.edu

ABSTRACT

Architecture-first approaches are increasingly widely adopted for addressing resiliency requirements in critical systems. In these approaches, the system is built from the ground-up to be resilient, starting with the system's architecture design. Therefore, it is crucial to ensure that the architecture design is robust, without any flaws that could compromise the system's ability to detect, prevent, react to or recover from adverse conditions, such as cyber-attacks. In this paper, we describe our ongoing efforts in aiding software architects in designing cyber-resilient systems by automatically detecting weaknesses in their architectural models.

CCS CONCEPTS

• **Security and privacy → Logic and verification**; • **Software and its engineering → Software architectures**; **Software verification**; **Formal software verification**; **System modeling languages**.

KEYWORDS

Architectural flaws, Architecture Analysis and Design Language, Automated Architectural Weaknesses Detection

ACM Reference Format:
Joanna C. S. Santos, Selma Suloglu, Joanna Ye, and Mehdi Mirakhorli. 2020. Towards an Automated Approach for Detecting Architectural Weaknesses in Critical Systems. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3387940.3392222

1 INTRODUCTION

Architecture-first development approaches are increasingly becoming the mainstream for addressing cyber-resiliency concerns in mission-critical and software-intensive systems [5]. In these approaches, resiliency is built into the system from the ground up starting with a robust software architecture design [6, 8]. As a result, *weaknesses* in the architecture of a software system can have a greater impact on the system's ability to anticipate, withstand, recover from, and adapt to adverse conditions, stresses, attacks, or compromises on cyber resources.

Despite the importance of the architecture-first approach for enhancing and ensuring the cyber-resiliency of critical systems, existing research in the field has focused on aiding the verification of a single quality attribute (e.g. safety, security) in an architecture [2, 4, 12, 20, 21]. However, *cyber-resiliency* is a much broader concern that involves multiple quality attributes including not only security but also evolvability, performance, etc [5]. Therefore, aiding the construction of resilient critical systems requires comprehensive reasoning over the system's design. In this paper, we articulate potential research directions in addressing this challenge by creating an approach that can automatically detect weaknesses in the design of a cyber-resilient system.

2 A CATALOG OF ARCHITECTURAL WEAKNESSES

While designing a critical system to fulfill resiliency requirements, architects can adopt ***architectural tactics***. They are reusable design mechanisms to detect, resist (withstand), react to, recover from, and prevent adverse conditions to compromise the system's behavior and resources [3, 5]. In our earlier work, we developed the CAWE catalog (Common Architectural Weaknesses Enumeration) [18]. An entry in the current version of our CAWE catalog describes an **architectural weakness** associated with an architectural **security tactic** adopted in a software system that results in a vulnerability. Table 1 shows an example of a weakness in the CAWE catalog (*Insufficient Compartmentalization*), which is due to a mistake in the adoption of the *Authorize Actors* tactic [3]. As shown Table 1, each entry is composed of several sections, such as the *impacted tactic*, the *impact type* (omission, commission, realization [17, 19]), a *description*, *source code examples*, *mitigation techniques*, *common consequences*, and *attack patterns.* The catalog currently has **223** weaknesses categorized based on their impact over **11** security tactics.

2.1 Architectural Tactics for Cyber-resiliency

Our earlier effort in developing the CAWE catalog [18] organized architectural weaknesses per architectural *security* tactics. However, other architectural tactics can also be important for cyber-resilient systems, such as *availability tactics* and *reliability tactics* [3, 5]. Thus, we conduct an extensive literature review of research papers and technical reports to identify the set of architectural tactics that are relevant for cyber-resilient systems (we refer to such tactics as *"cyber-resiliency tactics"*). We aim to obtain an extensive **list of cyber-resiliency tactics** and how they are applied to software systems (i.e., their **roles** and **properties**). Moreover, we identify the architectural weaknesses associated with them.

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392222

Impacted Tactic	Name: Authorize Actors Description: Enforces that agents have the required permissions before performing certain operations, such as modify data or access a resource.	**Impact Type**	Commission Flaw
Weakness	Title: Insufficient Compartmentalization (CWE-653) Description: The product does not sufficiently compartmentalize functionality or processes that require different privilege levels, rights, or permissions. When a weakness occurs in functionality that is accessible by lower-privileged users, then without strong boundaries, an attack might extend the scope of the damage to higher-privileged users. Common Consequences: • *Access Control*: The exploitation of a weakness in low-privileged areas of the software can be leveraged to reach higher-privileged areas without having to overcome any additional obstacles ○ *Technical Impact*: Gain Privileges or Assume Identity; Bypass Protection Mechanism. Demonstrative Example: **Example 1:** Single sign-on technology is intended to make it easier for users to access multiple resources or domains without having to authenticate each time. While this is highly convenient for the user and attempts to address problems with psychological acceptability, it also means that a compromise of a user's credentials can provide immediate access to all other resources or domains. **Example 2:** The traditional UNIX privilege model provides root with arbitrary access to all resources, but root is frequently the only user that has privileges. As a result, administrative tasks require root privileges, even if those tasks are limited to a small area, such as updating user man pages. Some UNIX flavors have a "bin" user that is the owner of system executables, but since root relies on executables owned by bin, a compromise of the bin account can be leveraged for root privileges by modifying a bin-owned executable. Potential Mitigations: • *Architecture and Design Phase:* Break up privileges between different modules, objects or entities. Minimize the interfaces between modules and require strong access control between them. **(...)**		

Table 1: An Example of an Entry in the CAWE Catalog

2.2 Formally Specifying Weaknesses

Although the entries in the current version of the CAWE catalog give a high-level understanding of the nature of architectural weaknesses, they are not documented at a level of formalism that can enable the automated reasoning of an architecture. Hence, we extend the CAWE catalog to also provide a **formal specification** for each entry. We call such formal specifications "**CAWE-Models**". They include the following elements:

– A ***conceptual graph***, that indicates how the weakness occurs in a system. It has the following elements:

- A set of interrelated *architectural roles* that are the nodes in the conceptual graph. Architectural roles correspond to the *components* involved in the architectural tactic.
- *Expected properties* that indicate how a system will (mis)behave in the occurrence of an adversity. They correspond to the data and control flow between the conceptual roles in a CAWE-Model. These data and control flow are **directed edges** in the conceptual graph.

– ***Generic formal rules*** that can be used to detect the weakness in an architectural model and associated *mitigation techniques*.

CAWE-Model Examples. To illustrate these formal specifications, consider the tactics *Validate Inputs*, *Authenticate Actors* and *Encrypt Data* [3]. Each of these tactics have a total of 39, 29, 38 associated weaknesses, respectively. Among these weakness there are: *Improper Input Validation* (CAWE-20), *Missing Authentication for Critical Function* (CAWE-306) and *Transmission of Sensitive Data* (CAWE-319) [17–19]. These weaknesses are modeled in conceptual graphs as shown in Figure 1. In these graphs, the *architectural roles* are depicted with a white background and bold borders. The "Intermediary" nodes indicate that there can be other elements in between these tactical components. The conditional flow (i.e., it only flows if a condition is true) is modeled as dashed arrows whereas data flows are modeled as full arrows. For instance, the `Critical Component` is only accessed if the identity check performed by the `Authenticator` had succeeded.

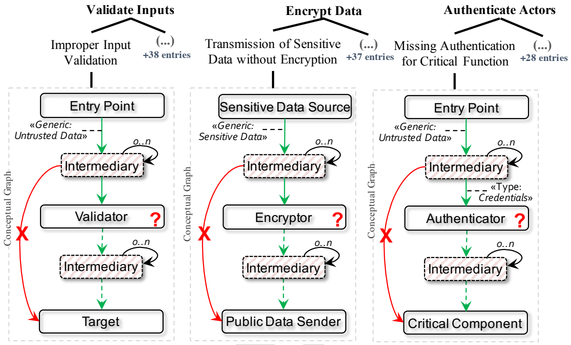

Figure 1: Conceptual Graphs for Three Weaknesses

The first conceptual graph (from left to right) models a case where the critical system does not correctly validate external inputs (either because an untrustworthy input *bypassed* the validation — connection highlighted in red with an "X" on it — or the system lacked a `Validator` component). The second graph models a scenario in which the system leaks sensitive data due to lack of encryption before transmission (either caused by a bypass or missing an `Encryptor` element). The third graph concerns the lack (or bypass) of the `Authenticator`.

For each conceptual graph, we have a set of formal rules written in the Resolute language [9]. They capture how the system needs to be designed to prevent the weakness. Listing 1 contains the top-level rule for the CAWE-20 which claims that (i) the system has a `Validator` component and (ii) all inputs to `Target` elements are properly validated before use. The claims (i) and (ii) are further verified in sub-claims that we do not show due to space constraints.

```
check_cawe20(s: system) <=
    ** "The system " s " correctly validate all inputs (CAWE-20)" **
    system_has_validator(s) andthen all_targets_receive_validated_inputs(s)
```

Listing 1: Generic Formal Rules for CAWE-20

3 DETECTING A WEAKNESS (CAWE)

Our architecture-first approach for detecting architectural weaknesses relies on the *CAWE-Models* described in the previous section. Our assumption is that a software architect selects a *CAWE-Model* while performing architecture analysis and then traces it to the system's architecture (modeled using AADL or SysML) to find an architectural problem. However, in practice, manually mapping such conceptual graphs to an architecture model is very difficult without appropriate tool support. Therefore, it is important that architects have a tool to automatically detect architectural tactics adopted in the system as well as trace and map a CAWE-Model associated with those tactics to the model. The existence of such a mapping between a CAWE-Model and elements in the architectural design indicates the presence of an architectural weakness. To fulfill this need, our technique is developed as a plug-in to architectural modeling IDEs and it performs three major steps: (i) *annotation* of the system components which implement a tactic; (ii) *embedding* the *formal rules* from CAWE-Models into the annotated models; and

(iii) analysis of the augmented models to determine if an instance of a weakness exists in the system. These steps are detailed below.

3.1 Architectural Models Annotation

As previously mentioned, we assume that architects had created architectural models and provided them as inputs to our technique. To detect weaknesses, we first identify the components that are used for implementing a given architectural tactic as well as their specific roles. We refer to these components involved in the implementation of an architectural tactic as ***"tactical components"***.

Our technique will have a user-friendly interface that allows the system's engineers to manually annotate their architectural models. Specifically, the engineer can adjust three types of metadata to these annotations: (i) the tactics being implemented, (ii) where they are implemented (the components), and (iii) what is the role of each of these components in the tactic. As a result of this first step, we obtain an **annotated version** of the architectural models provided as input to the technique. The annotated version explicitly indicates which components implement an architectural tactic (**tactical components**) and their **roles**. For AADL, we create a property set file containing the definitions of the annotations and rules which can be imported to the architectural model. For SysML, the tags are created as standard UML stereotypes in the format ≪ *tactic.role* ≫ (e.g. ≪ *Authorization.EntryPoint* ≫).

3.2 Embedding Rules into the Models

After obtaining the annotated architectural model, we need to identify the subset of CAWE-Models from our CAWE catalog that applies to the system being analyzed. For this purpose, we enumerate all architectural tactics adopted by the system by parsing the annotated model. Subsequently, we query all the CAWE-Models associated with these adopted architectural tactics. Lastly, we insert the rules from these CAWE-Models into the architectural model.

3.3 Resolute-Based Model Checkers

Our approach to automatically detect architectural weaknesses is built on top of the Resolute Environment [9], which is an open-source language and tool for creating assurance cases. Resolute allows its users to define claims which can be verified by analyzing the underlying architectural model. Our approach uses the rules embedded in the previous step and performs an analysis of the system's properties to verify whether the weakness exists. When an architectural weakness is detected, we notify the developer and share the vulnerability location in the model. Furthermore, our technique will also share fix recommendations based on the list of mitigation techniques documented in our catalog of architectural weaknesses (CAWE catalog).

4 CASE STUDY

We conducted an exploratory study on top of an *Attitude and Orbit Control System* (AOCS) [7], which is a system for controlling satellites. A satellite is controlled by a *ground station* through radio frequency links. It receives *telecommands* from the ground station and it replies with *telemetry data* (i.e. the satellite's status and the data collected by its sensors). The *telecommands* control many aspects of the satellite such as its *nominal attitude* and *nominal orbit*. Figure 2 has a partial AADL model of the AOCS system[1], showing a *Telecommand Processing* (TCP) thread that processes incoming *telecommands* and then computes the *nominal attitude*, *nominal orbit*, *manoeuvre command* and *thresholds* which are forwarded to the *Attitude Control Function* (ACF) thread, *Orbit Control Function* (OCF) thread, and *Manoeuvre Execution* (ME) thread, *Failure Detection Isolation* (FDR), respectively.

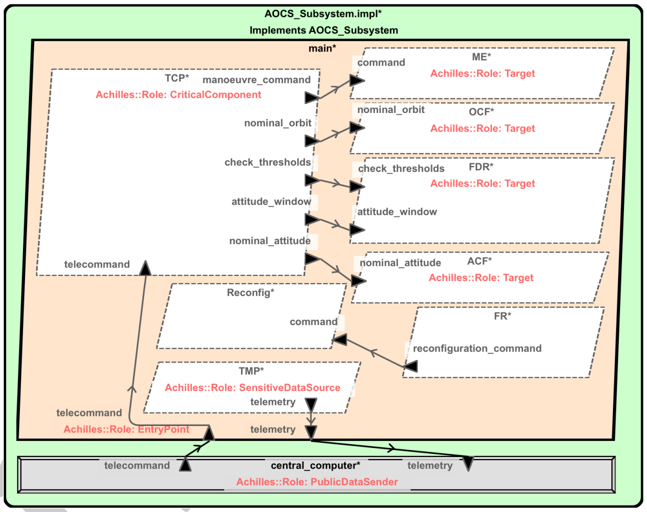

Figure 2: Partial AADL Model Diagram for the AOCS [7, 13]

— **Step 1: Annotation of the Architectural Model**: Since the AOCS receives and processes remote commands, the *ME*, *OCF*, *FDR*, and *ACF* threads are all involved in the *Input Validation* tactic [10, 19] and they are the ***Targets*** of external inputs. We also indicate that the main thread's *telecommand port* is the system's ***Entrypoint***. Moreover, since telemetry data is periodically sent to the ground station, the Telemetry Processing thread (*TMP*) is tagged with the ***Sensitive Data Source*** role. Since the *TCP* thread processes telecommands from the ground station, it is tagged as ***Critical Component*** to indicate that it can only be accessed after proper authentication of the ground station. Listing 2 shows the annotated version of the AOCS' architectural model (annotations are highlighted in light yellow). It is worth mentioning that the architectural model is also modified to import our rules (line 11).

— **Step 2: Embedding the Rules from the CAWE-Models**: After generating the model's annotated version, our technique then infers that the system may be susceptible to the following weaknesses:

- *(CAWE-20) Improper Input Validation*: the AOCS has to enforce that all inputs passed to target components are validated. A lack of validation could be used by attackers to conduct denial-of-service attacks or instruct the satellite to misbehave or even crash.
- *(CAWE-306) Missing Authentication for Critical Function*: the AOCS has to check the authenticity of the ground station before accepting remote commands.
- *(CAWE-319) Transmission of Sensitive Data without Encryption*: the data generated by the *TMP* thread and sent to the central computer to be forwarded to the ground station has to be encrypted.

[1] Due to space constraints we only present in Figure 2 the components under discussion. The full architectural model is available at [15]

Without encryption, an adversary could eavesdrop the traffic and gain sensitive information about the satellite.

Given these potential weaknesses, we embed the CAWE-Model's rules into the system's annotated models for checking the existence of these weakness (lines 191-195 in Listing 2).

```
1   package AOCS
2   public
…     (...)
11    with Achilles;
12    process AOCSprocessing
13      features
14        sensored_attitude: in data port dataaocs::GYR_data;
15        telecommand: in data port dataaocs::telecommand {Achilles::role => EntryPoint;};
…       (...)
64    end AOCSprocessing;
65    process implementation AOCSprocessing.others
66      subcomponents
67        ACF: thread software_aocs::Attitude_Control_Function {Achilles::role => Target;};
68        OCF: thread software_aocs::Orbit_Control_Function {Achilles::role => Target;};
69        TCP: thread software_aocs::Telecommand_Processing {Achilles::role => CriticalComponent;};
70        TMP: thread software_aocs::Telemetry_Processing {Achilles::role => SensitiveDataSource;};
71        FDR: thread software_aocs::Failure_Detection_Isolation {Achilles::role => Target;};
72        ME: thread software_aocs::Manoeuvre_Execution {Achilles::role => Target;};
…       (...)
90    end AOCSprocessing.others;
91    system implementation AOCS_Subsystem.impl
92      subcomponents
93        main: process AOCSprocessing.others;
94        cpu: processor Leon_Processor;
…         (...)
191     annex resolute {**
192       prove(check_cawe319(this))
193       prove(check_cawe20(this))
194       prove(check_cawe306(this))
195     **};
196   end AOCS_Subsystem.impl;
197 end AOCS;
```

Listing 2: Annotated AADL Architecture Model

— **Step 3: Resolute-Based Verification** The technique detects the existence of the aforementioned three weakness because: (i) the AOCS receives *telecommands* and it does not have a subcomponent that performs validation (i.e., it matches the CAWE-Model depicted in Figure 1), (ii) there is not a component for Encrypting data generated by the TMP thread, and (iii) the architecture lacks an *Authenticator* component for checking the ground station's identity.

5 RELATED WORK

Previous work has proposed methods for facilitating the analysis and evaluation of a security architecture [11, 16, 22] and techniques for reverse engineering security design decisions from source code [6]. There are also works that focused on identifying potential threats and vulnerabilities from the underlying architecture [1, 2, 4, 12, 20, 21, 23]. However, they currently can detect very specific instances of architectural flaws related to race conditions [1], anomalous component interactions [23], multi-tier business applications implemented in Java [4] as well as the verification of one single quality attribute (e.g. safety, security) etc) [2, 12, 20, 21]. Although these works can aid architects to identify threats and to appropriately adopt security tactics into a system, such activities may not be enough to develop cyber-resilient systems. Unlike past research, our focus is to tackle the *resiliency* of critical systems, which involves satisfying not only security attributes, but also other quality attributes crucial for the system to successfully prevent, react to, recover from, and adapt to cyber events. Furthermore, there are works on creating architectural models that comply to domain-specific requirements (e.g., avionics) [14, 21], or on creating reusable modeling components [12, 13]. Our approach, however, is intended to be agnostic to the domain of the critical system.

6 CONCLUSION

In this paper, we outlined our ongoing efforts in developing an automated architecture-first verification and reasoning of critical systems. The approach leverage the identification of architectural tactics used in the system and a set of rules in a catalog of architectural weaknesses. We demonstrated our approach in the context of an Attitude and Orbit Control system, which demonstrated the potential for use in practice.

ACKNOWLEDGEMENT

This work is partially supported by Defense Advanced Research Projects Agency (DARPA) under grant number N660011914009.

REFERENCES

[1] S. Al-Azzani and R. Bahsoon. SecArch: Architecture-level evaluation and testing for security. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, pages 51–60. IEEE, 2012.

[2] M. Almorsy, J. Grundy, and A. S. Ibrahim. Automated software architecture security risk analysis using formalized signatures. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 662–671. IEEE, 2013.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley Professional, 3rd edition, 2012.

[4] B. J. Berger, K. Sohr, and R. Koschke. Extracting and analyzing the implemented security architecture of business applications. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 285–294. IEEE, 2013.

[5] D. Bodeau and R. Graubart. Cyber resiliency design principles. *MITRE*, 2017.

[6] M. Bunke and K. Sohr. An architecture-centric approach to detecting security patterns in software. In *International Symposium on Engineering Secure Software and Systems*, pages 156–166. Springer, 2011.

[7] V. Cechticky, G. Montalto, A. Pasetti, and N. Salerno. The AOCS framework. *European Space Agency-Publications-ESA SP*, 516:535–540, 2003.

[8] Q. Feng, R. Kazman, Y. Cai, R. Mo, and L. Xiao. Towards an architecture-centric approach to security analysis. In *13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 221–230, 2016.

[9] A. Gacek, J. Backes, D. Cofer, K. Slind, and M. Whalen. Resolute: An assurance case language for architecture models. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, pages 19–28, New York, NY, USA, 2014. ACM.

[10] M. Hafiz, P. Adamczyk, and R. E. Johnson. Growing a pattern language (for security). In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 139–158, New York, NY, USA, 2012. ACM.

[11] S. T. Halkidis, N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides. Architectural risk analysis of software systems based on security patterns. *IEEE Transactions on Dependable and Secure Computing*, 5(3):129–142, 2008.

[12] T. Heyman, R. Scandariato, and W. Joosen. Reusable formal models for secure software architectures. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, pages 41–50. IEEE, 2012.

[13] J. Hugues. AADLib: a library of reusable AADL models. Technical report, SAE Technical Paper, 2013.

[14] M. Munoz. Space systems modeling using the architecture analysis & design language (AADL). In *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 97–98. IEEE, 2013.

[15] OpenAADL/AADLib. Library of AADL models. https://github.com/OpenAADL/AADLib. (Accessed on 01/20/2020).

[16] J. Ryoo, R. Kazman, and P. Anand. Architectural analysis for security. *IEEE Security & Privacy*, (6):52–59, 2015.

[17] J. C. S. Santos, A. Peruma, M. Mirakhorli, M. Galster, J. V. Vidal, and A. 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. IEEE, 2017.

[18] J. C. S. Santos, K. Tarrit, and M. Mirakhorli. A catalog of security architecture weaknesses. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 220–223, April 2017.

[19] J. C. S. Santos, K. Tarrit, A. Sejfia, M. Mirakhorli, and M. Galster. An empirical study of tactical vulnerabilities. *Journal of Systems and Software*, 2018.

[20] L. Sion, K. Tuma, R. Scandariato, K. Yskout, and W. Joosen. Towards automated security design flaw detection. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE, 2019.

[21] D. Stewart, M. W. Whalen, D. Cofer, and M. P. Heimdahl. Architectural modeling and analysis for safety engineering. In *International Symposium on Model-Based Safety and Assessment*, pages 97–111. Springer, 2017.

[22] E. Taspolatoglu and R. Heinrich. Context-based architectural security analysis. In *13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 281–282, 2016.

[23] E. Yuan and S. Malek. Mining software component interactions to detect security threats at the architectural level. In *Software Architecture (WICSA), 2016 13th Working IEEE/IFIP Conference on*, pages 211–220. IEEE, 2016.