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,ssxsvse,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


ACM Reference Format:

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.
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 weaknesses 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.

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.

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
We conducted an exploratory study on top of an weaknesses (CAWE catalog).

After obtaining the annotated architectural model, we need to identify 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.

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.

\(\text{Figure 2: Partial AADL Model Diagram for the AOCS [7, 13]}\)

---

\(\text{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).}

---

\(\text{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.

---

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

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 Projects Agency (DARPA) under grant number N660011914009.

REFERENCES


