ABSTRACT
Determining whether a software architecture meets its security requirements is an early step in assuring the security of the products developed from the architecture. In this paper, we propose a tool-based technique using an authorization scheme to analyze the security of software architectures. Such technique will serve as debugging support for software architectures to identify the portion in the software architecture that fails to meet the required level. Security is analyzed in terms of its aggregate attributes: availability, confidentiality, and integrity. In this paper, we address confidentiality and show that integrity is measured in a complementary manner to confidentiality. A scenario based approach is taken to analyze security in a software architecture. Our work is implemented in the OSAFE environment and analyzes software architectures modeled using AADL (Architecture Analysis and Design Language).

Categories and Subject Descriptors
D.2.11 [Software Architectures]

General Terms
Design, Security

Keywords
Software Architecture, Debugging, Quality Attribute

1. INTRODUCTION
A secure software system begins with a high quality software architecture. It is often the case that design level defects are the cause of a software system not meeting its non-functional requirements. A model of the software architecture is used to evaluate the initial design of large software systems and it is often difficult to analyze whether the modeled software architecture satisfies the non-functional requirements, i.e. quality attributes. Questioning techniques such as ATAM (Architecture Tradeoff Analysis Method) [3] can be used to analyze whether a specific quality attribute is met by the software architecture. However, such techniques are better at determining a failure to meet a quality requirement than they are at determining what portion of the architecture fails to measure up. Architecture debugging techniques are needed to supplement testing techniques such as the ATAM.

Security is considered by some to be a composite quality attribute composed of three sub-qualities [7].
- Availability: Software system’s amount of uptime.
- Confidentiality: Allowing only authorized users to access the confidential data or perform protected operations of a software system.
- Integrity: Unauthorized data changes are not allowed.

We analyze an architecture for security by analyzing each of these sub-quality attributes separately. Due to space limitations we will only address confidentiality and show how a complementary technique will work for integrity.

Our technique for software architecture definition is a chain that begins with developing a set of use cases. The use cases provide scenarios that are the starting point for a responsibility-driven architecture model captured in ArchE, an expert architecture assistant developed by the Software Engineering Institute (SEI) [11]. This architecture model serves as the input to a more detailed model written in the Architecture Analysis and Design Language (AADL) [2]. It is the AADL model that contains sufficient detail to analyze.

Debugging is a common activity during program development but not so common during architecture design. Debugging is a search for the location of a fault in the architecture definition given that testing has encountered a failure indicating there is a fault. In an architecture, unlike a program, a fault may not be a single incorrect statement. It may be an incorrectly applied reference architecture or other decision that affects a subsystem of the product.

The main contribution of this paper is a description of our technique for debugging a software architecture based on the presence of failures to meet confidentiality requirements, one of the components of security. We present a tool-based technique for debugging a software architecture given that we know the architecture fails to provide the level of security required. We require that the software architecture be modeled using AADL (Architecture Analysis and Design Language). In the remainder of this paper we present our
technique for finding where an architecture fails to meet its confidentiality requirements.

2. BACKGROUND

Software architecture is a high-level design that describes the overall structure of a system [3]. It is often the first software artifact created from the requirements document. A software architecture should satisfy all the functional requirements as well as nonfunctional requirements for the system. To document the software architecture of a system, we use an architecture description language, such as AADL [2] (Architecture Analysis and Design Language). AADL is a standard of SAE (Society of Automobile Engineers) and is used in many large scale projects in the automotive and avionics industries.

One of the many advantages in having a well-constructed software architecture is that early design analyses can be performed based on the software architecture. The software architecture behavior can be analyzed and predicted before implementing the actual system. The SEI provides an open source tool kit called OSATE (Open Source AADL Tool Environment) that provides a set of architecture definition and analysis tools, including tools that analyze the static properties of an architecture such as schedulability and flow latency. Our tool extends OSATE to create a plugin to perform security quality attribute analysis for confidentiality and integrity.

Much research has been performed to define architecture level analyses. In [10], scheduling and memory requirement analysis of AADL models are investigated. The Ocarina toolset [6] provides mechanisms to use a Petri Net representation of the AADL model's behavior to check for deadlocks and inconsistent states. Simulation tools such as ADiS [1] can simulate an AADL model to analyze its behavioral properties. The goal of these analysis tools are the same, which is to analyze the architecture before implementation in order to find and remove any defects at the design level where it is easier and cheaper. Defects that are not removed during the design phase will propagate down to later stages of software development, which increases the cost of fixing the defect when compared to fixing the same defect at the design phase [9].

3. CONFIDENTIALITY

Our approach to debugging confidentiality relies on our approach for measuring confidentiality. We measure confidentiality using a technique that requires only factual knowledge of the architecture and the products to be produced from the architecture. The architect assigns a value to the read authorization property of each element in the architecture that indicates the authorization level required to read the data maintained within the element. This information should be available from the authorization scheme that is used by the organization defining the architecture and from the data confidentiality requirements. The use of an authorization scheme aids in controlling access to a resource and determining what access level is required to get to the resource, and is reported to be useful in the security domain [4]. This approach can be applied to either the logical or physical architecture.

A reasoning framework for confidentiality, used in ArchE, traverses paths through the architecture for a set of scenarios [7]. For each scenario the actor exercising the scenario has an authorization level that is compared to the allowable authorization of the architectural elements along the scenario path. As long as the authorization levels of the elements are less than the authorization level for the actor, the scenario maintains confidentiality.

A second method of measuring confidentiality measures whether a breach of confidentiality “will” occur [5]. This is usually expressed as a risk or probability of occurrence. In this approach a risk value is assigned to each node where the possibility of breach is located. This results in a more complex measure that requires judgment about likelihoods of attack. This is addressed by using attack patterns from the literature, but as types of attacks change the evaluation of a specific architecture would change as well.

We have chosen our approach, based on facts about the product requirements, because our purpose is to give actionable advice to architects. The second approach attempts to address factors - the behavior of humans - that are beyond the control of the architect. It is the case that the risk-based approach can tell the architect which of the confidentiality breaches will be most costly, but only if the actual pattern of attacks is the same as the assumed pattern. We believe the simplicity of our measure and its factual basis gives the most guidance to architects with the minimum of assumptions about the architecture model.

The meaning of any quality attribute is tied to the context in which the measure is collected and the technique for collecting it. Confidentiality and integrity are often discussed in two very different contexts. Our definition of confidentiality and our method for collecting a measure is based on whether a breach in confidentiality “can” occur. We examine the paths through the product that correspond to scenarios of use and determine whether any of the activities along the path can be accessed by a user who does not have sufficient authorization. This results in a fairly simple, Boolean result that is based on facts defined in the product requirements.

4. METHOD

The security requirement of a software system is specified during requirements elicitation. Any reasoning framework defines a general scenario and the concrete scenarios are derived from it. We use a use case analysis and use the appropriate scenario type within each use case. The security quality attribute goals should be defined in terms of concrete values for availability, confidentiality, and integrity. In our case that means giving specific authorization levels for different types of information. We have defined an AADL Property Set that allows the specification of the authorization levels for each architectural element.

![Figure 1: Confidentiality and Integrity Attribute Values](image)
The ordinal scale represents the access level that is required to invoke the read or write operations in a particular component in the software architecture. The authorization scale is used during the design of the software architecture in AADL. Any thread that reads or writes data should possess an authorization property value greater than the authorization level of the architecture element.

As mentioned previously, we use a scenario-based approach to identify elements for which the confidentiality attribute is greater than the authorization level for a specific scenario. For our purposes, we define a scenario in AADL using an end-to-end flow. An end-to-end flow is defined as a “logical flow of information from a source to a destination through a sequence of threads that process and possibly transform the information” [2]. Using an end-to-end flow, we can describe a use case scenario of an actor’s activities and determine if the sequence of activities are allowable by checking the access levels specified in the path traveled by the flow. Each end-to-end flow also specifies the authorization level that has been given to the actor of the scenario as shown in Figure 2.

```
process implementation exp.impl
  subcomponents
    T1: thread prod.default;
    T2: thread recv.default;
    T3: thread recv.alt;
  connections
    conn1: data port T1.pd -> T2.pd;
    conn2: event port T2.pe -> T1.pe;
    conn3: data port T1.pd -> T3.pd;
  flows
    ETE1: end to end flow
      T1.fs1 -> conn1 -> T2.fsink {
        CUSE::readAuthorization => 3;
      };
    ETE2: end to end flow
      T1.fs1 -> conn3 -> T3.fsink {
        CUSE::writeAuthorization => 7;
      };
    ETE3: end to end flow
      T2.fs1 -> conn2 -> T1.fp1
              -> conn3 -> T3.fsink {
        CUSE::readAuthorization => 4;
      };
end exp.impl;
```

Figure 2: Example of End-to-End Flows with Actor’s Access Levels

We have built a prototype tool in which architects using our approach can easily analyze if a given confidentiality or integrity quality attribute is satisfied. The tool is built as an AADL plug-in and, when invoked on an AADL model, any end-to-end flow specified in the model is traversed and contributes to the output in Figure 3, which identifies any access level violations. If a violation is found, detailed information is given to aid the architect in identifying the location of the violation. The results are given in a table format, as in Figure 4, to help in comparing among the scenarios.

Once the tool has informed the architect that a confidentiality or integrity violation has occurred the architect will then have to find its cause in order to fix or refine the software architecture model to meet its quality attribute requirements. This is analogous to debugging in programming. The cause of a confidentiality or integrity violation can surface in the following ways.

- Scenario error: A scenario described through an end-to-end flow may not contain the correct or intended sequence of activities to be performed.
- Access level is too high: An authorization level may be set to be higher than intended.
- Scenario actor’s level is too low: An authorization level given to an actor of a scenario may be too low to perform the activities in a scenario.
• revise the AADL model by eliminating some of the new elements that result from the division.

The same scenarios are run against the revised model and this process is repeated until the desired quality attribute is satisfied by the software architecture model.

Defining and using the authorization scheme in the software architecture description adds the following capabilities in aiding the architect to produce a high quality architecture with most defects uncovered at the design phase.

• Architecture slice: Given an AADL model using the authorization scheme and a failed scenario, we can obtain an architecture slice. An architecture slice shows just the parts of the software architecture description that affect the variables or events of interest during the execution of the architecture [8]. For our purposes, an architecture slice will be showing the parts of the AADL model in which the sequence of activities taken for a scenario are shown, eliminating any parts that are not involved in the scenario’s path taken. This is particularly useful in large AADL models where reducing the search space is helpful in narrowing down the location of an error. In addition, given several failed scenarios for an AADL model, we can compare the architecture slices of all failed scenarios, and determine any similarity in the failing location to quickly identify the problem causing component or connector in the architecture.

• Making queries: Given an AADL model using the authorization scheme, architects have the ability to perform queries to aid in understanding the possible travel paths in the architecture. An architect may make a query that finds all possible travel paths given a specific actor authorization level. This helps to understand what paths are possible for the different levels of authorization specified in the architecture. In addition, such queries help in quickly identifying unwanted paths traveled by a specific authorization level and use the architecture slice to identify the portion in the architecture to refine.

• Handling multi-scale information: A software architecture description may have varying levels of details. The authorization level may appear at the top level component description of the architecture and also at the detailed inner level of a component. We do not expect all architecture descriptions to have a uniform level of detail and, as such, our scheme can accommodate for the varying levels of detail that are present in the architecture.

5. SUMMARY

With the use of our authorization scheme and our tool-based approach, software architects have the capability to easily identify any shortcomings in the architecture description related to the security quality attribute. Finding the location of the shortcoming in the architecture description can be accomplished by looking at the output given by the tool or by getting the architectural slice for the failed scenario. Once the location is found, the architect refines the architecture description, which can be as involved as changing the entire architecture to use a different architectural pattern. Such early determination of the satisfiability of a security quality attribute and refining the software architecture in meeting the security quality attribute goals helps in improving the quality of the software architecture. Ultimately, this positively influences the quality of the system that is being built from the software architecture.

6. REFERENCES