Clustering Functional Sections Using Finer Grained ACDC

A Rationale Document

By Zhaoqi

Team Members

Zhaoqi Zhu

Tianhang Liu

Pengxinag Zhu

1. Background

Architectural recovery refers to the process of retrieving a software system's architecture from implementation level artifacts. Among many recovery methods, ACDC is a widely used technique that relies on structural patterns of the implementation. It can help identify different architectural components of a system. However, because ACDC relies on java class-level dependencies, it cannot cluster sub-class-level elements, such as methods, which are essential to identify security-related architectural decisions which usually span over several classes or even components.

2. Finer Grained ACDC with BFS Proximity to Cluster Functional Sections

	Dependency level	Clusters
Original ACDC	Class	Arch Components
Finer ACDC	Method	"Functional Black Boxes"
Finer ACDC with BFS proximity	Method	"Functional Black Boxes" further broken down into "Functional Sections"

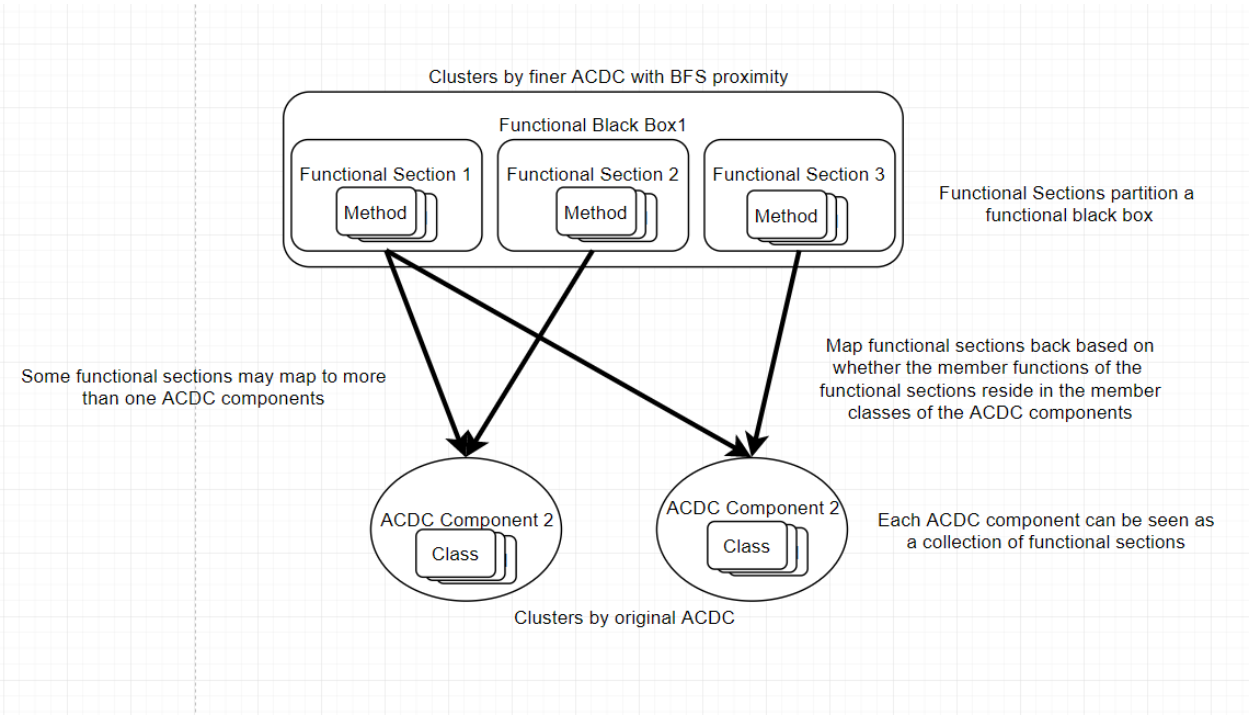
As already mentioned, the original ACDC relies on class-level dependencies, so the clusters are architectural components (hereafter referred as "ACDC components"). In this project, our team proposes that we can modify ACDC to take method-level dependencies as input and output clusters where the member elements are methods rather than classes. We know from the original ACDC paper that in each cluster, all the members are accessible only through a "dominator." [1] As such, we can say that the member methods in the finer version of ACDC are also only accessible through the dominator method of that cluster. In other words, all the member methods of a cluster are "hiding behind" the dominator method of that cluster. We can thus understand those method-level clusters as "functional black boxes."

To bring this modification to ACDC a step further, our team is also adding the proximity of the methods in the dependency graph to consideration. The idea is, although many methods may end up being clustered to the same cluster in the finer ACDC, some of them are closer to each other while some others are scattered away from each other in the call graph. We can imagine those methods as forming small chunks while the chunks themselves can be scattered widely. As such, we added a "BFS proximity" patch to ACDC to identify those chunks, to which we give the name "functional sections". For example, in Tomcat 8.5, a functional black box identified by finer ACDC is "org.apache.catalina.realm.RealmBase.ss." This cluster is further broken down into 14 functional sections from "org.apache.catalina.realm.RealmBase.ss1" to "org.apache.catalina.realm.RealmBase.ss14." For more explanation on how this BFS proximity patch works, please refer to the readme file on GitHub.

Functional sections are very useful when it comes to recovering security-related architectural decisions. Functional black boxes can be understood as a collection of similar behaviors which together achieves a focused functionality. Since functional sections further break down functional black boxes based on the methods' proximity to each other in the call graph, they can be understood as the different aspects of the functionality provide by the black box. For example, "org.apache.catalina.realm.RealmBase.ss" is a functional black box that deal with the entire security

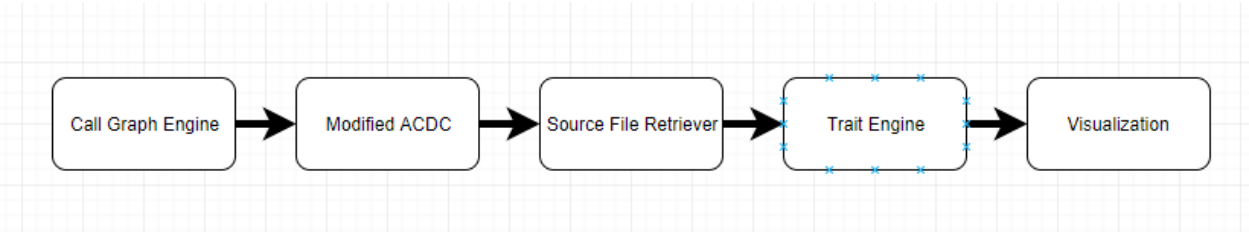
functionality of user credential handling/authentication, then the different function sections or aspects are “checking passwords,” “storing passwords,” “encoding passwords,” etc. As we can see, functional sections are very helpful as to accurately identifying where exactly, at the method level, are the implantations of security decisions and how they are sub-divided into different aspects. This information can be used both when we want to understand the security design of a system that we have never worked on before, and when we want to locate vulnerabilities and fix them.

The functional sections are a new conceptual idea which does not necessarily correspond to any existing architectural abstractions. However, this is ok. As we will talk about in the next section, we can link the functional sections back to the original ACDC components. Below is a diagram that explains the relationships between all the concepts mentioned above.



3. An Automated Pipeline

Our project does not stop at modifying ACDC itself; rather, we aim to build an entire automated pipeline which can analyze a target system expressly.



As we can see from the diagram, there are five components in the pipeline.

Call Graph Engine is a tool that takes any jar files as input, and outputs the method-level dependencies of the target system. This is exactly the input of our finer ACDC with BFS proximity.

Modified ACDC refers to the finer ACDC with BFS proximity we described above. It takes method-level dependencies as the input and output the clusters which are also known as functional sections.

Source File Retriever is a tool that parses the functional sections and outputs the source code of each as retrieved from the target system's code base.

Trait Engine is a tool that helps locate the functional clusters of interests. Specifically, we have written a series of different "traits" which are used to check against the source code of each functional section and score the functional sections. Those traits are essentially security-related characteristics in the source code. The idea is, with different combinations of traits, we can identify the implementation of different security decisions. With this tool, we can do two things: 1) **we are able to rank the functional sections by their scores**. This will help developers quickly locate the implementation of a security decision, so that they can understand the security design of the target system, locate vulnerabilities, fix bugs, etc. 2) **we are able to map the functional sections back to the original ACDC components and compute the latter's scores as the sum of the scores of the functional sections. And then, we can rank the original ACDC components by their scores.** This helps architects check quantitively which components are fulfilling which security decisions. A scenario is that if an architect spots a component which has a high score when it really should not according to the prescriptive architecture, the architect will then know that this component is suffering "functionality envy". There are also many other interpretations of the ranking, we will come up with a full list by the demo time.

Visualization. We have a bunch of visualization tools to help humans interpret the results from Trait Engine. For example, we can visualize the different functional sections interactively. We will dedicate time to this during the demo.

For a more detailed explanation of the five components and the instruction on how to run them, please refer to the readme file in the GitHub repository.

4. What is working?

By Friday, we have finished all the tools in the pipeline, prepared all the data to run them, and uploaded everything to GitHub. We plan to use the extra days after the deadline to come up with more traits, identify more security related decisions, revise the code, etc.

5. What's the Security-Related Architectural Decision You Picked?

This document aims to explain the rationale of the entire project. For a brief example, please refer to the other document written by my teammates.

Reference:

[1] V. Tzerpos and R. Holt, “ACCD: an algorithm for comprehension-driven clustering,” *Proceedings Seventh Working Conference on Reverse Engineering*.