# Tool-Assisted Componentization of Java Applications

Mahmoud M. Hammad
*Software Engineering Department*
*Jordan University of Science and Technology*
Irbid, Jordan
m-hammad@just.edu.jo

Ibrahim Abueisa
*Software Development Engineer*
*Amazon Inc.*
Amman, Jordan
abueis@amazon.com

Sam Malek
*Informatics Department*
*University of California, Irvine*
Irvine, USA
malek@uci.edu

***Abstract*—Many popular object-oriented (OO) programming languages, such as Java, do not provide explicit support for *architecture-based development*, i.e., do not provide programming-language constructs that are at the granularity of architectural constructs, such as components and ports. The gap between how engineers design their systems and how they implement them has been one of the leading causes of architectural drift—a situation in which the prescriptive architecture (the designed architecture) does not match the descriptive architecture (the implemented architecture). To mitigate this challenge, in its ninth iteration, Java introduced the concept of *Java Platform Module System (JPMS)*, which for the first time provides explicit implementation-level support for well-known architectural constructs, such as components (called *modules*) and ports (called *module directives*). Despite this, the majority of existing Java applications (apps) are still purely OO programs that do not make use of the new constructs, because converting them to well-structured component-based (CB) programs is a tedious and error-prone task. In fact, prior research has shown that when engineers convert OO apps to CB apps, they tend to be highly over-privileged, i.e., components are granted more access privileges than they actually need. To mitigate these challenges, we have developed *OO2CB*, an approach for conversion of an OO Java app to a least-privilege CB Java app. *OO2CB* employs component recovery techniques to assist the developer in determining a given OO app's components. It then statically analyzes the source code of the app to determine the dependencies among its recovered components and the required port types for facilitating their interaction. Finally, *OO2CB* generates a functionally equivalent CB app that satisfies the least-privilege security principle. Our experiments on several large real-world OO Java apps corroborate the effectiveness of *OO2CB*.**

## I. Introduction

A software system's architecture consists of a set of principal design decisions governing the system [1]. Ensuring that the *prescriptive architecture* (the designed architecture) matches the *descriptive architecture* (the implemented architecture) is of utmost importance, and particularly challenging when a system evolves [1], [2]. One of the leading causes of architectural mismatch, known as architectural *erosion* or *drift*, is that many popular programming languages, such as Java, support Object Oriented (OO) programming but not architecture-based development. An architecture of a software system is typically conceived in terms high-level constructs, such as components, connectors, and ports, but implemented in terms of a different set of low-level constructs, such as packages, classes, and methods. This gap makes it exceptionally challenging to keep the implementation of a software system in sync with its architecture.

In a promising development, Java, arguably the most widely used OO programming language, released the Java Platform Module System (JPMS) in version 9 of Java [3]. JPMS explicitly supports architecture-based development, allowing the software engineers to define components (called *modules*), ports (called *module directives*), and interfaces.[1] In addition, Java 9 itself is also modularized, meaning that one can require subset of the Java Runtime Environment (JRE) system modules. The benefits of JPMS are threefold: (1) better encapsulation of the software in terms of its architecturally significant elements, (2) reduction in software bloat by reducing the size of the code loaded at runtime, and (3) improved security through the reduction of attack surface.

Despite its promise, approximately 5 years after the public release of Java 9, the great majority of existing open-source OO Java applications (apps) have not been converted to a component-based (CB) equivalent using JPMS.[2] We believe this can be attributed, at least partly, to the lack of tool support to help developers modularize their OO Java apps. Manually porting an OO Java app to JPMS is a cumbersome task, as it requires the developer to fundamentally refactor the code base and to carefully consider the dependencies among the system's components. Prior work [4] has shown that when developers manually refactor their apps to use the newly introduced JPMS constructs, they tend to take the easy way out, which is to simply open each module to the rest of the system, thereby creating highly over-privileged apps, i.e., a situation in which components (i.e., modules) are granted more access than they need to function. As a result, none of the benefits associated with the adoption of JPMS is fully materialized.

To systematically overcome the aforementioned problem, we have developed *OO2CB*, a system for conversion of a Java OO app to a Java CB app, and determination of the least-privilege architecture for the resulting app. A *least-privilege architecture*

[1] Explicit support for software connector as a first class construct is still missing.

[2] In our analysis of over 1,300 open-source Java projects, we found that only 33 are utilizing JPMS capabilities. This comports with the results reported in prior work [4].

is an architecture in which each component is granted the exact privileges, in terms of inter-component communications as well as the required JRE modules, it needs to provide its functionality[1].

More precisely, given an OO app in Java, *OO2CB* (1) aids the developer with determining the components of the app using well-known component recovery tools, (2) leverages static program analysis to determine the dependencies among the app components as well as the JRE components, (3) determines the required ports between the components, and finally (4) generates the corresponding CB version of the app. Our evaluation of *OO2CB* on large real-world OO Java apps corroborates the effectiveness of our approach, including the ability of *OO2CB* to accurately determine the least-privilege architecture of OO apps and to automatically create their corresponding CB versions.

The remainder of this paper is structured as follows. Section II provides an illustrative example to motivate our work and to illustrate our approach. Section III describes *OO2CB* in more details and Section IV provides its implementation. Section V evaluates *OO2CB*. VI shows the threats to the validity of our approach and results. Section VII overviews the related research efforts. Finally, Section VIII concludes the paper and sheds light on future directions.

## II. Illustrative Example

This section provides an illustrative example of two versions of a Java app. The first version is a Java 8 OO app and the second version is a Java 9 CB app. The Java app is part of a simple university system that we have developed. The illustrative example is used to motivate our work and illustrate our approach.

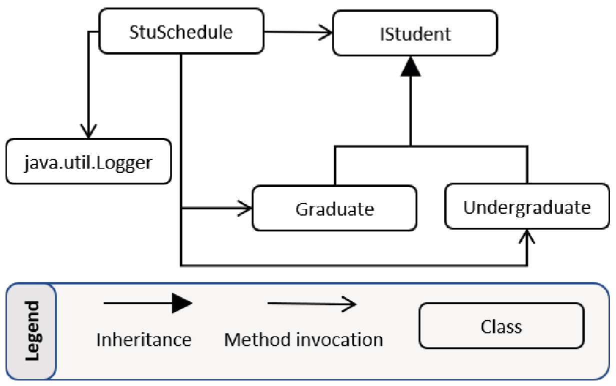


Fig. 1. Structure of the object-oriented (OO) application.

Figure 1 shows the descriptive architecture as a class diagram of the Java 8 OO application. The class `StuSchedule` builds a suggested schedule for a student and logs any information to a log file using the Java 8 `java.util.Logger` class. A student can be either an `Undergraduate` or a `Graduate` student. Both of these two classes implement the Java interface `IStudent`. All of these classes belong to different packages but we removed them to reduce the clutter and more importantly because, in Java 8, if a class is a *public* class, all other classes in the system can communicate with it regardless of the packaging structure.

As shown in Figure 1, the OO structure shows only the class communications in terms of method invocations and inheritance relationships but lack the architectural constructs such as components and ports. To get the many benefits of the architectural development provided by the JPMS, we have converted this OO app to a CB app using our approach.

The Java Platform Module System (JPMS) [3] is a central part of the project Jigsaw [5]. The idea of the JPMS is to provide a scalable module system for the Java platform. The JPMS allows software engineers to build their applications in terms of architectural constructs: components (called *modules*) and ports (called *module directives*). These constructs provide a higher level of abstraction than Java packages and classes, as in the Java 8 and prior versions. The Java 9 JRE is also modularized, meaning that an app can require part of the JRE instead of requiring the whole JRE, resulting in a reduction of the JRE runtime size. Moreover, Java 9 allows developers to create a *custom runtime image* of their CB applications using *jlink* tool [6]. The custom runtime image contains the app, 3rd-party libraries, and only the required Java 9 modules and their dependencies, hence reducing the size of the CB app. This self-contained executable custom runtime image can run on an operating system (OS) even if the required JRE version is not installed on that OS.

In the JPMS, each module has a configuration file, or descriptor, named “module-info.java” file, which defines the ports and the dependencies of this component with other components; both the system (JRE) modules and the user-defined components. The JPMS supports various types of ports. These ports allow a component to export part of the services it provides or all of them. Similarly, a component can require services from other components.

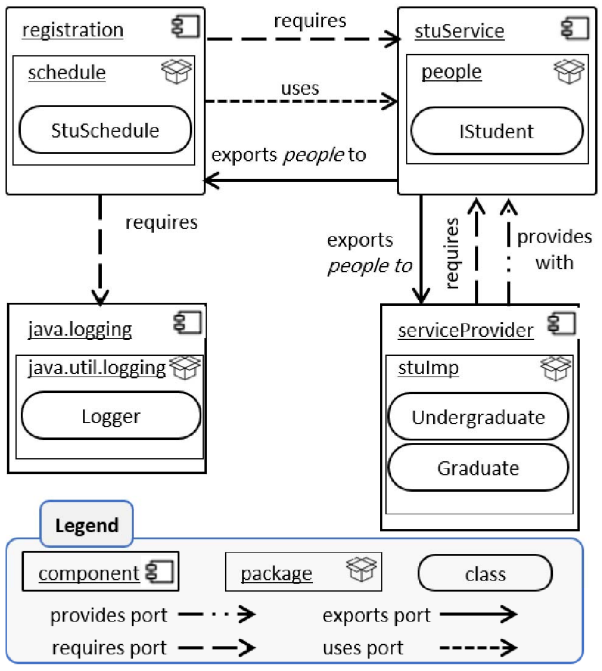
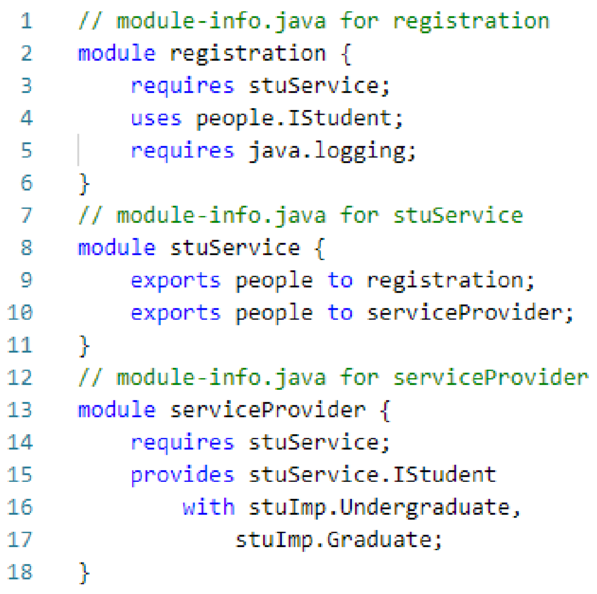


Fig. 2. Component-based (CB) application.



```
1   // module-info.java for registration
2   module registration {
3       requires stuService;
4       uses people.IStudent;
5       requires java.logging;
6   }
7   // module-info.java for stuService
8   module stuService {
9       exports people to registration;
10      exports people to serviceProvider;
11  }
12  // module-info.java for serviceProvider
13  module serviceProvider {
14      requires stuService;
15      provides stuService.IStudent
16          with stuImp.Undergraduate,
17              stuImp.Graduate;
18  }
```

Fig. 3. module-info.java file

Figure 2 depicts part of the component-based architecture of the university system with three components, while Figure 3 depicts the "module-info.java" files of the three components. In practice, a module is likely to have more Java classes and dependencies than the modules of our illustrative example.

As shown in Figure 2, the `stuService` component contains the `IStudent` interface inside the `people` package. In order for the `Undergraduate` and `Graduate` classes from the `serviceProvider` component to implement the interface, the `stuService` needs to export the `people` package either to public using the `export` port or to a specific component using the `exports to` port. Here, the latter satisfies the least-privilege security principle, whereas the former may violate it, if not all of the components require the `people` package in providing their services. Line 10 of Figure 3 shows the logic for creation of the `exports to` port. In addition to the `exports to` port, the `serviceProvider` component needs to define two more ports. One port to require the `stuService` component as shown in Line 14 of Figure 3 and another `provides with` port to provide the functionalities of the `IStudent` interface using the `Graduate` and `Undergraduate` implementation as shown in Lines 15 – 17 of Figure 3 .

As depicted in Figure 2, the `registration` component contains the `StuSchedule` class inside the `schedule` package. Since the `StuSchedule` class uses the `IStudent` interface from the `people` package in its code, the `registration` component needs to define a `requires` port in its "module-info.java" file as shown in Line 3 of Figure 3 and the `stuService` component needs to define an `export` port to export the `people` package to the `registration` component as shown in Line 9 of Figure 3. Now since the `IStudent` is an interface, the `registration` component also needs to define a `uses` interface as shown in Line 4 of Figure 3.

Finally, since `StuSchedule` class communicates with the `java.util.Logger` Java 8 class as depicted in Figure 1, the CB app needs to define a `requires` port to be able to use the JRE module named `java.logging` in Java 9.

The above example illustrates the steps one would take to transform an OO app to its equivalent CB app in Java. While this may be a straightforward process for a small app, it is indeed a cumbersome manual process when applied to a large OO app. It requires the developers to manually determine (1) the components of their app, (2) the dependencies among those components, (3) the JRE modules utilized by the app, and (4) the types of dependencies among components. Upon that determination, developers have to refactor their code (i.e., create the proper directory structure and prepare the *module-info.java* files), prior to building their apps in Java 9. Unfortunately, in practice, many developers simply place all of their code in a single module and open that module to the public to rapidly build their apps in Java 9, thereby failing to reap the benefits of a truly CB architecture.

## III. Approach

This section describes our approach, named *OO2CB*, for assisting software engineers to determine the least-privilege architecture of an OO app and converting it to a CB app. The least-privilege architecture has the precise dependencies each component needs to fulfill its functionality. The benefits of the least-privilege architecture are two-fold: (1) reducing the attack surface of the generated CB app and (2) reducing the software bloat. As depicted in Figure 4, our approach consists

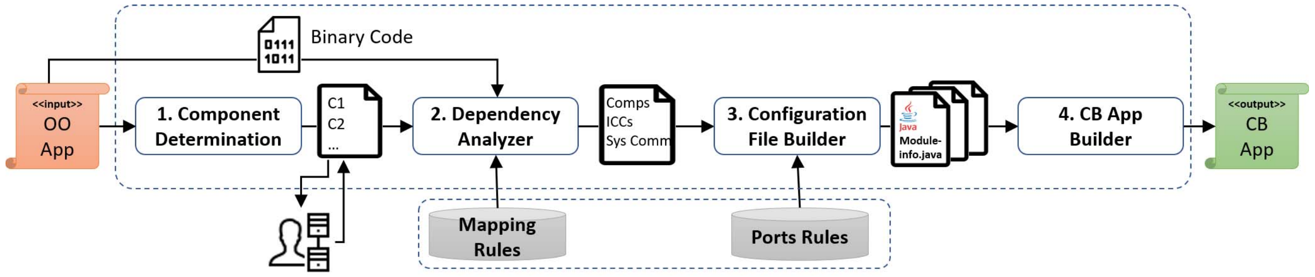


Fig. 4. Overview of *OO2CB*

of four steps to convert an OO app to the corresponding CB version of the app. The rest of this section describes each step in detail.

## A. Step 1: Component Determination

In order to build a CB app, we need to determine the components that constitute the app and the classes that belong to each component. Large body of previous research efforts investigated and developed various tools and techniques to automatically determine software components from the binary or source code of OO apps including [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], and [21]. These research efforts followed various approaches to automatically determine the components of an OO app such as having accurate and up-to-date UML diagrams of the system, rule-based techniques, machine learning models, and clustering algorithms. However, the existing tools are known to suffer from inaccuracies, and different tools sometimes even return different components for the same system [14].

To determine the components of an OO app, in this step, *OO2CB* supports three options to assist developers.

- *Option #1.* Automatically determine the components of an OO app using a component recovery tool. The tool can be added as a plug-in to our approach.
- *Option #2.* Allowing software engineers to adjust the results of the leveraged component recovery tool. In this option software engineers can add or remove components and correct the classes in each component.
- *Option #3.* Software engineers can manually determine the components and the classes that belong to each component without running any component recovery tool.

We integrated a well-maintained component recovery framework implemented by Garcia et al. [14], called ARCADE, in our implementation of *OO2CB*. The ARCADE framework utilizes several well-known component recovery tools including Architecture Recovery using Concerns (ARC) [15], Bunch [16], scaLable InforMation BOttleneck (LIMBO) [17], Algorithm for Comprehension-Driven Clustering (ACDC) [18], a tool implemented by Corazza et al. [19], and Weighted Combined Algorithm (WCA) [20]. Although any of the above-mentioned recovery techniques can be used in *OO2CB*, in our experiments we have used the results of the ACDC tool, since as concluded in prior work [14], it is one of two best performing component determination tools.

We also attempted to integrate with another recovery technique, called ROMANTIC [21], that employs a type of class clustering algorithm. Unfortunately, even with the help of the authors we were unable to run it correctly and obtain reliable results.

## B. Step 2: Dependency Analyzer

Prior work [4] has shown that when manually porting a Java app to use JPMS features, developers do not determine the accurate dependencies between components and they tend to open the entire app as one module. Failing to determine the precise dependencies between components not only increases the attack surface of the app, but also contributes to unnecessary memory usage, i.e., software bloat. Therefore, in this step, *OO2CB* takes the recovered components and the binary code of an OO app as input and determines its *least-privilege architecture* as output. The *least-privilege architecture* is an architecture in which each component is only granted access to components and resources that are needed to provide its functionality. The *Dependency Analyzer* component identifies two categories of dependencies (1) the allowed inter-component communications (ICC) and (2) the allowed communications between the app's components and JRE modules, referred hereafter as the *system communications*.

To determine the exact communications each component requires to provide its services, *OO2CB* first builds a call graph of an OO app, where the nodes are classes and the edges are method invocations. To build the call graph, *OO2CB* generates the Abstract Syntax Tree (AST) of the OO app using the Byte Code Engineering Library (BCEL) [22]. The BCEL is a widely-used Apache library for analyzing Java binary files, i.e., the *.class* files. Then, *OO2CB* traverses the call graph and determines the communications between the various classes and the communications between the OO app and the JRE classes, i.e., the Java 8 classes.

Using the determined components in the first step and the communications between the classes in the call graph, *OO2CB* determines the inter-component communications (ICCs). For example, since the `StuSchedule` class communicates with the `IStudent` interface as shown in Figure 1, the *Dependency Analyzer* of our approach adds a required ICC between the

TABLE I
NUMBER OF CLASSES.

| Item | Count |
|---|---|
| JDK 8 classes | 7,701 |
| JDK 9 classes (JPMS classes) | 19,368 |
| Automatically mapped classes | 7,469 |
| Manually mapped classes | 191 |
| Unmapped classes | 41 |

`registration` component and the `stuService` component to the CB architecture.

As mentioned earlier, the JRE itself is modularized, allowing a CB app in Java to require a part of the JRE instead of requiring the whole JRE with all its components (83 system components in Java 9). Therefore, *OO2CB* determines the required JRE components (JRE modules) and the system communications for each component. However, mapping between Java 8 classes and Java 9 classes is not a trivial task as Java 9 introduced many changes to the structure and the arrangement of the JDK classes. These changes include moving classes between different packages, adding and removing some classes, changing class names with the same functionalities, and changing functionalities of some classes.

A software engineer who wants to manually convert an OO app to a CB app needs to be aware of these changes. Unfortunately, there is no reference document describing all of these changes, turning this into an arduous process for software engineers. To overcome this problem, we semi-manually mapped Java 8 classes to Java 9 classes and stored the results in the `Mapping Rules` repository of our approach, shown in Figure 4.

To conduct the class mapping, our approach automatically scanned Java 8 classes and matched them with the Java 9 classes based on the name and the path of the class. For example, the Java 8 class `com.sun.org.apache.xalan.internal.xsltc.dom.SingletonIterator.java` is mapped to the Java 9 class `com.sun.org.apache.xalan.internal.xsltc.dom.SingletonIterator.java` since they have the same name and path. However, not all Java 8 classes are mapped to Java 9 classes. Therefore, we manually investigated and mapped additional classes through (1) studying a Java 8 class's implementation and looking for it counterpart in Java 9, (2) reading the Java documentation of the Java 8 class and trying to find the matching Java 9 class, or (3) reading online resources for software engineers who faced the same problems and solved them.

At the end of the semi-manual mapping process, some Java 8 classes were not mapped to any Java 9 classes for a variety of reasons. For example, the Java 8 class `com.sun.image.codec.jpeg.JPEGCodec.java`, a factory class for implementing a JPEG image decoder/encoder, cannot be mapped to any Java 9 class, since Oracle has removed it from the JPMS JDK as indicated in an Oracle bug report [23]. The same bug report does not provide any alternative class, instead it indicates that *"For JDK 9, the module system would need to decide how to expose/export this."*

As a summary of the mapping process, given that the Java 8 JDK contains 7,701 classes and the JPMS JDK contains 19,368 classes, we automatically mapped 7,469 Java 8 classes to Java 9 classes. We further manually mapped an additional 191 Java 8 classes. However, 41 Java 8 classes could not be mapped to any Java 9 classes. Table I summarizes the statistics of the mapping process.

Using the `Mapping Rules` repository, *OO2CB* traverses the generated call graph and determines the required JRE system modules as well as the system communications for each component. For example, from the communication between the `StuSchedule` class and the Java 8 `java.util.Logger` class in Figure 1, the *Dependency Analyzer* determines that the `registration` component requires the `java.logging` Java 9 module and adds a system communication between them in the CB architecture.

## C. Step 3: Configuration File Builder

In this step, *OO2CB* creates a configuration file, called "module-info.java" file, for each component. The configuration file contains the ports and the communication privileges for each component. To create well designed ports following the least-privilege security principle, we have carefully defined rules for creating various types of ports.

The following rules are definitions for various types of ports which *OO2CB* applies in this step to create ports for each component and adds them to the configuration file of each component in the CB app.

*Definition 1 (Requires):* Let $C_1$ be a class that belongs to module $M_1$ and $C_2$ be a class or an interface that belongs to module $M_2$. If $C_1$ communicates with $C_2$, then $M_1$ requires $M_2$.
$Requires(M_1, M_2) \equiv (C_1 \in M_1) \wedge (C_2 \in M_2) \wedge \exists\, communicates(C_1, C_2)$

According to Definition 1, *OO2CB* adds a `requires` port to the `registration` component, since the class `StuSchedule` communicates with `IStudent`, by defining objects of this type, and writes the port creation code in the "module-info.java" file of the `registration` component as shown in Line 3 of Figure 3.

*Definition 2 (Exports To):* Let $C_1$ be a class that belongs to package $P_1$ in module $M_1$ and $C_2$ be a class that belongs to package $P_2$ in module $M_2$. If $C_2$ communicates with $C_1$, then module $M_1$ needs to export package $P_1$ to $M_2$, otherwise, $C_2$ will not be able to communicate with $C_1$.
$ExportsTo(M_1, P_1, M_2) \equiv (C_1 \in P_1) \wedge (P_1 \in M_1) \wedge (C_2 \in P_2) \wedge (P_2 \in M_2) \wedge \exists\, communicates(C_2, C_1)$
According to Definition 2, *OO2CB* adds an `exports to` port to the `stuService` component to export the package `people` since the `StuSchedule` communicates with the

IStudent, and adds this port creation rule to the "module-info" file of the stuService as shown in Line 10 of Figure 3. This definition strictly follows the least-privilege security principle. *OO2CB* exports people of stuService component to only the modules that need it, thereby reducing the attack surface.

*Definition 3 (Uses):* Let $C_1$ be a class that belongs to module $M_1$, and $A_2$ be an interface or an abstract class in module $M_2$. If $C_1$ uses $A_2$, then $M_1$ uses $A_2$ in $M_2$.
$Uses(M_1, A_2, M_2) \equiv (C_1 \in M_1) \wedge (A_2 \in M_2) \wedge type(A_2) \in [interface, abstract\ class] \wedge (\exists\ communicates(C_1, A_2))$

According to Definition 3, *OO2CB* adds a uses port to registration component since the StuSchedule class uses the interface IStudent from the stuService and adds the port creation rule to the "module-info.java" file of the registration component as shown in Line 4 of Figure 3.

*Definition 4 (Provides With):* Let $C_1$ be a class in module $M_1$ and $A_2$ be an abstract class or an interface in module $M_2$. If $C_1$ implements or extends $A_2$, then $M_1$ provides $A_2$ with $C_1$.
$ProvidesWith(M_1, A_2, C_1) \equiv (C_1 \in M_1) \wedge (A_2 \in M_2) \wedge type(A_2) \in [interface, abstract\ class] \wedge (\exists\ implements(C_1, A_2) \vee \exists\ extends(C_1, A_2))$

According to Definition 4, since the classes Undergraduate and Graduate in module serviceProvider implements the interface IStudent from the stuService component, *OO2CB* adds a provides with port to serviceProvider to provide the functionality of IStudent through the implementation of Registration. Then, *OO2CB* add the port creation code to the "module-info.java" file as shown in Lines 15 – 17 of Figure 3. In such a case, a component can utilize the java.util.ServiceLoader from the java.base JDK 9 module to load implementations of a service ($A_2$ in our definition).

### D. Step 4: CB App Builder

In this step, *OO2CB* builds a CB app considering the determined least-privilege architecture. It takes the user-defined components along with their classes. Next, it creates an OS directory for each component and adds all the Java packages and classes that belong to the component into the directory. Then, *OO2CB* modifies the header and the import section of each class in every component to make it compatible with the new CB application structure.

Finally, the result of our approach is a compiled CB app following the least-privilege security principle with precise dependencies between components.

## IV. Implementation

*OO2CB* is a Java application that takes a Java OO application as input and determines the least-privilege architecture and builds a CB app as an output. As mentioned earlier, to determine the components of an OO app, *OO2CB* assists developers in recovering the components of the app using well-known component recovery tools. *OO2CB* integrates with the ARCADE framework [14] which has various recovery methods. In the experiments reported in this paper, we have relied on the results of the ACDC, one of the recovery methods in ARCADE, but *OO2CB* can be easily configured to take the results of other recovery methods. After the components are identified, *OO2CB* determines the required ICCs, the needed JPMS modules, the system communications, and the required ports for each component to provide its functionality. The implementation of our approach consists of more than $8,000$ lines of code, excluding the implementation of ARCADE. The full implementation of *OO2CB* is available online as an open-source tool [24].

## V. Experimental Evaluation

This section describes the experimental evaluation of our approach. Our evaluation addresses the following research questions:

- **RQ1** How effective is *OO2CB* in successfully converting real-world OO apps to CB apps?
- **RQ2** How much *OO2CB* is able to reduce the attack surface of converted apps?
- **RQ3** How much *OO2CB* is able to reduce the runtime memory footprint of converted apps?
- **RQ4** What is the performance of *OO2CB*?

To conduct our experiments and evaluate our approach, we utilized five real-world large Java 8 applications. Following are the five applications:

1) Apatche Nutch release 2.3, a mature web crawler relying on the Hadoop data structure.
2) Openjpa version 2.4.1, a widely used open-source Java object-relational mapping (ORM) tool.
3) Apache CXF version 3.1.6, an open-source web services framework.
4) Apache Camel version 2.17.0, an open-source Java message-oriented middleware.
5) Apache Lucene-4.6.1, a widely used open-source search engine library.

In addition to the above, we also utilized the university system discussed in Section II. We use the university system as a ground-truth, since we know the OO structure and the CB structure of the app.

For evaluation, we automatically recovered the components of the Java 8 applications using the ACDC [18] tool. We chose ACDC since, as concluded in the ARCADE framework [25], it is one of two best performing component recovery tools as evaluated on expert-recovered and carefully-verified architectures. In practice, we envision a developer familiar with the source code of an app may adjust the recovered components produced by a recovery method, such as ACDC. For an unbiased evaluation, however, we did not make such adjustments in the results reported here.

TABLE II
RQ1: THE EFFECTIVENESS OF *OO2CB*.

| App | OO App | CB App | | | | Compiled? |
|---|---|---|---|---|---|---|
| | Classes | Comps | Ports | ICC | Sys Comm | |
| University system | 16 | 5 | 23 | 6 | 7 | Y |
| Nutch release 2.3 | 347 | 57 | 257 | 93 | 69 | Y |
| Openjpa 2.4.1 | 1,352 | 95 | 1,358 | 426 | 101 | Y |
| CXF-3.1.6 | 2,958 | 325 | 9,319 | 3,609 | 853 | Y |
| Camel-2.17.0 | 1,414 | 67 | 1,490 | 496 | 138 | Y |
| Lucene-4.6.1 | 1,081 | 41 | 1,490 | 397 | 95 | Y |
| Average | 1,194.7 | 98.3 | 2,322.8 | 837.8 | 210.5 | |

## A. *RQ1: Effectiveness of OO2CB*

This research questions measures the effectiveness of our approach in successfully taking an OO app as input and determining the least-privilege architecture and successfully building a CB app as an output. To do this experiment we ran our approach on the 5 real-world large OO apps and our university system. Table II compares between the structures of the OO apps and the CB apps converted by *OO2CB* and shows if *OO2CB* was able to successfully build a compiled Java 9 CB app.

The `OO App Classes` column of Table II shows the number of classes in each OO app. For example, the CXF has 2,958 classes.

The `CB App` column of Table II shows the structure of the determined least-privilege CB architecture and the created CB app by *OO2CB* in terms of the number of determined components (`Comps`), number of created ports (`Ports`), number of inter-component-communications (`ICC`), and number of communications between the CB app and the JRE system modules (`Sys Comm`). For example, the determined least-privilege CB architecture of the CXF app contains 325 components, 9,319 ports, 3,609 ICCs, and 853 JRE system communications.

As shown in Column `Compiled?` of Table II, *OO2CB* was able to successfully determine and build CB apps for all Java 8 OO apps. The results in Table II corroborate the effectiveness of our approach in determining the least-privilege architecture and successfully building compiled CB apps.

## B. *RQ2: Attack Surface Reduction*

The main purpose of our approach is to assist developers in converting their OO apps to CB apps with precise dependencies. As prior work [4] showed, when software engineers convert their OO apps to CB apps, they tend to create over-privileged architectures in which components are granted more privileges than they need. An over-privileged architecture for an app has security implications, since it increases the attack surface of the app.

By determining the precise dependencies among components, *OO2CB* aids security architects (or security analysis tools) to understand the security posture of the system and reduces the attack surface.

Table III shows the attack surface reduction measured in 3 dimensions for the 6 subject apps.

- `D1. ICC Reduction` column compares between the allowed inter-component communications (ICCs) and the determined ICCs. The allowed ICCs are the communications that are allowed between the components of an app without applying the least-privilege principle. This reflects a situation in which developers leave the modules open, i.e., allowing the modules to freely communicate with one another. The determined ICCs are computed by our approach following the least-privilege security principle. As shown in Table III, the CXF app has 105,300 `Allowed ICCs` and only 3,609 `Determined ICCs`, resulting in a 96.6% ICC Reduction. On average, our approach achieved 87.2% ICC reduction.
- `D2. Sys Comm Reduction` column compares between the allowed system communications and the determined system communications. The allowed system communications are what is allowed by default without determining the exact JRE modules required by each component in the app. The determined system communications are JRE module communications that *OO2CB* found to be necessary. For example, as shown in the table, the Openjpa CB app has 7,885 allowed system communications and only 101 determined system communications, resulting in a 98.7% reduction. On average, our approach achieved 97.9% in the system communication reduction dimension.
- `D3. Required Sys Comp Reduction` column measures the amount of reduction in the required JRE modules, i.e., all the 83 JRE modules vs. the JRE modules that were determined by our approach. For example, the over-privileged Nutch app requires all the 83 JRE system modules, whereas the least-privilege Nutch app created

TABLE III
RQ2: ATTACK SURFACE REDUCTION.

| CB App | Comps | D1. ICC Reduction | | | D2. Sys Comm Reduction | | | D3. Required Sys Comp Reduction | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Determined ICC | Allowed ICC | Reduction (%) | Determined Sys Comm | Allowed Sys Comm | Reduction (%) | Determined JRE Comps | Allowed JRE Comps | Reduction (%) |
| University system | 5 | 6 | 20 | 70.0 | 7 | 415 | 98.3 | 2 | 83 | 97.6 |
| Nutch release 2.3 | 57 | 93 | 3,192 | 97.1 | 69 | 4,731 | 98.5 | 4 | 83 | 95.2 |
| Openjpa 2.4.1 | 95 | 426 | 8,930 | 95.2 | 101 | 7,885 | 98.7 | 10 | 83 | 88.0 |
| CXF-3.1.6 | 325 | 3,609 | 105,300 | 96.6 | 853 | 26,975 | 96.8 | 12 | 83 | 85.5 |
| Camel-2.17.0 | 67 | 496 | 4,422 | 88.8 | 138 | 5,561 | 97.5 | 8 | 83 | 90.4 |
| Lucene-4.6.1 | 41 | 397 | 1,640 | 75.8 | 95 | 3,403 | 97.2 | 7 | 83 | 91.6 |
| Average | 98.3 | 837.8 | 20,584.0 | 87.2 | 210.5 | 8,161.7 | 97.9 | 83.0 | 7.2 | 91.4 |

by our approach only requires 4 JRE system modules, resulting in a 95.2% reduction. On average, our approach achieved 91.4% reduction in this dimension.

The average of attack surface reduction achieved by our approach over the three dimensions is 92.2%

### C. RQ3: Software Bloat

Since JDK is modularized in Java 9, a CB app can require only a subset of the JDK modules that it needs to create a lightweight JRE, and hence reduce its runtime memory requirement. To that end, we evaluated the runtime memory reduction between the over-privileged CB app, an app in which all components have access to all JRE modules, and the determined least-privilege CB app, where each component is granted the exact JRE modules it needs to function.

Table IV shows the runtime memory reduction in the custom JRE. As shown in the table, an over-privileged CXF app that requires all JRE modules consumes, if all modules are loaded, 236 MB of memory, while the CXF app produced by our approach requires only 12 JRE modules, which consumes 95.9 MB of memory, resulting in a 59.4% reduction. As shown in Table IV, on average, *OO2CB* achieves a 65.1% memory reduction in the subject apps. Such a substantial reduction in memory requirement is crucial for deployment in resource-constrained devices, such as IoT devices.

### D. RQ4: Performance of OO2CB

We measured the performance of our approach in determining the least-privilege component-based architecture of OO apps and successfully building CB versions of the 6 apps. We conducted our performance experiments on a Lenovo laptop with 2.2 GHz Intel Core i7 8th generation and 12 GB DDR4 RAM. Table V shows the results in minutes in the `Performance` column. As shown in the table, it took our approach 14.56 minutes to determine the least-privilege architecture and successfully build a compiled CB app of the Camel application. On average, to determine the least-privilege architecture of a large OO app and building the CB app, *OO2CB* takes 2.8 minutes. These results corroborate the efficiency of *OO2CB* for practical use.

## VI. THREATS TO VALIDITY

The main threat to internal validity of our work is the false positives caused by the static analysis of our approach. Static analysis results may overestimate the communications between components which might lead to granting a component more privileges than it needs in the resulted CB app. To reduce this threat, we leveraged the BCEL [22], a widely-used library in the industry to analyze Java apps. Notably, our approach is not susceptible to false negatives, corroborated by the fact that we were able to successfully build large open-source software systems after they were componentized.

The selection of our Java apps is a threat to the external validity of our results. To mitigate this threat, we included OO Java apps that are large in size and widely used in the industry. Another threat to external validity is whether the ports we create are comprehensive and enough to create CB apps. To overcome this threat, we defined all ports except the ones responsible for Java reflection and dynamic class loading techniques, i.e., the `open` and `opens with` ports. However, Java reflection and dynamic class loading are infrequently used in Java apps. Part of our future work involves extending support to these additional port types.

Another threat to the external validity of our experimental results is the use of component recovery tools. To reduce this threat in our experiments, we leveraged the best-performing component recovery tool, i.e., ACDC [18], as concluded in prior comparative studies of recovery techniques [14].

## VII. RELATED WORK

Our work is related to large body of previous research efforts in recovering components and converting OO systems to CB systems. In this section, we briefly discuss prior efforts in light of our work.

Xia Cai et al. [26] surveyed component-based software technologies, described their advantages and disadvantages, and discussed the features they inherit. They also addressed quality assurance (QA) issues for component-based software systems. They proposed a quality assurance (QA) model for component-based software development which covers component requirement analysis, component development, component certification, component customization, and system architecture design, integration, testing, and maintenance. Remco C. de Boer [27] identified the main characteristics an architectural

TABLE IV
RQ3: SOFTWARE-BLOAT REDUCTION.

| App | # Required System Modules | JRE Size with All Modules (MB) | JRE Size with the Required Modules | Runtime Reduction (%) |
|---|---|---|---|---|
| University system | 2 | 236 | 36.1 | 84.7 |
| Nutch release 2.3 | 4 | 236 | 86.7 | 63.3 |
| Openjpa 2.4.1 | 10 | 236 | 96.1 | 59.3 |
| CXF-3.1.6 | 12 | 236 | 95.9 | 59.4 |
| Camel-2.17.0 | 8 | 236 | 90.2 | 61.8 |
| Lucene-4.6.1 | 7 | 236 | 89.4 | 62.1 |
| Average | 7 | 236 | 82.4 | 65.1 |

TABLE V
RQ4: PERFORMANCE OF *OO2CB*.

| App | CB App | | | | Performance (Min) |
|---|---|---|---|---|---|
| | Comps | Ports | ICC | JPMS Sys. Comm | |
| University system | 5 | 23 | 6 | 7 | 0.01 |
| Nutch release 2.3 | 57 | 257 | 93 | 69 | 0.02 |
| Openjpa 2.4.1 | 95 | 1358 | 426 | 101 | 0.13 |
| CXF-3.1.6 | 325 | 9319 | 3609 | 853 | 1.73 |
| Camel-2.17.0 | 67 | 1490 | 496 | 138 | 14.56 |
| Lucene-4.6.1 | 41 | 1490 | 397 | 95 | 0.18 |
| Average | 98.3 | 2,322.8 | 837.8 | 210.5 | 2.8 |

knowledge discovery method should exhibit. They concluded that Latent Semantic Analysis (LSA) is a promising technique for architectural knowledge discovery. LSA statistically infers the meaning of words in a context. To that end, LSA discovers the semantic structure latent in a collection of documents. This semantic structure can be employed to satisfy the requirements for successful architectural knowledge discovery. However, the distributional model of LSA is not an efficient representation compared with the recent deep neural network approaches. In addition, LSA representation is dense and hard to index based on individual dimensions. The aforementioned research works did not convert OO apps to CB apps, instead they focused on showing the benefits of component-based development which, among other research works, inspired us to develop *OO2CB*.

Kim and Chang [7] proposed a systematic UML-based method to generate component-based architecture of an application. To identify components, they used various techniques including clustering algorithms, metrics, decision rules and a set of heuristics. Their approach provides detailed instructions on how to design components. Using the proposed method, they showed that high quality components can be identified and designed given that an up-to-date and accurate UML design of a targeted system is available. Unfortunately, such an assumption is not valid in most software systems. In fact, most of the time, there is no model or the model is obsolete. Similarly, in their later work [28], they built an L2CBD (Legacy to Component Based Development) method to convert legacy systems to components-based systems. They relied on a specific re-engineering process that was manual. Their approach instructs developers with fine-grained steps to convert their legacy systems to CB systems. These approaches are hard to apply in practice due to the extensive manual effort. They are different from our work, which aims to provide a tool to assist the developers with the componentization process.

Gholam et al. [13] proposed a clustering-based technique to automatically generate a CB system. They identify components based on features explicitly stated in the use case diagrams of the system and an expert to choose the best results. Bertolino and Mirandol [8] proposed an approach called CB-SPE for specifying and analyzing performance-related properties in component-based systems. The proposed approach is considered a starting point towards an engineering approach to encompass performance validation in component-based systems on the basis of the architectural specification. Lee et al. [9] proposed an object-oriented component development methodology (COMO). They used COMO to develop software components. COMO extends the Unified Modeling Language (UML) and Rationales Unified process with semantics related to component development. COMO extends the UML notations by adding message flows between components and the ability to define classes within a component. However, it only considers data dependency among classes and components. Hamza et al. [11] proposed a framework that used the theory of Formal Concept Analysis (FCA) and the concepts of Software Stability Model (SSM) to build a component-based architecture of a system. The framework requires use-case diagrams and class diagrams to identify components. The framework was evaluated on a simple case study. Similarly, Mishra et al. [29] proposed an approach called CORE to convert legacy OO systems to CB systems using reverse engineering techniques depending on their UML diagrams. Unlike our approach, these research efforts depend on up-to-date UML-based diagrams to build the component-based architecture.

Jain et al. [10] showed that component-based software development is carried out in two phases: component building and application assembly. The approach leveraged a clustering algorithm named CompMaker, certain constraints, a predefined rule and a set of heuristics to identify business components in a system. The proposed approach uses static and dynamic relationship between classes to group related classes into components. Manolios et al. [30] tried to solve the system assembly problem directly from system requirements. Their framework includes an expressive language for declaratively describing system-level requirements, including component interfaces and dependencies, resource requirements, safety properties, objective functions, and various types of constraints. Moreover, Barrett R. Bryant et al. [31] presented an approach towards automatic component assembly based on aspect-oriented generative domain modeling. Aspect-oriented techniques are applied to capture the crosscutting concerns that emerge during the assembly process. Subsequently, those concerns are woven to generate glue/wrapper code for assembling heterogeneous components to construct a single integrated system. Unlike our work, these research efforts require up-to-date system requirements and they do not convert OO apps to CB apps.

Other research effort have focused on testing of component-based software systems as in [32], [33], [33]. Wu et al. [32] analyzed different test elements that are critical to test component-based software systems. They proposed a group of UML-based test adequacy criteria that can be used to test CB systems. Similarly, Harrold et al. [33] described issues and challenges in applying analysis and testing techniques to CB systems and presented an approach for analyzing and testing CB systems. Although such research work has different motivation than our work, their tools can be used by software engineers to test the generated CB applications by our approach.

Recently, Ghorbani et al. [4] proposed a technique, named DARCY, to detect architectural inconsistencies in Java 9 apps. While *OO2CB* aims to convert a Java 8 app to a Java 9 app that properly uses the JPMS constructs, DARCY takes a Java 9 app as input and checks if the app has architectural inconsistencies, and if so, attempts to fix them. Finally, the work of Hammad et al. in [34] and [35] determines and enforces the least-privilege architecture of an Android system at runtime. However, their works are specific to Android apps and cannot be applied to Java OO apps.

## VIII. Conclusion

This paper presents *OO2CB*, a system for determining the least-privilege architecture of an OO Java app and its conversion to a CB Java app. The determined least-privilege architecture reduces the software-bloat and narrows the attack surface of a component-based software system in Java. Our experimental evaluation on several real-world object-oriented Java apps show the effectiveness of our approach in successfully converting OO apps to CB apps. Notably, through careful analysis of the dependencies and communication requirements, *OO2CB* achieved an average of 87.2% reduction in inter-component communications, 97.9% reduction in system communications, and 91.4% reduction in the number of required JRE system components.

Since some security attacks take advantage of the Java reflection and dynamic class loading features to implement their malicious intent, in our future work we intend to extend support to additional JPMS ports, namely `open` and `opens with`, which would mitigate such threats. Furthermore, as part of our future work, we will conduct user studies with developers to evaluate the degree to which *OO2CB* can aid developers in componentization of their Java apps.

Our research artifacts, including tools and evaluation data, are available publicly [24].

## IX. Acknowledgments

This research is partially funded by Jordan University of Science and Technology. Research Grant Number: 301-2019

## References

[1] R. N. Taylor, N. Medvidovic, E. M. Dashofy, Software architecture: foundations, theory, and practice.(2009), Google Scholar Google Scholar Digital Library Digital Library.

[2] D. M. Le, D. Link, A. Shahbazian, N. Medvidovic, An empirical study of architectural decay in open-source software, in: 2018 IEEE International conference on software architecture (ICSA), IEEE, 2018, pp. 176–17609.

[3] Jpms, http://openjdk.java.net/projects/jigsaw/spec/.

[4] N. Ghorbani, J. Garcia, S. Malek, Detection and repair of architectural inconsistencies in java, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 560–571. `doi:10.1109/ICSE.2019.00067`.

[5] OpenJDK: Jigsaw Project, https://openjdk.java.net/projects/jigsaw/.

[6] jlink tool, https://docs.oracle.com/javase/9/tools/jlink.htm#JSWOR-GUID-CECAC52B-CFEE-46CB-8166-F17A8E9280E9.

[7] S. D. Kim, S. H. Chang, A systematic method to identify software components, in: 11th Asia-Pacific software engineering conference, IEEE, 2004, pp. 538–545.

[8] A. Bertolino, R. Mirandola, Modeling and analysis of non-functional properties in component-based systems, Electronic Notes in Theoretical Computer Science 82 (6) (2003) 158–168.

[9] S. D. Lee, Y. J. Yang, F. S. Cho, S. D. Kim, S. Y. Rhew, Como: A uml-based component development methodology, in: Proceedings Sixth Asia Pacific Software Engineering Conference (ASPEC'99)(Cat. No. PR00509), IEEE, 1999, pp. 54–61.

[10] H. Jain, N. Chalimeda, N. Ivaturi, B. Reddy, Business component identification-a formal approach, in: Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference, IEEE, 2001, pp. 183–187.

[11] H. S. Hamza, A framework for identifying reusable software components using formal concept analysis, in: 2009 Sixth International Conference on Information Technology: New Generations, IEEE, 2009, pp. 813–818.

[12] M.-S. Choi, E.-S. Cho, A component identification technique from object-oriented model, in: International Conference on Computational Science and Its Applications, Springer, 2005, pp. 778–787.

[13] G. Shahmohammadi, S. Jalili, S. M. H. Hasheminejad, Identification of system software components using clustering approach., Journal of Object Technology 9 (6) (2010) 77–98.

[14] J. Garcia, I. Ivkovic, N. Medvidovic, A comparative analysis of software architecture recovery techniques, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2013, pp. 486–496.

[15] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, Y. Cai, Enhancing architectural recovery using concerns, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), IEEE, 2011, pp. 552–555.

[16] B. S. Mitchell, S. Mancoridis, On the automatic modularization of software systems using the bunch tool, IEEE Transactions on Software Engineering 32 (3) (2006) 193–208.

[17] P. Andritsos, V. Tzerpos, Information-theoretic software clustering, IEEE Transactions on Software Engineering 31 (2) (2005) 150–165.

[18] V. Tzerpos, R. C. Holt, Acdc: an algorithm for comprehension-driven clustering, in: Proceedings Seventh Working Conference on Reverse Engineering, IEEE, 2000, pp. 258–267.

[19] A. Corazza, S. Di Martino, V. Maggio, G. Scanniello, Investigating the use of lexical information for software system clustering, in: 2011 15th European Conference on Software Maintenance and Reengineering, IEEE, 2011, pp. 35–44.

[20] O. Maqbool, H. Babri, Hierarchical clustering for software architecture recovery, IEEE Transactions on Software Engineering 33 (11) (2007) 759–780.

[21] A. Shatnawi, A. Seriai, H. Sahraoui, Z. Al-Shara, Mining software components from object-oriented apis, in: International conference on software reuse, Springer, 2015, pp. 330–347.

[22] BCEL byte code engineering library, https://commons.apache.org/proper/commons-bcel/.

[23] Oracle bug database, https://bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8038838.

[24] OO2CB artifact and evaluation data, Honoring-the-double-blind-policy-of-ICSA-2021.

[25] J. Garcia, I. Krka, N. Medvidovic, C. Douglas, A framework for obtaining the ground-truth in architectural recovery, in: 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, IEEE, 2012, pp. 292–296.

[26] X. Cai, M. R. Lyu, K.-F. Wong, R. Ko, Component-based software engineering: technologies, development frameworks, and quality assurance schemes, in: Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000, IEEE, 2000, pp. 372–379.

[27] R. C. de Boer, Architectural knowledge discovery: why and how?, ACM SIGSOFT Software Engineering Notes 31 (5) (2006) 1.

[28] H.-K. Kim, Y.-K. Chung, Transforming a legacy system into components, in: International Conference on Computational Science and Its Applications, Springer, 2006, pp. 198–205.

[29] S. K. Mishra, D. S. Kushwaha, A. K. Misra, Creating reusable software component from object-oriented legacy system through reverse engineering., Journal of object technology 8 (5) (2009) 133–152.

[30] P. Manolios, D. Vroon, G. Subramanian, Automating component-based system assembly, in: Proceedings of the 2007 international symposium on Software testing and analysis, ACM, 2007, pp. 61–72.

[31] F. Cao, B. R. Bryant, C. C. Burt, R. R. Raje, A. M. Olson, M. Auguston, A component assembly approach based on aspect-oriented generative domain modeling, Electronic Notes in Theoretical Computer Science 114 (2005) 119–136.

[32] Y. Wu, M.-H. Chen, J. Offutt, Uml-based integration testing for component-based software, in: International Conference on COTS-Based Software Systems, Springer, 2003, pp. 251–260.

[33] M. J. Harrold, D. Liang, S. Sinha, An approach to analyzing and testing component-based systems, in: First International ICSE Workshop on Testing Distributed Component-Based Systems, Los Angeles, CA, 1999, pp. 333–347.

[34] M. Hammad, H. Bagheri, S. Malek, Determination and enforcement of least-privilege architecture in android, in: 2017 IEEE international conference on software architecture (ICSA), IEEE, 2017, pp. 59–68.

[35] M. Hammad, H. Bagheri, S. Malek, Deldroid: an automated approach for determination and enforcement of least-privilege architecture in android, Journal of Systems and Software 149 (2019) 83–100.