Object-Oriented Programming (OOP) has become the dominant programming paradigm during the last few decades. It introduced the idea of using objects to represent different components of a given system by breaking down a problem into separate objects, and having each object grouping together data and behaviors into a single entity. Such an approach aids in writing complex applications while maintaining comprehensible source code [77]. However, some requirements do not decompose efficiently into a single entity, and thus scatter in various places in the application source code. To this end, Aspect-Oriented Programming is introduced to solve this issue and separately allows for the specification of the different concerns of a system [77].

Aspect-Oriented Programming (AOP) [114] is based on the idea of separation of cross-cutting concerns. In other words, it separately specifies the different concerns that cross-cut the application source code in many places, and then defines a mechanism, called weaving, to compose the different parts into a coherent program. These concerns may vary depending on the application domain; they can be functional or non-functional, they may be high-level or low-level features. The objective of aspect-orientation is to realize these scattered concerns into single elements, called aspects, and eject them from the various locations of the program [77]. AOP techniques have emerged into various families of programming languages. They can be defined over different languages, such as C, C++, PHP, and Java.

Many approaches were proposed in the literature to achieve the goals of AOP, such as Pointcut-Advice [131], Multi-Dimensional Separation of Concerns [159], and Adaptive Programming [158] models. According to the study conducted in [33], the pointcut-advice model is the most appropriate one for security hardening.

The remainder of this chapter is organized as follows. In Sect. 3.1, we present an overview of the main AOP models. In Sect. 3.2, we discuss the appropriateness of these AOP models from a security perspective. Section 3.3 presents the main constructs of the pointcut-advice model. Section 3.4 introduces the main concepts of AOM. Finally, Sect. 3.5 concludes this chapter.
3.1 AOP Models

Various AOP models have been proposed to achieve the goals of AOP. The most important ones are: Pointcut-Advice [131], Multi-Dimensional Separation of Concerns [159], and Adaptive Programming [158] models. In the following, an overview of each model is presented.

3.1.1 Pointcut-Advice Model

The fundamental concepts of the pointcut-advice model are: join points, pointcuts, and advices. A join point is an event during the execution of a program such as a method call or a method execution. A pointcut is an expression that designates a set of join points. An advice is a piece of code specifying how the behavior of an application should be adapted at specific points. Advice code can be executed before, after, or around a specific join point. Before-advice and after-advice are executed before and after the intercepted join point, respectively. Whereas, around-advice executes in place of the intercepted join point. Moreover, the computation of the original join point can be executed within the body of the around-advice using a special construct named `proceed`. AspectJ [113] is the most known representative of the pointcut-advice model. Figure 3.1 shows a tracing aspect written in AspectJ where the pointcut `ptrace` picks out any call to any method. Before-advice and after-advice are used to display the start time and the end time respectively.

Around-advice must be declared with a return type, like a method because it is allowed to return a value. Within the body of around-advice, the computation of the original join point can be executed with the special syntax `proceed(...)`. The `proceed` form takes as arguments the context exposed by the pointcut of the around-advice, and returns whatever the around-advice is declared to return. Accordingly, the around-advice, shown in Fig. 3.2, doubles the second argument to `foo` whenever it is called, and then halves its result.

```java
public aspect TracingAspect {
    pointcut ptrace(): call (* *(..));
    before(): ptrace()
    { System.out.println("Start Time: "+System.currentTimeMillis()); }
    after(): ptrace()
    { System.out.println("End Time: "+System.currentTimeMillis()); }
}
```

Fig. 3.1 AspectJ tracing example