An Approach to Analyze Vulnerability of Information Flow in Software Architecture

Tingyang Gu 1,2, Minyan Lu 1,2, Luyi Li 3 and Qiuying Li 1,2,*

1 The Key Laboratory on Reliability and Environmental Engineering Technology, Beihang University, Beijing 100191, China; gutingyang@126.com (T.G.); lmy@buaa.edu.cn (M.L.)
2 School of Reliability and Systems Engineering, Beihang University, Beijing 100191, China
3 China Academy of Electronics and Information Technology, China Electronics Technology Group Corporation, Beijing 100041, China; liluyinal1@126.com
* Correspondence: li_qiuying@buaa.edu.cn

Received: 10 December 2019; Accepted: 2 January 2020; Published: 4 January 2020

Abstract: Current research on software vulnerability analysis mostly focus on source codes or executable programs. But these methods can only be applied after software is completely developed when source codes are available. This may lead to high costs and tremendous difficulties in software revision. On the other hand, as an important product of software design phase, architecture can depict not only the static structure of software, but also the information flow due to interaction of components. Architecture is crucial in determining the quality of software. As a result, by locating the architecture-level information flow that violates security policies, vulnerabilities can be found and fixed in the early phase of software development cycle when revision is easier with lower cost. In this paper, an approach for analyzing information flow vulnerability in software architecture is proposed. First, the concept of information flow vulnerability in software architecture is elaborated. Corresponding security policies are proposed. Then, a method for constructing service invocation diagrams based on graph theory is proposed, which can depict information flow in software architecture. Moreover, an algorithm for vulnerability determination is designed to locate architecture-level vulnerabilities. Finally, a case study is provided, which verifies the effectiveness and feasibility of the proposed methods.

Keywords: software architecture; vulnerability; information flow vulnerability of software architecture; locating vulnerability; service invocation diagram; algorithm for determining vulnerability

1. Introduction

Due to the increasing complexity of software systems, designers tend to focus on the design and implementation of functional requirements. Non-functional requirements are often neglected, such as security [1]. So, vulnerabilities are inevitably introduced in design phase. In essence, software vulnerabilities are a special kind of defects. They are the manifestations of errors that violate security policies, which exist in every phase of the software life cycle, including requirement analysis, design, coding, testing and operation phases. Such vulnerabilities are the root causes of software security problems [2]. If these vulnerabilities cannot be discovered in time and reside in software until later phases of software development cycle, the normal operation of software systems will be faced with great threats. Software vulnerability analysis is the general term for techniques of locating vulnerabilities, and analyzing their production mechanism and action modes [3–5]. It is an important research area in cyber security.

However, although current approaches on software vulnerability analysis can be used for identifying software vulnerabilities, they are mostly targeted at source code and executable programs.
These methods can only be applied after the coding phase, when source code and executable programs are available. According to statistical data, about 50%–75% of defects are introduced during the design phase [6], and the repair cost will increase with the delay of discovery time of these defects. As a result, current methods on software vulnerability analysis targeted at source code and executable programs have the following shortcomings: (1) delay of time in discovering vulnerabilities; (2) difficulty fixing these vulnerabilities; (3) high cost in revision.

As the major product of the design phase, software architecture is composed of components and interactions between components [7]. Software architecture can be used to depict not only the static structure of the software, but also the information flow and propagation resulting from component interactions. It is a high-level abstraction of a software system [8]. The architecture defines functional, allocated, and product baselines of a system [9]. It can also be used to track current and future descriptions of a system composed of components and their interconnections, the actions or activities those components perform, and the rules or constraints for those activities [10]. Software architecture plays a crucial role in determining software quality [11].

Analyzing and evaluating software architecture during the design phase has been proved to be an effective way to find potential problems in the early stages of software life cycle, reduce costs and assure software quality [12]. Guided by this idea, some researchers have conducted research on architecture-level security analysis and design [13–19]. These methods can be used for verifying whether the design of software architecture has met the security requirements, or assuring the security of software architecture by designing security policies that constrain the components, connectors and configurations. However, none of these methods can be used for directly locating vulnerabilities in software architecture.

As a result, by conducting vulnerability analysis for software architectures that only focus on functional requirements but neglect non-functional requirements, and locating the information flow and propagation that violates security policies at design phase, can help find and fix these problems and prevent these vulnerabilities from residing until later phases. Locating the architecture-level information flow and propagation that violate security policies enables us to fix these vulnerabilities as early as possible, avoiding the possibility for them to reside until later phases of software development cycle. Some researchers have proposed that it is necessary to analyze software vulnerability on the architecture level [20,21], but what they propose is only the belief and necessity for architecture-level vulnerability analysis, lacking the description of the meaning of software architecture vulnerability as well as feasible or practical approaches.

As a result, this paper provides a graph-theory-based approach to analyze information flow vulnerabilities in software architecture. The key contribution of this paper is to provide an architecture-level vulnerability analysis method by examining the information flow between components. The proposed method can be used for directly locating architecture-level vulnerabilities. In detail, first, the concept of information vulnerability in software architecture is proposed. This concept complements and improves current theory of software vulnerability in design phase. Second, based on Bell–LaPadula model and Biba model, the policies for determining information flow vulnerability in software architecture are proposed. The proposed policies are extensions of Bell–LaPadula model and Biba model to architecture level. These policies constrain the direction of information flow. Third, a graph-theory based modeling method is proposed. The constructed service invocation diagram model can be used to depict the information flow and propagation in architecture, with consideration of security levels of each node and operation types between nodes. Fourth, an algorithm for locating information flow vulnerabilities in software architecture is proposed.

The rest of the paper is organized as follows: In Section 2, related works are analyzed. The proposed method is elaborated in Section 3. Section 4 presents a case study. Conclusions and future work are given in Section 5.
2. Related Work

2.1. Software Vulnerability

In this paper, we will research the definition of information flow vulnerability in software architecture. So, we will firstly review the research on definitions of software vulnerability.

Research on the definitions of software vulnerability can help researchers better understand the essence of software vulnerability. As a result, researchers have proposed various definitions and elaborations for software vulnerability from different perspectives.

Current definitions of software vulnerability can be classified into four categories: definitions based on access control, definitions based on state space, fuzzy definitions, and definitions based on security policies, as shown in Table 1.

<table>
<thead>
<tr>
<th>Categories of Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access control [22]</td>
<td>An access control vulnerability is whatever causes the operating system to perform operations that are in conflict with the security policy as defined by the access control matrix.</td>
</tr>
<tr>
<td>State space [23]</td>
<td>A computer system is viewed as the composition of configuration states of entities. System operation is achieved through transitions between states. State transitions are classified as authorized and unauthorized. A vulnerable state refers to an authorized state that can transitions into an unauthorized state through authorized transition.</td>
</tr>
<tr>
<td>Fuzzy [24]</td>
<td>Software vulnerabilities are weak points that can be exploited by threats and cause potential losses.</td>
</tr>
<tr>
<td>Security policies [2]</td>
<td>A software vulnerability is an instance of an error in the specification, development, or configuration of software such that its execution can violate the security policy.</td>
</tr>
</tbody>
</table>

Among the four kinds of definitions of software vulnerability, fuzzy definition is widely used in books, dictionaries, standards and cyber security literature. But this definition is too general and abstract. It lacks the clear elaboration of the essence of software vulnerability.

Compared with fuzzy definition, the definitions based on access control and state space are clearer. But access control definition can only be used for describing the vulnerabilities that violate access control models. State space definition on the other hand is targeted at only UNIX and Windows NT, not suitable for other software.

By comparison, the definition of software vulnerability based on security policies clearly states that software vulnerabilities exist in every phase of software development cycle, and describes the relationship between software vulnerabilities and security policies that specifies what is illegal, what can happen, and what is allowed in a software system. It emphasizes that in essence software vulnerabilities are errors that violate security policies. However, this definition still does not associate with specific features of different phases in software development lifecycle. Thus, in this paper, on the basis of the definition of software vulnerability based on security policies, and with a focus on the information flow and propagation on architecture level, we will research into the definition of information flow vulnerability in software architecture.

2.2. Vulnerability Analysis on Architecture Level

Currently, research on architecture-level vulnerability analysis is still in starting phase [20]. As comprehensively as possible, we collected three papers related to this.

In [25], the authors proposed a method and a corresponding tool called SecArch to evaluate security-critical architectures. SecArch is an incremental evaluation tool for secure architectures. It employs multiple architecture diagrams and scenarios to locate unanticipated interaction patterns in the architecture, to evaluate its security. However, since they did not clarify the meaning of software architecture vulnerability, they did not clearly state in the conclusion that it is in the modules containing abnormal interaction patterns that vulnerabilities exist. In comparison with their work, we focus on the
information flow and propagation resulting from interactions between components. The models we employ are not existing architecture diagrams, but a graph-theory-based model that can explicitly depict the information flow in software architecture. Moreover, we will clarify the definition of information flow vulnerability in software architecture, and propose policies for determining information flow vulnerabilities in software architecture.

Another study [26] utilized the features of software architecture to find backdoors of the software. They analyzed the relationship between architecture, dynamic behavior, and security vulnerabilities to identify potential backdoors in architecture. Their main idea was to compare the actual architecture that was obtained after the system is completely developed, with the architecture in the original version and discovered vulnerabilities by injecting malicious code. Hence, their method cannot be applied at design phase to locate architectural vulnerabilities. By contrast, the method proposed in this paper is targeted at locating architectural vulnerabilities at design phase. Comparison between multiple versions of software architectures is not needed.

In [27], the authors proposed the concept of software vulnerability degree. In their definition, the less stable the operation is, the more vulnerable the software is. They built mathematical models of software structural vulnerability, constructed a transition equation set to describe the operation process of software elements, and evaluated the vulnerability of software structure by use of the metric of defect density in source code. Also, they give a control method for software vulnerability degree. However, they did not propose the method for locating structural vulnerabilities. By comparison, our definition for software vulnerability is different from theirs. The object we employ in case study is not source code, but an architecture. Most importantly, we propose a method for locating architectural vulnerabilities.

2.3. Static Vulnerability Analysis of Source Code and Executable Programs

Because vulnerability analysis on architecture level belongs to the research domain of software vulnerability analysis, and the literature of architecture-level vulnerability analysis is scarce, we extend the literature view to software vulnerability analysis. As mentioned in the Introduction section, current research on software vulnerability analysis is mostly targeted at source code and executable programs and the methods they utilize are static analysis and dynamic testing. Since the focus of this paper is architecture-level vulnerability, it is difficult to directly execute a software architecture [28]. As a result, in this subsection, we will review the research on static vulnerability analysis of source code and executable programs. By reviewing related work on static vulnerability analysis, we hope to borrow some ideas for our research.

Static analysis refers to analyzing source code or binary code without running the program [5]. Static analysis approaches include data flow analysis, symbolic execution, theorem proving, and so on. Dataflow analysis is the most widely used static analysis method. This method examines the definitions and values of variables in a program to analyze potential vulnerabilities [29,30]. When conducting dataflow analysis, source code is firstly abstracted into a syntax tree or control flow diagram. Then algebraic methods are used to calculate the use of variables, with which the runtime behavior of a program can be depicted. Finally, corresponding rules are used to find vulnerabilities in source code [31]. Symbolic execution [32–34] models source code into control flow diagrams or function invocation graphs. Then certain symbols are used to represent the values of variables to simulate program execution. Finally, vulnerabilities in source code are discovered by the use of vulnerability examination rules. Although symbolic execution is an effective approach to finding vulnerabilities in source code, in actual executions state explosion may occur. Besides, the analysis results rely heavily on the problem-solving capability of automatic tools [4]. Theorem proving [35,36] transforms source code into logical equations and theorems or rules are employed to demonstrate whether the program is a valid theorem in order to find the vulnerabilities in the source code. Compared with other static analysis methods, this method is more accurate, but needs more human intervention. As a result, it is not suitable for complex software.
By analyzing related research on static vulnerability analysis and the work in [5], we can summarize the basic steps of static vulnerability analysis as follows:

Step 1: model construction. In this step, source code or an executable program is modelling into a model that can be used for further analysis.

Step 2: pattern extraction. In this step, typical vulnerability patterns are extracted based on historical vulnerabilities.

Step 3: pattern matching. In this step, the model from step 1 and vulnerability patterns from step 2 are matched. Corresponding analysis methods are selected to obtain vulnerability information from the source code or executable programs.

The method proposed in this paper will be based on the above solution from static vulnerability analysis, including model construction, pattern extraction and pattern matching. However, by comparison, the modeling target of this paper is not source code, but software architecture. As for pattern extraction, there is no architecture-level vulnerability database for us to conduct pattern extraction. So, we will borrow ideas from information flow control theory. In later subsections, we will review studies on architecture analysis methods based on graph theory, and information control theory.

2.4. Architecture Analysis Based on Graph Theory

Graphs are a fundamental construct in complex systems research, and the use of graph theoretic algorithms and metrics to extract useful information from a problem is a primary method of analysis for complex systems [37]. A graph-based representation is used to provide a particular perspective on a problem. Generally, graph-theoretic representations of a problem emphasize aspects that are either structural such the connection of components or processes in the problem, or that depend on structural properties such as the robustness of a network of components. Graph representations are particularly useful in understanding systems involving multiple interacting entities. Such complex systems lend themselves to a graph-theoretic representation and the rich body of network flow methods that can represent and analyze the component structures and patterns of interactions at both local and global levels [38,39].

Architecture analysis methods based on graph theory have been widely adopted by researchers from systems engineering and software engineering area. In systems engineering area, graph theory provides a structural perspective for systems, with a focus on structural properties such as robustness, resilience, and security [40–42]. In software engineering area, researchers usually utilize methods based on graph theory to model the interactions between web services or classes [43–47]. Graph-based modeling can be classified into two categories: The first category focuses on topological structure, modeling the static interdependent structure of components, such as classes and services. The second category focuses on the dynamic interaction, modeling the data flow, interactions between services, invocation relationship between classes in software architecture. To improve the modeling capability of these graph-based models, researchers usually extend the nodes and edges in the graph to satisfy the needs of their respective research. This paper will be based on the second category of graph-based models.

2.5. Information Flow Control

Information flow is the information propagation and flow inside a system and across systems [48]. Information flow control is a significant means in cyber security area to ensure end-to-end information security. Information flow control employs security policies or security models to control the flow of information [49]. In this paper, we will use the thinking of information flow control to analyze the information flow vulnerabilities in software architecture.

First of all, we will clarify the concepts and terms used in information flow security, including security goals, security policies, security models, and lattice model of information flow.

Security goals of information flow include confidentiality goals and integrity goals. Confidentiality is the property that information is not made available or disclosed to unauthorized individuals, entities,
or processes. Integrity is the property that information cannot be modified in an unauthorized or undetected manner, and the accuracy and completeness of information is maintained and assured over its entire lifecycle.

Security policies of information flow are a set of policies designed for a group of security goals, which define what is allowed and what is prohibited. Similar to security goals, security policies include confidentiality policies and integrity policies. Confidentiality policies prevent information from flowing to entities that are not authorized to access the information. Integrity policies prevent information from flowing to entities with higher integrity [50].

Security policies are usually represented by security models. Security models are representations of security policies by use of formal or mathematical terminology. John Mclean made a detailed summary of security models in [51]. In this paper, we will make use of two important information flow security models, which are Bell–LaPadula model and Biba model. The Bell–LaPadula model is suitable for military security policies. It ensures system confidentiality by preventing the diffusion of unauthorized information [52]. The basic principle of the Bell–LaPadula model is that a subject with a higher security level is not allowed to write an object with a lower security level, and a subject with a lower security level is not allowed to read an object with a higher security level, which can be characterized as “no read up, no write down.” The Biba model [53] is used to protect integrity. Its basic principle is that a subject with a higher integrity level can write an object with a lower integrity level, and a subject with a lower integrity level can read an object with a higher integrity level, which can be characterized as “no write up, no read down”.

In various formal or mathematical descriptions of information flow security models, the most widely used model is the lattice model of information flow proposed by Denning. The information lattice model is defined as follows [48]:

If \((L, \leq)\) is a semi-order set and there is a maximum lower bound and a minimum upper bound for every element \((a, b)\) in \(L\), then the tuple \((L, \leq)\) is lattice.

This model is used for classifying information into different security groups. Each group of information forms a security class. Information is only allowed to flow between specified security classes. According to confidentiality policies, information is only allowed to flow inside a security class (class A) or into a security class (class B) with a higher security level. According to integrity policies, information is only allowed to flow inside a security class or into a security class with a lower security level.

In later sections, we will use the lattice model to depict the information flow security in software architecture.

3. Proposed Approach

3.1. Assumptions and Framework of the Proposed Method

**Assumption 1.** The understanding of software architecture in this paper is based on [7]. Software architecture is composed of components and interactions between components. In modelling software architecture, we borrow the concepts from service-oriented architecture (SOA), which is widely used in both software engineering and systems engineering. In SOA, each service is an independent function unit composed of a single component or several components [54]. So, the software architecture in this paper is viewed as a composition of services and interactions between services.

**Assumption 2.** As mentioned in the Introduction section, designers tend to focus on the design and implementation of functional requirements, while non-functional requirements tend to be neglected [1]. Hence, the second assumption is that the architectures that this paper focuses on are designed without considering non-functional requirements, especially security requirements. So, these architectures only reflect functional-level information flow and propagation.
Assumption 3. The types of service invocation operation are limited to reading and writing. While in software engineering, there are three types of operations—Reading, writing, and execution. The execution operation is not considered in this paper, because execution operations occur in source code level or executable program level [55]. Execution operation is not suitable for architecture level.

In this paper, we first propose a definition and formal description of information flow vulnerability in software architecture. Then, security policies of information flow in architecture are given based on Bell–LaPadula model and Biba model. After that, the construction method for service invocation diagram of software architecture is proposed based on graph theory. Finally, we designed and implemented a corresponding algorithm for locating vulnerabilities. The framework of the research method is shown in Figure 1, where green blocks represent background and domain knowledge needed in this research, rounded rectangles represent research that needed to be conducted, blue rectangles represent the results and outputs of each part of the research, gray rectangles represent the analysis results.

Figure 1. Framework of the proposed approach.

3.2. Vulnerability of Information Flow in Software Architecture

In this subsection, we will propose the concept of vulnerability of information flow in software architecture and corresponding formal description. As mentioned above, software architecture can be used for depicting the information flow and propagation resulting from interactions between components. As a result, before giving the definition of information flow vulnerability in software architecture, we will firstly clarify two concepts, including what is software architecture vulnerability, what is the information flow in architecture.

Definition 1. Software architecture vulnerability. Software architecture vulnerability is the errors resulting from violations of corresponding security policies by components and interaction between components that forms the software architecture.

Based on this definition, we can further explain and understand the essence of software architecture vulnerability from four aspects. First, the essence of vulnerability is a kind of design errors. Second, the core of software architecture vulnerability is the violation of security policies by architecture
design. Third, this definition can help architecture designers and analyzers identify which part of the architecture needs to be protected by an architecture-level security mechanism [2]. Fourth, this definition is a generalization of all kinds of software architecture vulnerabilities. It can be further divided into subcategories, which will not be covered in this paper.

To better differentiate our work from other current work on software vulnerability, we compare software architecture vulnerability with traditional code-level vulnerability, shown in Table 2.

### Table 2. Comparison of software architecture vulnerability with code-level vulnerability.

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Point of View</th>
<th>Architecture-Level Vulnerability</th>
<th>Code-Level Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Difference</td>
<td>Abstraction granularity</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Object to be analyzed</td>
<td>Software architecture</td>
<td>Source code or binary program</td>
<td></td>
</tr>
<tr>
<td>Focus</td>
<td>Components, interactions between components, topology resulting from interactions</td>
<td>Lexical, syntactic, semantic</td>
<td></td>
</tr>
<tr>
<td>Similarity</td>
<td>Manifestation form</td>
<td>Flow and propagation of data, transition of states, passing of control power</td>
<td>Data flow, control flow, state space</td>
</tr>
</tbody>
</table>

The most significant differences between software architecture and code-level vulnerability are as follows. First, they are of different abstraction granularities. Architecture-level vulnerabilities are at a higher level of abstraction granularity. Besides, objects to be analyzed in architecture-level and code-level vulnerability are different. In architecture-level vulnerability, the software architecture is analyzed. Finally, they have different focuses. Architecture vulnerability analysis focuses on the structural level, such as components, interactions between components and the topology resulting from these interactions while code-level vulnerability analysis focuses on lexical, syntactic and semantic aspects.

**Definition 2.** Information flow in architecture. This is the flow and propagation of information of the components, interactions between components and the topology that forms from their interaction. The information in this definition is the same as data.

Based on the understanding of information flow in architecture above, vulnerability of information flow in software architecture is defined as follows:

**Definition 3.** Vulnerability of information flow in software architecture. Vulnerability of information flow in software architecture is errors resulting from the violation of corresponding security policies during the flow and propagation of information in components, interactions between components and the topology formed from those interactions.

Normally, the flow and propagation of information should conform to confidentiality and integrity policies [49]. Hence, the manifestation of information flow vulnerability in software architecture can be categorized into vulnerability under confidentiality and integrity policies.

**Definition 4.** Vulnerability of information flow in software architecture under confidentiality policies. Information carried by the architecture only being allowed to flow inside services at the same security level or from services of low security level to those of high security level. All services and interactions between services violating this principle are vulnerabilities of information flow in architecture under a confidentiality policy.

This definition can be described based on a lattice as follows:

For software architecture SA, there are confidentiality policy requirements. When \([L, \leq]\), \(A, B \in L\), if there exists passing of information from A to B, then \(A \leq B\) holds;
If there exists \( a \in A, b \in B, \) and \( a \rightarrow b \), then \( a \rightarrow b \) is secure;
If there exists \( a \in B, b \in A, \) and \( a \rightarrow b \), then \( [a \rightarrow b] \in V_{sa} \).

**Definition 5.** Vulnerability of information flow in software architecture under integrity policies

Information carried by the architecture is only being allowed to flow inside services of the same security level or from services of high security level to those of low security level. All services and interactions between services violating this principle are vulnerabilities of information flow in the architecture under integrity policy.

This definition can be described based on lattice as follows:
For software architecture \( SA \), there are integrity policies requirement. When \( \{ L, \geq \} \) and \( A, B \in L \), if there exists data passing from \( A \) to \( B \), then \( A \geq B \) holds;
If there exists \( a \in A, b \in B, \) and \( a \rightarrow b \), then \( a \rightarrow b \) is secure;
If there exists \( a \in B, b \in A, \) and \( a \rightarrow b \), then \( [a \rightarrow b] \in V_{sa} \).

In the description above, \( SA \) represents the software architecture set, \( L \) represents the set of secure classes, \( \geq \) denotes the relationship between low and high secure classes, \( A \) and \( B \) are elements in \( L \), \( a \) represent a service, \( a \rightarrow b \) means \( a \) is of security level \( A \rightarrow \) denotes the direction of information flow between services, and \( V_{sa} \) represents software architecture vulnerability.

To explain Definitions 4 and 5, Figures 2 and 3 are presented. In Figure 2, we can see that the architecture is composed of service 1 to service 5. The rectangular boxes represent services. The color represents the security level of each service, in which blue represents low security level while yellow represents high security level. The boxes with the same color represent services with the same security level. The symbol “\( \rightarrow \)” represents the interaction between services and the direction of data flow. In Figure 2, the part inside the red box is the vulnerability of information flow in software architecture under confidentiality policies.

**Figure 2.** Graphic example of Definition 4.

**Figure 3.** Graphic example of Definition 5.
Similarly, in Figure 3, we can see that the architecture is composed of service 1 to service 5. The rectangular boxes represent services. The color represents the security level of each service, in which pink represents high security level while green represents low security level. The boxes with the same color represent services with the same security level. The symbol “→” represents the interaction between services and the direction of data flow. In Figure 3, the part inside the red box is the vulnerability of information flow in software architecture under integrity policies.


In this subsection, we propose security policies of information flow in software architecture based on Bell–LaPadula model and Biba model. These policies specify the direction of information flow in three operations that occur in service interactions, including reading operation, writing operation as well as reading and writing operation.

3.3.1. Security Policies of Information Flow under the Goals of Confidentiality

As mentioned above, the confidentiality policy of information flow is intended to prevent information from flowing to subjects that are not authorized to access the information, that is, information is only allowed to flow inside a security class or from a class with low security level to a class with higher security level. We propose security polices as follows:

**Policy 1. Policy for reading operation in service invocation under the goal of confidentiality.**

If there is an invoking service A and an invoked service B, A can perform the reading operation on B if and only if $C_a \geq C_b$. The information flow is $B \rightarrow A$. $C_x$ stands for the confidentiality level of service $x$.

**Policy 2. Policy for writing operation in service invocation under the goals of confidentiality.**

If there is an invoking service A and an invoked service B, A can perform the writing operation on B if and only if $C_a \leq C_b$. The information flow is $A \rightarrow B$.

**Policy 3. Policy for reading and writing operation in service invocation under the goals of confidentiality.**

If there is an invoking service A and an invoked service B, A can perform both reading and writing operation on B if and only if $C_a = C_b$. The information flow is $A \leftrightarrow B$.

3.3.2. Security Policies of Information Flow under the Goals of Integrity

An integrity policy of information flow is intended to prevent information from flowing to subjects or data with a higher integrity level and ensures that information only flows inside a security class or to a class with lower security level. Integrity refers to the feature that information cannot be destroyed, lost, or modified without authorization when it is transmitted, exchanged, stored, or processed. There are two kinds of modification, direct and indirect. Direct modification means service A performs the writing operation on service B, and indirect modification means that service A performs the reading operations on service B. Aiming at reading, writing, as well as reading and writing operations that occur during service invocation, we propose security polices as follows:

**Policy 4. Policy for writing operation in service invocation under the goal of integrity.**

If there is an invoking service A and an invoked service B, A can perform the writing operation on B if and only if $I_a \geq I_b$. The information flow is $A \rightarrow B$. $I_x$ stands for the integrity level of service $x$.

**Policy 5. Policy for reading operation in service invocation under the goal of integrity.**

If there is an invoking service A and an invoked service B, A can perform the reading operation on B if and only if $I_a \leq I_b$. The information flow is $B \rightarrow A$.

**Policy 6. Policy for reading and writing operation in service invocation under the goal of integrity.**
If there is an invoking service A and an invoked service B, A can perform both reading and writing operations on B if and only if \( I_a = I_b \). The information flow is \( A \rightleftharpoons B \).

### 3.4. Construction Approach of Service Invocation Diagram Based on Software Architecture

From Section 3.3, we can see that two key points of security policies of information flow in architecture are the direction of information flow during service invocation and the security class of each service. As a result, the model constructed by the proposed approach should have corresponding descriptive ability. That is, the model should be able to reflect the direction and path of information flow in the architecture as well as the confidentiality and integrity level of the information carried by services. Software architecture design specification contains some diagrams that can reflect information interaction, such as data flow diagrams, flow charts, activity diagrams, and state diagrams, but these diagrams cannot be used for analysis directly. This is because the uncertainty of the future operation environment of the system, the complexity of the software, the cognitive limitation of the designer and service reuse in current service-oriented architectures make it impossible for all potential information flow and propagation to be included in these diagrams. To further explain this problem, we give an example, shown in Figure 4.

![Data flow path](image)

**Figure 4.** Data flow path of (a) architecture design phase vs. (b) data flow path in actual operation.

In the architecture design specification of a piece of software, there is a statement that when performing a certain function, a calculation service X is supposed to pass the calculated value to another service Y. According to this description, a designer designs the information flow path as shown in Figure 4a. However, to achieve this in actual operation of the system, before passing the value to calculation service Y, calculation service X invokes storage service Z. Then Z invokes calculation service z and passes the value to y. From this process, we can see clearly that an implicit information flow \( X \rightarrow Z \rightarrow Y \), which does not exist in the design phase, emerges, as shown in Figure 4b.

So, it is not enough to use architecture design specification and architecture diagrams to analyze the vulnerability of information flow in software architecture. It is necessary to construct a model that can capable of depict all potential information flow and propagation based on architecture. In the 18th century, Euler, a renowned mathematician, solved the famous problem of the seven bridges of Konigsberg by abstracting paths into a graph and hence set up graph theory. As a data structure, a graph can represent complex relationships between data elements that cannot be determined intuitively. In this paper, we model the information flow and propagation in architecture by use of graph theory.
3.4.1. Determination and Representation of Service Invocation Elements

There are many relationships between and among services, such as invocation, containment, association, aggregation, cascade, and replacement. In these relationships, invocation is the most capable of reflecting the direction and propagation of the carried information. Before determining the service invocation relationship, we first propose a definition for service invocation and a series of related concepts as the basis for describing the invocation relationship between services.

**Definition 6.** Service invocation. Service invocation refers to the process of information carried in one service is passed on to another service.

**Definition 7.** Set of service invocations. The set of service invocations is a set consisting of service invocation elements. It can be denoted by a quadruple \( I = (S,R,A,D) \), in which \( S \) stands for the pair of service involved in the invocation, \( R \) represents the invocation relationship, \( A \) is the operation sequence of service invocation, and \( D \) is the direction of data flow.

The definitions \( R, A, \) and \( D \) will be given in definitions 8–10.

Before defining the relationship of service invocation \( R \), forms of service invocation should be clarified, including the types of service invocation and forms of service interaction. In serviced-oriented software architecture, there are four types of service invocation: sequence structure, selection structure, parallel structure, and iteration structure [56], as shown in Figure 4. In data flow analysis, it is common to ignore the conditions of the paths, i.e., to assume that all paths are feasible [57]. As a result, based on the four types of service invocation, we can abstract three types of service interaction: one to one, one to many, and many to many. This is also consistent with the interaction relationship between common data elements [58]. Table 3 shows these three types of service interaction.

**Table 3.** Types of service interaction.

<table>
<thead>
<tr>
<th>Data Interaction Relationship</th>
<th>Service Interaction Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>One to one</td>
<td>( S_1 \rightarrow S_2 )</td>
</tr>
<tr>
<td>One to many</td>
<td>( S_1 \rightarrow S_2 )</td>
</tr>
<tr>
<td>Many to many</td>
<td>( S_3 \rightarrow S_1 )</td>
</tr>
</tbody>
</table>

**Definition 8.** The relationship of service invocation. The relationship of service invocation refers to the interaction form when service invocation occurs, which can be denoted by a tuple, \( R = (S_i,S_o) \). \( S_i \) stands for the invoking service, which is the service that invokes another service, and \( S_o \) stands for the invoked service, which is the service that is invoked. This tuple can be denoted as \( S_{io} \). For example, if service \( i \) invokes service \( j \), then the relationship of this invocation can be denoted as \( S_{ij} \).
Figure 5 and Table 3 provide examples to show the relationships of the four types of service invocation. For sequence structure, the corresponding type of service interaction is one to one, and the resulting relationship is S12 and S23 … Sn-1Sn. For parallel structure, the corresponding types of service interactions are one to many and many to one, and the resulting relationship are S1S2, S2S5 AND S1S3, S5S AND S1S4, S4S5. For selection structure, the corresponding type of service interaction is one to many and many to one, and the resulting relationship is S1S2, S2S5 OR S1S3, S3S OR S1S4, S4S5. For iteration structure, the corresponding type of service interaction is one to one, and the resulting relationship is S1S2 and S2S1.

![Types of service invocation](image)

Figure 5. Types of service invocation.

The service invocation process includes not only interaction types, but also operation sequence of service invocation A and direction of data flow D. Next, the operation sequence of service invocation A and the direction of data flow between services resulting from service invocation process will be defined.

**Definition 9.** Operation sequence of service invocation. Operation sequence of service invocation refers to the set of operations during service invocation. It can be denoted by a tuple, A = {r, w}, in which r represents reading operation and w represents writing operation.

**Definition 10.** Direction of data flow between services. Direction of data flow between services refers to the direction of data flow when reading and writing operation occur during service invocation, which can be denoted by the symbol “→”.

For example, if there exist data D1 ∈ Ss, D2 ∈ So, and a set of service invocation I = {Ss,So,r}, then this means invoking service Ss performs the reading operation on invoked service So. The direction of data flow can be represented by Ss → So, which means that when an invoking service performs the reading operation on an invoked service, the data flow from the invoked service to the invoking service. Similarly, if there exists data D1 ∈ Ss, D2 ∈ So, and a set of service invocation I = {Ss,So,w}, this means invoking service Ss performs the writing operation on invoked service So. The direction of data flow can be represented by Ss → So, which means that when an invoking service performs the writing operation on an invoked service, the data flow from the invoking service into the invoked service. In addition to performing the reading and writing operations separately, an operation sequence of service invocation can perform both operations concurrently. In this case, the direction of data flow can be explained as follows: If there exists data D1 ∈ Ss, D2 ∈ So and a set of service invocation I = {Ss,So,r and w}, it means the invoking service Ss performs both the reading and writing operations
on the invoked service $S_0$. The direction of data flow can be represented by $S_0 \rightarrow S_s$ and $S_s \rightarrow S_0$, which means that when an invoking service performs the reading and writing operations concurrently on an invoked service, data flows from the invoked service to the invoking service and from the invoking service into the invoked service at the same time.

3.4.2. Definition and Construction Approach for Service Invocation Diagram

From the research above, the information about service invocation can be obtained. Then by using graph theory, a service invocation diagram can be constructed, which can be used as basis for later vulnerability analysis of information flow in architecture.

To construct a service invocation diagram, we propose mapping rules between service invocation information obtained above and elements in the diagram. First, services are mapped into vertices in the diagram. Second, the invocation relationship is mapped into the edges. Third, the operation type of service invocation is mapped as weights on the edges. The weight value mapping from the reading operation is 1, and the weight value from the writing operation is 2. Moreover, the direction of an edge is based on the direction of data flow between services. As a result, the service invocation diagram constructed in this paper is a weighted and directed graph.

**Definition 11. Service invocation diagram.** Service invocation diagram is a data structure used to describe invocation information between services. It consists of a nonempty set of vertices $V$ (composed of $n$ vertices, $n > 0$) and a set of edges $E$ (relations between vertices), which can be denoted as

$$G = (V, E)$$

After learning the service invocation information and defining the service invocation diagram, we can describe the service invocation diagram by use of an adjacency matrix.

**Definition 12. Adjacency matrix of service invocation diagram.** For a service invocation diagram $G$, a one-dimensional array with $n$ elements $L[0, n-1]$ can be given to describe the information of vertices. Based on this, a two-dimensional array $A[0, n-1][0, n-1]$ can be given, which is the adjacency matrix for this service invocation diagram. In matrix $A$, the element $A_{ij}$ stores the information relation between vertices $i$ and $j$. The matrix can be denoted as

$$A_{ij} = \begin{cases} w_{ij} \\ 0 \end{cases}$$

When there exists a relationship between service $i$ and $j$, then there exists an edge between vertices $i$ and $j$, the weigh on the edge is $w_{ij}$. As mentioned above, $w_{ij} = 1$ or 2.

When there is no relationship between services $i$ and service $j$, the value of $w_{ij}$ is 0.

Finally, by traversing all invocation relations between services, the adjacency matrix and service invocation diagram can be obtained.

3.4.3. Approach for Determining Confidentiality and Integrity Levels of Services

After constructing the service invocation diagram, we need to assign values for the confidentiality and integrity levels of services. The core element for determining the security level of a service is to determine the security level of data carried on that service, that is, to assign values for the confidentiality level and integrity level for that data information. When assigning values, we need to comprehensively consider the confidentiality and integrity levels of the data as well as the security level of services. In this paper we adopt a method for assigning values and determining levels for assets from the
security area that are widely accepted [59–61] and considered practical. The equations for determining the confidentiality level and integrity level for services are as follows.

\[ \text{SER}_{cl} = \text{Round1} \sqrt{D_{cl} \times \text{Sec}_{cl} + 1} \] (1)

Equation (1) determines confidentiality level, in which \( \text{SER}_{cl} \) represents the confidentiality level of a service, \( D_{cl} \) represents the confidentiality level of data carried on that service, which is determined by the criticality level of data confidentiality, \( \text{Sec}_{cl} \) represents the security level of that service, and \( \text{Round1} \) means round up to one decimal. It is worth noting that +1 in the equation means the confidentiality of the service is higher than that of the data. To simplify, we take 1 for value 1.

\[ \text{SER}_{il} = \text{Round1} \sqrt{D_{il} \times \text{Sec}_{il} } \] (2)

Equation (2) determines the integrity level for services, in which \( \text{SER}_{il} \) represents the integrity level of a service, \( D_{il} \) represents the integrity level of data carried on that service, which is determined by the criticality level of data integrity, \( \text{Sec}_{il} \) represents the security level of the service, \( \text{Round1} \) means rounding up to one decimal.

The values of \( D_{cl} \) and \( D_{il} \) in Equations (1) and (2) are determined by experts according to Tables 4 and 5.

**Table 4. Criticality level of data confidentiality.**

<table>
<thead>
<tr>
<th>Value</th>
<th>Criticality Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Critical</td>
<td>Compromising of confidentiality will cause critical losses for system or software</td>
</tr>
<tr>
<td>3</td>
<td>Medium</td>
<td>Compromising of confidentiality will cause medium losses for system or software</td>
</tr>
<tr>
<td>2</td>
<td>Minor</td>
<td>Not very critical; compromising of confidentiality will cause minor losses for system or software</td>
</tr>
<tr>
<td>1</td>
<td>Low</td>
<td>Not crucial; compromising of confidentiality will cause only minimum losses for system or software</td>
</tr>
</tbody>
</table>

**Table 5. Criticality level of data integrity.**

<table>
<thead>
<tr>
<th>Value</th>
<th>Criticality Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Critical</td>
<td>Compromising of integrity will cause critical losses for system or software</td>
</tr>
<tr>
<td>3</td>
<td>Medium</td>
<td>Compromising of integrity will cause medium losses for system or software</td>
</tr>
<tr>
<td>2</td>
<td>Minor</td>
<td>Not very critical; compromising of integrity will cause minor losses for system or software</td>
</tr>
<tr>
<td>1</td>
<td>Low</td>
<td>Not crucial; compromising of integrity will cause only minimum losses for system or software</td>
</tr>
</tbody>
</table>

As for security level of services in Equations (1) and (2), when the security of the service is compromised, the software system, the user, and the organization to which the service belongs will be impacted. Different entities will suffer from the impacts of different severity levels. In this paper, we define three severity levels: medium, critical, and extreme. As a result, the security levels impacted by objects and severity levels together can be classified into five level, \{1,2,3,4,5\}. Level 1 means there is almost no impact; level 2 means there are minor impacts; level 3 means there are some medium impacts; level 4 means there are some significant impacts; level 5 means there are critical impacts. Referring to [62], we propose a determination matrix for service security levels, as shown in Table 6. According to this matrix, combined with the characteristics of the software system, and under the guidance of expert experience, the security level of each service can be obtained.

Finally, through Equations (1) and (2), the confidentiality and integrity levels of services can be obtained and thus a service invocation diagram based on architecture can be constructed. In the rest of the paper, we will use this model to locate vulnerabilities of information flow in software architecture.
Table 6. Determination matrix for service security level.

<table>
<thead>
<tr>
<th>Entity Impacted When Service Security is Compromised</th>
<th>Severity Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>software system</td>
<td>Medium</td>
</tr>
<tr>
<td>software system Level 1</td>
<td>Level 1</td>
</tr>
<tr>
<td>software system Level 2</td>
<td>Level 2</td>
</tr>
<tr>
<td>software system Level 3</td>
<td>Level 3</td>
</tr>
<tr>
<td>User</td>
<td>Level 2</td>
</tr>
<tr>
<td>Organization to which software system belongs</td>
<td>Level 3</td>
</tr>
</tbody>
</table>

3.5. Algorithm for Determining Vulnerability of Information Flow in Architecture

There are two major tasks in locating vulnerabilities based on the service invocation diagram: First, find all paths of information flow and propagation. Second, based on the confidentiality or integrity level of each vertex on the paths as well as the direction of information flow between adjacent vertices, locate vulnerabilities by looking for vertices that violate the security policies of information flow proposed above. In this paper, two algorithms are designed to identify and locate vulnerabilities in the service invocation diagram based on confidentiality and integrity policies. Key points of the algorithms are as follows:

- Use all vertices with an in degree of 0 as initial vertices, and sort them.
- Find paths that begin with these vertices with an in degree of 0 one by one. Stop until no vertices interact with these vertices. Then output these paths. Although the reading and writing operation (denoted by “r and w”) will cause loops in the diagram, there is no need to iterate these loops when finding paths, because by traversing the loop once, services pairs that violate security policies will be located. So, when finding paths, we only consider the loops once.
- Use these paths and determination rules under the goals of confidentiality or integrity as input. Compare the confidentiality or integrity levels of two adjacent vertices. For example, if the path is N₁, N₃, N₅, N₉, then compare with N₁N₃, N₃N₅, N₅N₉, respectively, in which N stands for a node.
- Save and output the vertices that conflict with determination rules, as well as their interaction relationships, until all vertices of each path are compared.

Algorithm 1 is the algorithm for determining the vulnerability of information flow in architecture under the confidentiality policies. Algorithm 2 is the algorithm for determining the vulnerability of information flow in architecture under the integrity policies.

Algorithm 1 Determining vulnerability under confidentiality policies

```
1:  Input a graph, G; initialize a list of paths allPaths and list of vulnerable nodes vNodes
2:  i ← 0
3:  for each node n in G do
4:     if in-degree of n=0 then
5:         while adjacent node j of node n exists do
6:             i ← FindPath(i,n,n,allPaths)
7:             i ← i+1
8:         end while
9:     end if
10: end for
11: for each path p in allPaths do
12:     for k ← 0 to p.length-2 do
13:         if security level of p[k]<p[k+1] then
14:             add p[k] in vNodes
15:             add p[k+1] in vNodes
```
16: end if
17: end for
18: end for
19: return allPaths and vNodes
20: Function FindPath(num, currentNode, root, allPaths)
21: while adjacent node y of currentNode exists do
22: add y in allPaths[num]
23: num ← num+1
24: for each node z in allPaths[num-1] do
25: add z in allPaths[num]
26: end for
27: num ← FindPath(num, y, root, allPaths)
28: end while
29: return num

Algorithm 2 Determining vulnerability under integrity policies
1: Input a graph, G; initialize a list of paths allPaths and list of vulnerable nodes vNodes
2: i ← 0
3: for each node n in G do
4: if in-degree of n=0 then
5: while adjacent node j of node n exists do
6: i ← FindPath(i, n, n, allPaths)
7: i ← i+1
8: end while
9: end if
10: end for
11: for each path p in allPaths do
12: for k ← 0 to p.length-2 do
13: if security level of p[k] > p[k+1] then
14: add p[k] in vNodes
15: add p[k+1] in vNodes
16: end if
17: end for
18: end for
19: return allPaths and vNodes
20: Function FindPath(num, currentNode, root, allPaths)
21: while adjacent node y of currentNode exists do
22: add y in allPaths[num]
23: num ← num+1
24: for each node z in allPaths[num-1] do
25: add z in allPaths[num]
26: end for
27: num ← FindPath(num, y, root, allPaths)
28: end while
29: return num

4. Case Study

4.1. Purpose

In this section, a real-world software architecture was selected as an example. When designing this system, non-functional requirements were neglected. The methods proposed in this paper were used to analyze its architecture. Results were observed to see whether vulnerabilities in the architecture could be located in order to verify the feasibility and effectiveness of the proposed method.
4.2. Introduction of the Object System

The object system used in the case study was a flight reservation system of an airline company. The architecture of the system was service-oriented. Its main functions included user information input and statistics, user information storage, transmission and receipt of ticket information, notice of ticket pick-up, generation and printing of bills, statistics and checking of ticket sales, processing of ticket information based on user needs, generation of flight information, inquiry and feedback of flight information, inquiry and feedback of travel agencies, analysis and management of sales, and so on. To implement these functions, designers classified the system into four major modules: client-side information reception, client-side information output, network reception and dispatch, and server modules. Figure 6 shows the composition of this system. Software designers performed the preliminary design and detailed design and wrote a software architecture design document for this system.

![Diagram of the flight reservation system](image)

**Figure 6.** The modules of the flight reservation system.

4.3. Experimental Design and Environment

In this paper, the client-side information reception module was selected as the experimental object to verify the analysis approach for vulnerability of information flow under the goal of confidentiality proposed in this paper. The network reception and dispatch module was selected as the experimental object to verify the analysis approach for vulnerability of information flow under the goal of integrity. The experiment was conducted on a computer installed with Ubuntu 19.04, PyCharm IDE, and an assisted modelling tool called Graphviz.

4.4. Experimental Process

Step 1. We used the architecture design document to determine and describe the services included in the client-side information reception module and network reception and dispatch module. The service composition of these two modules, the functions of these services, and the process to realize the whole module are shown in Appendices A and A.

Step 2. According to Appendices A and A and following the methods proposed in Section 3.4, we analyzed the client-side information reception module and network reception and dispatch module to obtain their service invocation elements and the confidentiality or integrity level of each service. The service invocation elements and confidentiality levels of each service in the client-side information reception module are shown in Tables 7 and 8. The service invocation elements and integrity levels of each service in the network reception and dispatch module are shown in Tables 9 and 10.
Table 7. Service invocation elements in client-side information reception module.

<table>
<thead>
<tr>
<th>Service Invocation Pair</th>
<th>Operations Performed</th>
<th>Direction of Data Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>S&lt;sub&gt;1&lt;/sub&gt; invokes S&lt;sub&gt;0&lt;/sub&gt;</td>
<td>S&lt;sub&gt;1&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;0&lt;/sub&gt;</td>
<td>S&lt;sub&gt;0&lt;/sub&gt; → S&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;1&lt;/sub&gt; invokes S&lt;sub&gt;2&lt;/sub&gt;</td>
<td>S&lt;sub&gt;1&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;2&lt;/sub&gt;</td>
<td>S&lt;sub&gt;2&lt;/sub&gt; → S&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;2&lt;/sub&gt; invokes S&lt;sub&gt;3&lt;/sub&gt;</td>
<td>S&lt;sub&gt;2&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;3&lt;/sub&gt;</td>
<td>S&lt;sub&gt;3&lt;/sub&gt; → S&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;3&lt;/sub&gt; invokes S&lt;sub&gt;4&lt;/sub&gt;</td>
<td>S&lt;sub&gt;3&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;4&lt;/sub&gt;</td>
<td>S&lt;sub&gt;4&lt;/sub&gt; → S&lt;sub&gt;3&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;4&lt;/sub&gt; invokes S&lt;sub&gt;0&lt;/sub&gt;</td>
<td>S&lt;sub&gt;4&lt;/sub&gt; Performs reading operation on S&lt;sub&gt;0&lt;/sub&gt;</td>
<td>S&lt;sub&gt;0&lt;/sub&gt; → S&lt;sub&gt;4&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;4&lt;/sub&gt; invokes S&lt;sub&gt;5&lt;/sub&gt;</td>
<td>S&lt;sub&gt;4&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;5&lt;/sub&gt;</td>
<td>S&lt;sub&gt;5&lt;/sub&gt; → S&lt;sub&gt;4&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;4&lt;/sub&gt; invokes S&lt;sub&gt;7&lt;/sub&gt;</td>
<td>S&lt;sub&gt;4&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;7&lt;/sub&gt;</td>
<td>S&lt;sub&gt;7&lt;/sub&gt; → S&lt;sub&gt;4&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;5&lt;/sub&gt; invokes S&lt;sub&gt;6&lt;/sub&gt;</td>
<td>S&lt;sub&gt;5&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;6&lt;/sub&gt;</td>
<td>S&lt;sub&gt;6&lt;/sub&gt; → S&lt;sub&gt;5&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;6&lt;/sub&gt; invokes S&lt;sub&gt;8&lt;/sub&gt;</td>
<td>S&lt;sub&gt;6&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;8&lt;/sub&gt;</td>
<td>S&lt;sub&gt;8&lt;/sub&gt; → S&lt;sub&gt;6&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;1&lt;/sub&gt; invokes S&lt;sub&gt;7&lt;/sub&gt;</td>
<td>S&lt;sub&gt;2&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;7&lt;/sub&gt;</td>
<td>S&lt;sub&gt;7&lt;/sub&gt; → S&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;2&lt;/sub&gt; invokes S&lt;sub&gt;9&lt;/sub&gt;</td>
<td>S&lt;sub&gt;2&lt;/sub&gt; Performs reading operation on S&lt;sub&gt;9&lt;/sub&gt;</td>
<td>S&lt;sub&gt;9&lt;/sub&gt; → S&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;8&lt;/sub&gt; invokes S&lt;sub&gt;10&lt;/sub&gt;</td>
<td>S&lt;sub&gt;8&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;10&lt;/sub&gt;</td>
<td>S&lt;sub&gt;10&lt;/sub&gt; → S&lt;sub&gt;8&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;9&lt;/sub&gt; invokes S&lt;sub&gt;11&lt;/sub&gt;</td>
<td>S&lt;sub&gt;9&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;11&lt;/sub&gt;</td>
<td>S&lt;sub&gt;11&lt;/sub&gt; → S&lt;sub&gt;9&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

Table 8. Confidentiality values of each service invocation element in client-side information reception module.

<table>
<thead>
<tr>
<th>Service</th>
<th>Confidentiality Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service S&lt;sub&gt;0&lt;/sub&gt;</td>
<td>2.4</td>
</tr>
<tr>
<td>Service S&lt;sub&gt;1&lt;/sub&gt;</td>
<td>4</td>
</tr>
<tr>
<td>Service S&lt;sub&gt;2&lt;/sub&gt;</td>
<td>4</td>
</tr>
<tr>
<td>Service S&lt;sub&gt;3&lt;/sub&gt;</td>
<td>4</td>
</tr>
<tr>
<td>Service S&lt;sub&gt;4&lt;/sub&gt;</td>
<td>4</td>
</tr>
<tr>
<td>Service S&lt;sub&gt;5&lt;/sub&gt;</td>
<td>4</td>
</tr>
<tr>
<td>Service S&lt;sub&gt;6&lt;/sub&gt;</td>
<td>4</td>
</tr>
<tr>
<td>Service S&lt;sub&gt;7&lt;/sub&gt;</td>
<td>4</td>
</tr>
<tr>
<td>Service S&lt;sub&gt;8&lt;/sub&gt;</td>
<td>5</td>
</tr>
<tr>
<td>Service S&lt;sub&gt;9&lt;/sub&gt;</td>
<td>5</td>
</tr>
<tr>
<td>Service S&lt;sub&gt;10&lt;/sub&gt;</td>
<td>5</td>
</tr>
<tr>
<td>Service S&lt;sub&gt;11&lt;/sub&gt;</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 9. Service invocation elements in network reception and dispatch module.

<table>
<thead>
<tr>
<th>Service Invocation Pair</th>
<th>Operations Performed</th>
<th>Direction of Data Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>S&lt;sub&gt;1&lt;/sub&gt; invokes S&lt;sub&gt;0&lt;/sub&gt;</td>
<td>S&lt;sub&gt;0&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;0&lt;/sub&gt;</td>
<td>S&lt;sub&gt;0&lt;/sub&gt; → S&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;1&lt;/sub&gt; invokes S&lt;sub&gt;2&lt;/sub&gt;</td>
<td>S&lt;sub&gt;1&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;2&lt;/sub&gt;</td>
<td>S&lt;sub&gt;2&lt;/sub&gt; → S&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;4&lt;/sub&gt; invokes S&lt;sub&gt;5&lt;/sub&gt;</td>
<td>S&lt;sub&gt;4&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;5&lt;/sub&gt;</td>
<td>S&lt;sub&gt;5&lt;/sub&gt; → S&lt;sub&gt;4&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;4&lt;/sub&gt; invokes S&lt;sub&gt;7&lt;/sub&gt;</td>
<td>S&lt;sub&gt;4&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;7&lt;/sub&gt;</td>
<td>S&lt;sub&gt;7&lt;/sub&gt; → S&lt;sub&gt;4&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;4&lt;/sub&gt; invokes S&lt;sub&gt;9&lt;/sub&gt;</td>
<td>S&lt;sub&gt;4&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;9&lt;/sub&gt;</td>
<td>S&lt;sub&gt;9&lt;/sub&gt; → S&lt;sub&gt;4&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;5&lt;/sub&gt; invokes S&lt;sub&gt;6&lt;/sub&gt;</td>
<td>S&lt;sub&gt;5&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;6&lt;/sub&gt;</td>
<td>S&lt;sub&gt;6&lt;/sub&gt; → S&lt;sub&gt;5&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;7&lt;/sub&gt; invokes S&lt;sub&gt;8&lt;/sub&gt;</td>
<td>S&lt;sub&gt;7&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;8&lt;/sub&gt;</td>
<td>S&lt;sub&gt;8&lt;/sub&gt; → S&lt;sub&gt;7&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;8&lt;/sub&gt; invokes S&lt;sub&gt;9&lt;/sub&gt;</td>
<td>S&lt;sub&gt;8&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;9&lt;/sub&gt;</td>
<td>S&lt;sub&gt;9&lt;/sub&gt; → S&lt;sub&gt;8&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;9&lt;/sub&gt; invokes S&lt;sub&gt;10&lt;/sub&gt;</td>
<td>S&lt;sub&gt;9&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;10&lt;/sub&gt;</td>
<td>S&lt;sub&gt;10&lt;/sub&gt; → S&lt;sub&gt;9&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;10&lt;/sub&gt; invokes S&lt;sub&gt;11&lt;/sub&gt;</td>
<td>S&lt;sub&gt;10&lt;/sub&gt; Performs writing operation on S&lt;sub&gt;11&lt;/sub&gt;</td>
<td>S&lt;sub&gt;11&lt;/sub&gt; → S&lt;sub&gt;10&lt;/sub&gt;</td>
</tr>
<tr>
<td>S&lt;sub&gt;11&lt;/sub&gt; invokes S&lt;sub&gt;12&lt;/sub&gt;</td>
<td>S&lt;sub&gt;12&lt;/sub&gt; Performs reading operation on S&lt;sub&gt;12&lt;/sub&gt;</td>
<td>S&lt;sub&gt;12&lt;/sub&gt; → S&lt;sub&gt;11&lt;/sub&gt;</td>
</tr>
</tbody>
</table>
Table 10. Integrity values of service invocation elements in network reception and dispatch module.

<table>
<thead>
<tr>
<th>Service</th>
<th>Integrity Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service S0</td>
<td>4.5</td>
</tr>
<tr>
<td>Service S1</td>
<td>3</td>
</tr>
<tr>
<td>Service S2</td>
<td>2.4</td>
</tr>
<tr>
<td>Service S3</td>
<td>2.4</td>
</tr>
<tr>
<td>Service S4</td>
<td>2.4</td>
</tr>
<tr>
<td>Service S5</td>
<td>2.4</td>
</tr>
<tr>
<td>Service S6</td>
<td>2.4</td>
</tr>
<tr>
<td>Service S7</td>
<td>2.4</td>
</tr>
<tr>
<td>Service S8</td>
<td>2.4</td>
</tr>
<tr>
<td>Service S9</td>
<td>2.4</td>
</tr>
<tr>
<td>Service S10</td>
<td>2</td>
</tr>
<tr>
<td>Service S11</td>
<td>1</td>
</tr>
<tr>
<td>Service S12</td>
<td>1</td>
</tr>
</tbody>
</table>

Step 3. Based on the obtained service invocation information above, the service invocation diagram was constructed. Then, we used Graphviz to visualize the service invocation diagrams for client-side information reception and network reception and dispatch modules, as shown in Figures 7 and 8, respectively. In these figures, the value 1 on the edge stands for a reading operation, while 2 stands for a writing operation.

Figure 7. Service invocation diagram of client-side information reception module.
Figure 8. Service invocation diagram of network reception and dispatch module.

Step 4. We used the two algorithms for determining the vulnerability of information flow in architecture proposed in this paper to locate vulnerabilities in the client-side information reception module and network reception and dispatch module. We developed a program which implemented these two algorithms and used the program to obtain the analysis results.

4.5. Result Analysis and Threats to Validity

4.5.1. Results

The results were obtained as follows.

For the client-side information reception module, we applied the algorithm for determining the vulnerability of information flow in architecture under the goal of confidentiality. The results are shown in Figure 9.

Figure 9. Output of vulnerability determination algorithm under the goal of confidentiality.

Paths containing vulnerabilities were obtained:
P4: $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_0$; P6: $S_0 \rightarrow S_4 \rightarrow S_7 \rightarrow S_0$. 
The corresponding vulnerability information is $S_7 \rightarrow S_0$. The reason for why this is vulnerable will be discussed in detail in Section 4.5.2.

For the network reception and dispatch module, we applied the algorithm for determining the vulnerability of information flow in architecture under the goal of integrity. The results are shown in Figure 10.

![Figure 10. Output of vulnerability determination algorithm under the goal of integrity.](image)

Paths containing vulnerabilities were obtained:
- P3: $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_{10} \rightarrow S_0 \rightarrow S_{10} \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_{12}$
- P4: $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_{10} \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_{12}$
- P5: $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_{10} \rightarrow S_9 \rightarrow S_{10} \rightarrow S_{11}$
- P8: $S_0 \rightarrow S_1 \rightarrow S_6 \rightarrow S_7 \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10} \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_{12}$

along with the corresponding vulnerability information:
- $S_{10} \rightarrow S_3$ and $S_{10} \rightarrow S_9$

The reason for why this is vulnerable will be discussed in detail in Section 4.5.2.

4.5.2. Results Analysis and Discussion

From the results of the vulnerability analysis of information flow in architecture under the goal of confidentiality conducted in the case, it can be seen that the vulnerability was $S_7 \rightarrow S_0$ (detailed descriptions of $S_7$ and $S_0$ are provided in Appendix A), which violated the first security policy of information flow under this goal. By analyzing the architecture design specifications with the help of domain experts, we learned that when verification service returned an error value to the login service, user and ticket information could be leaked, which could cause leakage of users’ private information such as name and itinerary. In addition, malicious attackers could use this vulnerability to try logging in to the system, for example by inputting erroneous information on purpose, and obtain return values of the system, which could be used to guess users’ passwords. If the login succeeded, they could imitate users to modify ticket information or conduct other operations in the system. According to the analysis above, we provided this feedback on architecture vulnerability to the developers and suggested that they modify the architecture. Specifically, we suggested that the developers bind the verification service with a login service in order to make sure that users’ identities can be verified when they attempt to login, rather than after their information is saved, as in this system.

From the result of the vulnerability analysis of information flow in architecture under the goal of integrity conducted in the case study, it can be seen that the vulnerabilities were $S_{10} \rightarrow S_3$ and $S_{10} \rightarrow S_9$ (detailed descriptions of $S_3$, $S_9$ and $S_{10}$ are provided in Appendix B), which violated the fourth security policy of information flow under the goal of integrity. By analyzing the architecture design specifications, we learned that $S_{10}$ was a monitoring service that could monitor and intervene in $S_3$ and $S_9$ by collecting their runtime information. However, in the design specifications, it was not stated clearly that $S_{10}$ needed a specific transmission protocol such as virtual private network
(VPN) to transfer data, nor was an encryption mechanism mentioned for data receipt and sending. What is worse, there was no permission or access control mechanism for the monitoring service. As a result, in operation, malicious attackers could modify, intercept, or delete data in S3 and S9 by attacking the monitoring service. This analysis result conforms to the reality that most software systems nowadays, especially complex software systems have integrated monitoring services, and because of the complex functional, performance, and accuracy requirement of the monitoring function, many designers tend to neglect security requirements when implementing the monitoring function, which could pose significant threats to the operation of software systems [63]. According to the analysis above, we provided feedback on these vulnerabilities to the developers and suggested that they modify the architecture. For example, we suggested employing VPN and data encryption for data transfer between the monitoring service and the main system, and the monitor service should not have operation privileges for the system.

To further demonstrate the effectiveness of the proposed method, we compare the vulnerability analysis results with common security vulnerabilities from famous vulnerability databases, including Common Vulnerabilities and Exposures (CVE), Open Web Application Security project (OWAP), and National Vulnerability Database (NVD). Common categories of vulnerabilities in these databases are listed in Table 11.

Table 11. Common categories of vulnerabilities in Common Vulnerabilities and Exposures (CVE), Open Web Application Security (OWAP), and National Vulnerability Database (NVD).

<table>
<thead>
<tr>
<th>Number</th>
<th>Category of Vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Buffer overflows</td>
</tr>
<tr>
<td>2</td>
<td>SQL injection</td>
</tr>
<tr>
<td>3</td>
<td>Cross-site scripting</td>
</tr>
<tr>
<td>4</td>
<td>Path traversal</td>
</tr>
<tr>
<td>5</td>
<td>Permission, privileges, and access controls</td>
</tr>
<tr>
<td>6</td>
<td>Cryptographic issues</td>
</tr>
<tr>
<td>7</td>
<td>Information transfer with plaintext</td>
</tr>
<tr>
<td>8</td>
<td>Unvalidated input</td>
</tr>
<tr>
<td>9</td>
<td>Improper error handling</td>
</tr>
<tr>
<td>10</td>
<td>Link following</td>
</tr>
<tr>
<td>11</td>
<td>Format string</td>
</tr>
<tr>
<td>12</td>
<td>Authentication issue, improper authentication</td>
</tr>
</tbody>
</table>

In Table 11, we marked four categories of vulnerabilities in Italic form. The vulnerabilities from the results of the case study belong to these four categories.

Besides, we learned from recent news that there are security vulnerabilities in Amadeus flight booking system [64]. The attacker just needs to know the victim’s passenger name record (PNR) number to exploit the vulnerability. By using the booking information (i.e., booking ID and last name of the customer) it is possible to access the user’s account and perform other operations. What is worse, the user formation is transmitted without encryption in Amadeus system, which may draw man-in-the-middle attack. It can be seen that the vulnerabilities in Amadeus system are similar to the analysis results in our case study, which further verify the effectiveness of our proposed methods.

4.5.3. Comparison with Other Methods

In previous subsections, we demonstrated the feasibility and effectiveness of the proposed method. To further show the advantages of our method over existing methods, we compare our method with others.

As mentioned in the Introduction, the major contribution of this paper is that we propose a method for locating vulnerabilities in software architecture in the design phase, so the comparison with other methods is also based on this perspective. The comparison results are shown in Table 12.
Table 12. Comparison with other methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>This Paper</th>
<th>[25]</th>
<th>[26]</th>
<th>[27]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object used</td>
<td>Architecture</td>
<td>Design phase</td>
<td>Architecture</td>
<td>Design phase</td>
</tr>
<tr>
<td>Development phase that can be applied</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Are multiple versions of architecture needed?</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can it be used to locate vulnerabilities?</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

From Table 12, it can be seen that compared with the method in [25], our method can be used to locate architecture vulnerabilities, while that one cannot. The method in [26] can be used to locate vulnerabilities in architecture. However, the architecture in that method must be obtained after the software development is finished, because in that vulnerability locating method, the architecture of the developed software must be compared with the original version of the software architecture. Then, by injecting malicious source code, vulnerabilities can finally be discovered. As a result, that method cannot be applied in the design phase. The concept of architecture vulnerability is mentioned in [27], but the object of the case study was the source code and vulnerabilities were not located at all.

4.5.4. Threats to Validity

There are some risks and threats to validity, include the following:

1. When assigning values for service confidentiality or integrity, although the assigning method proposed in this paper is based on widely accepted method in cyber security, such as CC (ISO/IEC Standard 15,408) [60] and ISO/IEC Standard 27,000 [59], in a specific assigning activity the method may be impacted by the domain knowledge and experience of researchers, as well as by whether the object system is thoroughly understood. So, when assigning values in this paper, in addition to researchers who possess domain knowledge, we also consulted experienced practitioners and invited them to discuss and confirm the value-assigning results. As a result, to some extent, there might be some subjectivity in the assigning results, but the correctness of the proposed method can be guaranteed.

2. The outputs of the proposed vulnerability analysis method are the services and interactions between services that violate security policies. Further analysis on the outputs requires domain background and experiences of analyzers. In this paper, we invited domain experts who possess strong background knowledge and domain experience. So, the further analysis on the results in this paper is solid.

3. A modeling approach and corresponding vulnerability analysis method is proposed in this paper, and concrete algorithm are given. However, automatic tools for the proposed work are lacking, which will be taken into consideration in our future work.

5. Conclusions and Future Work

In this paper, we propose an approach for analyzing the vulnerability of information flow in software architecture. First, based on confidentiality and integrity policies of information flow and current definitions of software vulnerability and combining features of software architecture, the concept and formal description of vulnerability of information flow in software architecture are given. We clarify that the vulnerability of information flow in software architecture refers to errors resulting from the violation of corresponding security policies during the flow and propagation of information in components, interactions between components, and the topology formed from those interactions. Moreover, targeting at the reading and writing operations during service interaction, security policies of information flow under the goals of confidentiality and integrity are proposed. Then, we propose an approach for constructing a service invocation diagram based on software architecture to depict the interaction and propagation of information flow in architecture as well as the confidentiality or integrity level of each service. After that, an algorithm is designed to located vulnerabilities of information flow in architecture by traversing all potential paths and comparing these
paths with predefined security policies of information flow. Finally, a flight reservation system of an airline company was chosen as a case study. After conducting experiments, we found a vulnerability of information flow in the architecture by using confidentiality policies, which was the possibility of information leakage involving users and tickets when the verification service returns error values to the login service. By using integrity policies, we found two vulnerabilities of information flow in the architecture, concerning integrity hazards of the monitoring service. The monitoring service has unnecessarily high privilege when interacting with the system and there is no encryption mechanism for data read from this monitored system. From the case study, we can see that the proposed approaches can be used to locate vulnerabilities of information flow in architecture, thus the effectiveness and applicability can be verified. The proposed method was compared with other methods, and the results show the advantages of our method.

In the future, we will apply the proposed method to more software systems to further verify its effectiveness and usability and improve it. In addition, we will conduct research on value-assigning methods for confidentiality and integrity of services to remedy the shortcoming of the current method which relies highly on experts’ experience, and improve the accuracy of the assigning method. Moreover, we will extend and complement the proposed modelling method by integrating existing security constrains of the architecture into the model. Finally, corresponding tools will be developed to support the proposed modelling approach and determination algorithms. With the refinement of our method and accumulation of experimental data, we will also consider incorporating machine learning methods from intelligent computing into our work, such as in [65,66]. In addition to the above work, we will also try to apply our research to automotive field, such as autopilot systems. We found several studies on reliability issues in this area [67,68]. As an important quality attribute, security should also be considered in future research.

Author Contributions: Conceptualization, T.G. and L.L.; funding acquisition, M.L.; investigation, T.G.; methodology, T.G., M.L., and L.L.; project administration, M.L.; resources, T.G.; supervision, M.L. and Q.L.; validation, T.G. and L.L.; visualization, T.G. and L.L.; writing—original draft, T.G. and M.L.; writing—review and editing, T.G., M.L., L.L., and Q.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by “National Key Laboratory of Science &Technology on Reliability & Environmental Engineering of China”, grant number “WDZC2019601A303”.

Acknowledgments: The authors would like to thank Zhang Dajian, chief manager at the Testing Center of Finance, China, Chen Qinmin, senior security engineer at the Beijing Certificate Authority, and Zeng Yong, architecture designer expert at the Xiangcao Software Company for their suggestions on value determination in Section 3.4.3 and further analysis on our experiment results in Section 4.5.2.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

Appendix A describes services in client-side information reception module.

To implement the major functions of the client-side information reception module, designers performed detailed design and wrote a software design document. According to the software design document, the workflow of the client-side information reception module can be abstracted into the following 12 services. The name and description and each service are shown in Table A1. To ease the modelling process, we numbered these services from 0 to 11.
Table A1. Name and description of each service in client-side information reception module.

<table>
<thead>
<tr>
<th>Service</th>
<th>Name and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service S0</td>
<td>Entrance service. The starting point of the module.</td>
</tr>
<tr>
<td>Service S1</td>
<td>PersInfoExam service. Performs preliminary examination for information input in traveler’s information panel. If errors exist, determines the type of error and uses corresponding PErrorType or PErrorRank as parameters to transmit into ErrorHandle service. If no errors exist, transfers to PersInfoInput service.</td>
</tr>
<tr>
<td>Service S2</td>
<td>PersInfoInput service. After examination without error, transmits data from input datasheet to PersInfoTempSave service.</td>
</tr>
<tr>
<td>Service S3</td>
<td>PersInfoTempSave service. Receives traveler’s information from PersInfoInput service and saves to a temp file PersInfoTemp.txt. This temp file will be used to verify data transferred from the server and should be deleted after the whole process is complete.</td>
</tr>
<tr>
<td>Service S4</td>
<td>CheckNoticeExam service. Performs preliminary verification of information input in check and ticket collection panel. If errors exist, determines the type of error and uses corresponding CErrorType or CErrorRank as parameters to transmit to ErrorHandle service. If no errors exist, transfer into CheckNoticeInput service.</td>
</tr>
<tr>
<td>Service S5</td>
<td>CheckNoticeInput service. After examination without error, transmits data from input datasheet to CheckNoticeTempSave service.</td>
</tr>
<tr>
<td>Service S6</td>
<td>CheckNoticeTempSave service. Receives traveler’s check and ticket collection information from CheckNoticeInput service and saves to a temp file CheckNoticeTemp.txt. This temp file will be used to verify data transferred from the server and should be deleted after the whole process is complete.</td>
</tr>
<tr>
<td>Service S7</td>
<td>ErrorHandle service. If errors are found in PersInfoInput or CheckNoticeInput service, transfers to this service. Performs corresponding dealing process and outputs error message.</td>
</tr>
<tr>
<td>Service S8</td>
<td>OrderCollectDetermination service. Determines whether this operation is ordering or collecting tickets based on the input of PersInfoInput service and set flags. According to the flags, transfers to OrderRequireTrans or CollectRequireTrans service.</td>
</tr>
<tr>
<td>Service S9</td>
<td>OrderRequireTrans service. Before transferring order requirements, performs hardware and software preparation, for example, prepares ticket ordering information, including traveler’s information, host’s information, and so on.</td>
</tr>
<tr>
<td>Service S10</td>
<td>CollectRequireTrans service. Before transferring collecting requirements, performs hardware and software preparation, for example, prepares ticket collecting information, including traveler’s information, host’s information, and so on.</td>
</tr>
<tr>
<td>Service S11</td>
<td>DataTrans service. Transmits information from client-side to server.</td>
</tr>
</tbody>
</table>

Appendix B

Appendix B describes services in network reception and dispatch module.

To implement the major functions of the network reception and dispatch module, designers performed detailed design and wrote a software design document. According to the software design document, the workflow of the network reception and dispatch module can be abstracted into the following 13 services. The name and description of each service are shown in Table A2. To ease the modelling process, we numbered these services from 0 to 12.

Table A2. Name and description of each service in network reception and dispatch module.

<table>
<thead>
<tr>
<th>Service</th>
<th>Name and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service S0</td>
<td>Entrance service. The starting point of the module.</td>
</tr>
<tr>
<td>Service S1</td>
<td>NetCheck service. Checks the status of the network. Sends some checking information and receives some feedback information. Checks whether the endpoint to establish a path is valid. If valid, prepares for response, and transfers to RecPre and SendPre service. If not, transfers to NErrorHandle service.</td>
</tr>
<tr>
<td>Service S2</td>
<td>RecPre service. After NetCheck service checks the network, prepares for receiving data.</td>
</tr>
<tr>
<td>Service S3</td>
<td>NetReceive service. Receives network packet. When receiving data packets, keeps connection with NetMonitor service. Accepts and returns network data transmission status. According to status information, continues receiving data packets or adjusts correspondingly.</td>
</tr>
<tr>
<td>Service S4</td>
<td>NetDataReceiveConfirm service. Confirms the acceptance of data transmitted from NetReceive service.</td>
</tr>
<tr>
<td>Service S5</td>
<td>NetDataUnpack service. Unpacks received data according to specific algorithm.</td>
</tr>
</tbody>
</table>
Table A2. Cont.

<table>
<thead>
<tr>
<th>Service</th>
<th>Name and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service S₆</td>
<td>SendPre service. After NetCheck service checks the network, prepares for sending data.</td>
</tr>
<tr>
<td>Service S₇</td>
<td>NetDataPack service. Packs data transmitted from server module according to specific algorithm.</td>
</tr>
<tr>
<td>Service S₈</td>
<td>NetDataSendConfirm service. Confirms before sending data to network.</td>
</tr>
<tr>
<td>Service S₉</td>
<td>NetSend service. Sends packed data packets. When sending data packets, keep connection with NetMonitor service. Accepts and returns network data transmission status. According to status information, continues sending data packets or adjusts correspondingly.</td>
</tr>
<tr>
<td>Service S₁₀</td>
<td>NetMonitor service. From establishment of connection from server to client to completion of data transmission and disconnection, NetMonitor services keeps monitoring network status. Meanwhile, sends status information to NetReceive and NetSend service, and receives status information of data receiving and sending from these two services. If errors occur in the status information, transfers to NErrorHandle service.</td>
</tr>
<tr>
<td>Service S₁₁</td>
<td>NErrorHandle service. Handles errors in network transmission module.</td>
</tr>
<tr>
<td>Service S₁₂</td>
<td>JudgeCSFlag service. Based on SetCSFlag in client input or server module, determines which module should receive transfer after network module is completed, client output or server module.</td>
</tr>
</tbody>
</table>

References

2. Krsul, I.V. Software Vulnerability Analysis; Purdue University: West Lafayette, IN, USA, 1998.


34. Aiken, A. Constraint-Based Program Analysis; Springer: Berlin/Heidelberg, Germany, 1999; p. 388.


53. Biba, K.J. Integrity Considerations for Secure Computer Systems; Purdue University Reports and Papers Archive; Purdue University: West Lafayette, IN, USA, 1977; pp. 1–21.


