


Secure Partitioning of Composite Cloud Applications

Alessandro Bocci ¹, Roberto Guanciale², Stefano Forti¹, Gian-Luigi Ferrari¹,
and Antonio Brogi¹

¹ University of Pisa, Pisa, Italy
alessandro.bocci@phd.unipi.it

² KTH Royal Institute of Technology, Stockholm, Sweden

Abstract. The security of Cloud applications is always a major concern for application developers and operators. Protecting their users' data confidentiality requires methods to avoid leakage from vulnerable software and unreliable cloud providers. Recently, hardware-based technologies emerged in the Cloud setting to isolate applications from the privileged access of cloud providers. One of those technologies is the Separation Kernel which aims at isolating safely the software components of applications. In this article, we propose a declarative methodology supported by a running prototype to determine the partitioning of a Cloud multi-component application in order to allow its placement on a Separation Kernel. We employ information-flow security techniques to determine how to partition the application, and showcase the methodology and prototype over a motivating scenario from an IoT application deployed to a central Cloud.

Keywords: Data Confidentiality · Separation Kernel · Information-flow Security

1 Introduction

The security aspects of Cloud Computing represent a major concern of both fundamental research and development, aiming at protecting data confidentiality and integrity of the applications running on the Cloud [21, 28]. Some of those applications are composed of large codebases that rely on third-party software subject to frequent updates or short time-to-market. This makes it difficult to verify or certificate the security assurances of the released software, also exposing it to bugs that lead to exploitable vulnerabilities. Moreover, application operators exploiting the Cloud rely on cloud providers. Cloud providers deliver hardware and software infrastructure capabilities maintaining high privileges on the access to the infrastructure [8]. From the data security point of view, cloud providers cannot be considered fully reliable, e.g. a malicious insider could abuse its access rights to steal secret information [29].

To protect applications data we consider the following scenario to place a multi-component application on the Cloud. On one hand, we want to isolate the components of an application in order to avoid leak of sensitive data by exploiting vulnerabilities of some components. On the other, we want to prevent

access to the data of the application by unreliable cloud providers. To fulfill those requirements, application developers can exploit Trusted Execution Environments (TEEs) [1, 2, 7] to isolate the components of applications in *domains* allowing the data flow only using explicit communication channels and to elude the privileges of the hardware platform providers. TEEs provide in the Cloud the same memory and register isolation that is given by the Separation Kernel (SK) technology [23]. In order to use these technologies, developers must decide which components should be grouped in the same domain, i.e. how to find a *partitioning* of the application to avoid data leaks. Given that the problem of how to partition an application is not dependent on the hardware that isolates the domains, in the rest of our discussion we refer to SKs as the supporting technology that comprehends TEEs and other similar technologies.

In this article, we tackle the partitioning problem employing information-flow security [24] methodologies (*i*) to understand whether the software components leak sensitive data outside the SK and (*ii*) to partition the application in order to avoid data leaks between components hosted on the same SK domain. We call *partitionable* those applications that do not leak data outside the SK, and we aim at finding a *minimal eligible partitioning* of such applications, namely a partitioning with the minimum number of domains that avoid data leaks.

Our contribution consists of:

- (a) The definition of a declarative model to represent a multi-component application exploiting information-flow security to discriminate data confidentiality and to check whether the components manage their data without leaks,
- (b) The (formal) definition of partitionable application and of the eligible partitioning problem, and
- (c) A prototype of the above, SKnife³, implemented in Prolog, to determine the minimal eligible partitioning of a partitionable application and an extension of the prototype capable of determining relaxed constraints of a non-partitionable application in order to be able to find the eligible partitioning.

The rest of this article is organised as follows. After giving background information and describing our motivating example (Sect. 2), we illustrate SKnife methodology and implementation (Sect. 3), which is then used to solve and discuss the motivating example (Sect. 4). After discussing some closely related work (Sect. 5), we conclude by pointing to some directions for future work (Sect. 6).

2 Background and Motivating Example

In this section, we first introduce basic notions behind the methodologies employed in our work and our motivating example.

2.1 Background

Scenario. We consider the public Cloud setting, where the cloud providers deliver hardware and software infrastructure to their customers. The applications are multi-components. Each software component of the application has a set of

³ Open-sourced and freely available at: <https://github.com/di-unipi-socc/sk>.

security-relevant characteristics, properties or software dependencies of the component, e.g. the use of a non verified third-party library. Those characteristics determine the degree of trust of a component in order to establish if the component can manage its data without leaks. The usage of software with a low degree of trust can lead to attacks that compromise the confidentiality of the data of an application. Hence, application developers need mechanisms *(i)* to identify how reliable software manage sensitive data, and *(ii)* to securely isolate components in order to avoid data leaks from unreliable software and *(iii)* to be protected by unreliable cloud providers.

Separation Kernels. An SK is a security kernel that creates an environment that is indistinguishable from a distributed system, where information can only flow from one isolated machine to another along explicit communication channels [23]. SKs are hardware or software mechanisms that partition available resources in isolated *domains* (or partitions), mediate the information flow between them, and protect all the resources from unauthorized accesses. In the Cloud setting, hardware-based technologies (TEEs) [1, 2, 7] are emerging to allow cloud providers customers to create isolated memory domains for code and data, which are also not accessible by the privileged software that is controlled by the cloud provider. Our approach is focused on the data separation given by SKs, we do not consider other sophisticated mechanisms offered, like the separation of resources and the timed scheduling of domains.

Information-Flow Security. In information-flow security, labels are assigned to variables of a program to follow its data flow in order to verify desired properties (e.g. non-interference [25]) and avoid covert channels. Labels are ordered in a security lattice to represent the relation of the labels from the highest ones (e.g. top secret) to the lowest ones (e.g. public data). Our goal is to prevent that data with a certain label (e.g. high) do not reach an external component that has a lower label (e.g. low). A violation of this security property indicates a data leakage.

Threat model. Our goal is to protect the *data confidentiality* of multi-component Cloud applications. Vulnerable software components can be attacked by external attackers and by unreliable cloud providers that can exploit their superuser privileges on the infrastructure. Application developers are assumed trusted, the information they give about the application to protect is considered reliable. Our Trusted Computing Base (TCB) leverages on the SK technology to isolate the software components of the application in separate domains, guaranteeing that the information flows only along the explicit communication lines given by the application developers and avoiding other side channels. This model is consistent with threat model of several Trusted Execution Environments (e.g. [26]).

2.2 Motivating Example

We consider a Cloud centralised Internet of Things (IoT) system that collects data sampled by the sensors and send to the Cloud application, where data is stored and used to decide which commands to issue to the actuators. The users of this application can make requests on the status of the devices and can configure remotely the application. Nowadays, those kinds of applications are

well established in the home automation field [5], services-devices composition [6] and platforms offered by Cloud providers [3, 4].

The architecture of the example application is depicted in Fig. 1. We consider

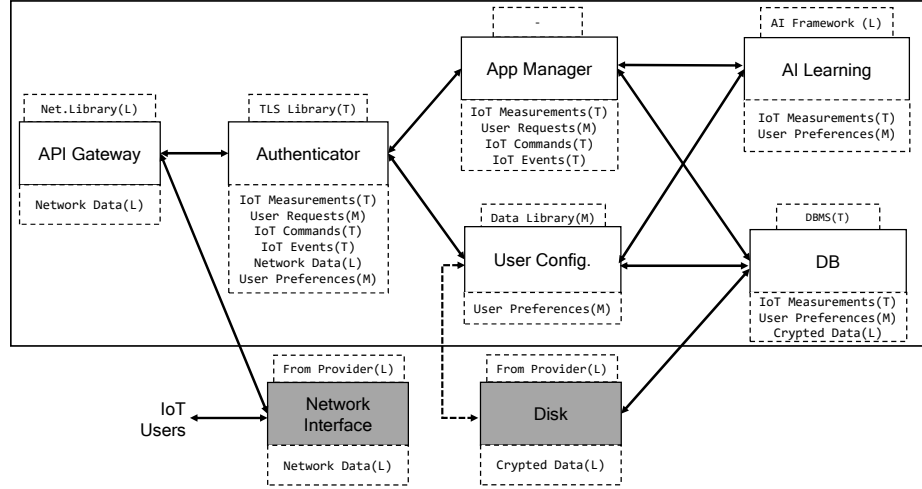


Fig. 1: Application architecture.

six software components and two hardware components – depicted in grey – that are used by the application.

All the inbound communication passes through the **Network Interface** and is received by the **API Gateway**. **Authenticator** decrypts and authenticates the inbound data and forwards it to the intended recipient. Application users can send General requests and configuration requests. The former are requests of explicit actuation or data previously sampled and are delivered to the **App Manager**, which is the main component of the application that implement the business logic. The latter are requests of reading or updating the current application configuration and are delivered to the **User Configuration** component, which manages the configuration of the application. The IoT devices send either sampled data or events, which are dispatched to the **App Manager** component. The outbound communication consists of responses to the users based on their requests or IoT commands from the **App Manager** toward the IoT devices. To store the relevant data of the application – IoT Measurement and User Preferences – the application relies on the component **DB**, a database that is the only one connected to the **Disk**. Finally, **AI Learning** is a machine learning module that uses IoT Measurement and User Preferences to perform predictions and support the decision making of the **App Manager**.

Each component has explicit links to other components, its own data – depicted in the lower boxes of the components –, and its relevant characteristics – depicted in the upper boxes of the components. For instance the component **AI Learning** has data IoT Measurement and User Preference, its relevant characteristic is **AI Framework** and it is linked to **App Manager** and **User Configuration**. The characteristics are properties, third-party libraries etc., all things that impact on the

trust of the components. For instance, the `Disk` is owned by the Cloud provider that in our setting make the component unreliable. The measure of the trust level of components is mandatory to determine if they can manage their data in order to avoid leaks. For instance, the `API Gateway` must be able to manage its data to avoid the leak of such data toward the `Network` interface. The dotted arrow between `User Config` and `Disk` represent a link consisting of an alteration of the application architecture. Fig. 1 represents two different application architectures with only that link as a difference. The base application – identified as `iotApp1` – does not have the dotted link. The modified application – identified as `iotApp2` – has the dotted link. We will use those two slightly different architectures in Sect. 4 when discussing the partitioning of the two applications.

This application results in a large codebase, which includes the operating system, communication stacks, AI frameworks, etc. It may also require frequent updates or short time-to-market. These factors make it hard to verify or certificate the security of the released software. Here, we propose to assign labels to the relevant characteristics of the application to determine the level of trust of software components. We also use the labels for the data of the application in order to establish a direct relationship between data and the trust of the software components. Here we adopt the security lattice of Fig. 2, modelling the labels pertaining to certain sensitive data from **top** (i.e. secret) to **low** (i.e. public), and trusted characteristics from **top** (i.e. highest trust) to **low** (i.e. not reliable). The labelling is represented in Fig. 1 by the letters between brackets placed near the data and characteristic names, where **L** stands for **low**, **M** stands for **medium**, and **T** stands for **top**.

A component having characteristics considered unreliable by the application developer is not able to manage secret data. This could cause a leak of its data toward the directly connected components or toward the software components hosted in the same isolation environment, i.e. container, virtual machine, SK domains. For instance, if the DBMS used by `DB` is not reliable – either because it is malicious or because it has vulnerabilities – the data of `DB` can leak toward the `Disk`, component owned by the Cloud Provider. Furthermore, if `DB` is isolated in the same SK domain of `API Gateway` (assuming a unreliable `Net Library`), the leak of data could flow from the `DB` to the `Network Interface` through the `API Gateway`.

We emphasise again that we aim at protecting data confidentiality of applications placed on the Cloud finding *eligible partitionings*, i.e. grouping the software components in non-empty subsets that allow placing the application in SK domains in such a way that the data and trust of the components are homogeneous in every domain, avoiding that less trusted components share the environment with components that manage sensitive data. For instance, we already discussed that `AI Framework` is a library of `AI Learning` considered not reliable, it may contain malicious code or its vulnerabilities may be exploited by



Fig. 2: Example security lattice.

an external attacker. Placing all the software components in the same environment (e.g. domain or virtual machine) exposes the data of the application to be read by AI Framework and sent outside the environment. Partitioning the application components to isolate their data and exploiting SKs isolation mitigates those kinds of threats.

Moreover, we want to reduce the number of domains used by the eligible partitioning to reduce the SK overhead that can heavily impact the application performance. For example, switching domain during the execution has a cost in terms of time that is influenced by the sanitising of used resources and by the domains scheduling algorithm of the SK. This kind of situation discourages the creation of a high number of domains. For this reason, we aim at finding the *minimal partitioning*, the eligible partitioning that creates the lowest number of domains.

3 Methodology and Prototype

This section describes the methodology which allows us to determine an *eligible partitioning* of an application onto an SK. We also discuss the Prolog⁴ prototype SKnife supported by the methodology through simple examples excerpted from Sect. 2.2.

3.1 Modelling applications and labelling

Application developers model their application as in

```
application(AppId, ListOfHardware, ListOfSoftware).
```

where AppId is the application identifier, ListOfHardware is the list of hardware components interacting with the application, and ListOfSoftware is the list of software components to place on the SK.

Example. The `iotApp1` application of our motivating example is declared as

```
application(iotApp1, [network, disk],[UserConfig, appManager,
    authenticator, aiLearning, apiGateway, db]).
```

Software and hardware components are declared as in

```
software(SwId, ListOfData, ListOfCharacteristics, [LinkedHW, LinkedSW] ).
hardware(HwId, ListOfData, ListOfCharacteristics, [LinkedHW, LinkedSW]).
```

where SwId and HwId are the unique identifiers of each component, ListOfData is the list of names of the data managed by the component, ListOfCharacteristics is the list of names of the component characteristics, LinkedHW is the list of linked hardware components and LinkedSW is the list of linked software components.

Example. The `db` and `disk` components are declared as

⁴ A Prolog program is a finite set of *clauses* of the form: `a :- b1, ..., bn.` stating that `a` holds when `b1 ∧ ... ∧ bn` holds, where $n \geq 0$ and `a`, `b1`, ..., `bn` are atomic literals. Clauses with empty condition are also called *facts*. Prolog variables begin with upper-case letters, lists are denoted by square brackets, and negation by `\+`.

```
software(db, [IotMeasurements, userPreferences, crypteData],
         [dbms], ([disk],[userConfig, managementLogic])).
hardware(disk, [crypteData], [fromProvider],[[],[db]]).
```

Application developers must also declare a security lattice formed by ordered labels and they have to label the relevant data of the application and the relevant characteristics of the components. The higher is the label of data, the higher is the secrecy of the data. Similarly, the higher is the label of a characteristic, the higher is the trust of the characteristic. We call the labels assigned to data *secrecy labels* and the labels assigned to characteristics *trust labels*.

Every data and characteristic can be labelled using

```
tag(Name, Label).
```

where *Name* is the name of the data or characteristic to be labelled and *Label* is the assigned label. Obviously, the labels must be part of the the lattice.

Example. The security lattice of Fig. 2 and the label of the data and the characteristics of the Disk are declared as

```
tag(crypteData, low).
tag(fromProvider, low).
```

which represents *crypteData* data with low secrecy label and *fromProvider* characteristic with low trust label.

3.2 Eligible Partitioning

Our methodology and SKnife as well assign to every component a pair of labels, one indicating its secrecy level and one indicating its trust level. A component is *trusted* if its trust label is greater or equal ⁵ than its secrecy label, otherwise, it is considered *untrusted*. A trusted component is able to manage its data without the risk of leaking them. The labelling of a component is performed by the predicate *labelC/3* of Fig. 3, using the lists of data and characteristics of the component. The secrecy label is determined by the highest label of its data in

```
1 labelC(Data, Characteristics, (MaxType,CharactSecType)):-
2   dataLabel(Data,Labels),
3   highestType(Labels,MaxType),
4   characteristicsLabel(Characteristics, ListOfCharactTypes),
5   lowestType(ListOfCharactTypes, CharactSecType).
```

Fig. 3: The *labelC/3* predicate.

order to consider the most critical data managed by the component. The trust label is determined by the lowest label of its characteristics because the worst characteristic could compromise the trust of the component, e.g. a component using a simple logging library and a certified encryption software could be endangered by a bug in the former one. A component without relevant characteristics is considered reliable and its trust label is the highest one of the security lattice.

⁵ All the comparisons between labels are based on the ordering of the security lattice.

We choose this level of granularity (i.e., the developer labels data and characteristics instead of directly labeling the components) to have a better insight of the application and to allow advanced features as the one that will be described in Sect. 3.3.

Untrusted components can leak their data to directly linked components. If such components have a trust label lower than the leaked data they can propagate the leakage through their links. If such data reach a hardware component, then an *external leak* occurs. An external leak is a path from an untrusted software component to a hardware component where all the components of the path have the trust label lower than the secrecy label of the first software component of the path. The presence of such paths indicates the potential for a data leakage from an untrusted component toward the outside of the SK that is not avoidable by the partitioning.

We say that an application is called *partitionable* if there is no leakage outside the SK: i.e., all its hardware components are trusted and all its untrusted software components do not have an external leak of data.

The predicate `hardwareOk/1` of Fig. 4 checks that all the hardware components of the application are trusted, which avoids hardware attacks that cannot be countered by the SK partitioning. The predicate recursively scans the list of hardware components to check their labelling. Initially, information of a single component is retrieved (line 2), then the labelling of the hardware component is determined (line 3). The predicate checks for the component trustability (line 4), where `gte/2` checks if the trust is greater or equal than the secrecy. Finally, `hardwareOk/1` recurs on the rest of the list (line 5) until it is empty (line 6).

```

1 hardwareOk([H|Hs]) :-
2   hardware(H, Data, Characteristics,_),
3   labelC(Data, Characteristics, (TData,TChar)),
4   gte(TChar,TData),
5   hardwareOk(Hs).
6 hardwareOk([]).
```

Fig. 4: The `hardwareOk/1` predicate.

The predicate `softwareOk/1` of Fig. 5 checks that no software components (line 2) that is untrusted (line 3) has an external leak toward an untrusted hardware component (line 4).

```

1 softwareOk(LabelledSoftware):-
2   \+ (member((Sw,TData,TChar), LabelledSoftware),
3       lt(TChar,TData),
4       externalLeak([Sw], [], TData, LabelledSoftware)).
```

Fig. 5: The `softwareOk/1` predicate.

The software components of a partitionable application can be split and placed on SK domains. A domain is a triple $(DTData, DTChar, HostedSw)$ where $DTData$ is the secrecy label of the domain, $DTChar$ is the trust label of the domain, and $HostedSw$ is the list of the software components hosted by the domain. Inside

a domain, software components share the same environment. To avoid placing components in an environment containing data that they are not able to manage, a domain must be *data consistent*:

$$\begin{aligned} \forall \text{software}(\text{Sw}, \text{Data}, \text{Characteristics}, _) \in \text{HostedSw} : \\ \text{labelC}(\text{Data}, \text{Characteristics}, (\text{CTData}, _)) \rightarrow \text{PTData} = \text{CTData} \end{aligned}$$

meaning that in a domain there is no software component with a secrecy label different from the domain secrecy label, i.e. all the software components hosted by a domain have the same secrecy label of the domain. This property avoids that a software component is placed in a domain that contains data more sensitive than the ones the component is supposed to deal with.

Another aspect to consider is that untrusted components bring out the risk to leak sensitive data to other components of the domain or to linked components outside the domain. In order to isolate such components, domains must be *reliable*:

$$\begin{aligned} \forall \text{software}(\text{Sw}, \text{Data}, \text{Characteristics}, _) \in \text{HostedSw} : \\ \text{labelC}(\text{Data}, \text{Characteristics}, (\text{CTData}, \text{CTChar})) \rightarrow \text{CTData} \geq \text{CTChar} \\ \vee \\ \forall \text{software}(\text{Sw}, \text{Data}, \text{Characteristics}, _) \in \text{HostedSw} : \\ \text{labelC}(\text{Data}, \text{Characteristics}, (\text{CTData}, \text{CTChar})) \rightarrow \text{CTChar} = \text{PTChar} \end{aligned}$$

meaning that all the software components of a domain are either trusted or have the same trust label of the domain. Domains hosting only trustable software components are considered secure from data leaks. Every component can manage its data and can exchange it outside the domain without risk of leaks, according to the trust assigned by the developer. Untrusted components must be isolated strongly, they can share a domain only with other untrusted components having the same trust label, in order to have a homogeneous level of trust inside the domain and mitigate the danger of data leak.

Eligible partitionings split a partitionable applications into a set of data consistent and reliable domains. The top-level `sknife/2` (Fig. 6) finds the eligible partitioning of a partitionable application. After retrieving the application information (line 2), it basically performs two main steps, first it checks whether the application is partitionable (lines 3–5) and second it creates the set of data consistent and reliable domains splitting the software component across them (line 6), starting from an empty partitioning (`[]` of line 6).

The `partitioning/3` predicate is listed in Fig. 7 and it has the task to split labelled software components placing them in data consistent and reliable domains. The predicate recursively scans the list of labelled software components (first argument) to place every component starting from a partitioning (second argument) that will be updated in the resulting partitioning (third argument). The domains of the resulting partitioning are data consistent and reliable by construction. Every software component is placed in a domain with the same

```

1 sKnife(AppId, EligiblePartitioning) :-
2   application(AppId, Hardware, Software),
3   hardwareOK(Hardware),
4   softwareLabel(Software, LabelledSoftware),
5   softwareOk(LabelledSoftware),
6   partitioning(LabelledSoftware, [], EligiblePartitioning).

```

Fig. 6: The sKnife/2 predicate.

secrecy label to satisfy the data consistency of the domain. Trusted components are placed together in domains with the trust label named `safe`, indicating that all the hosted components are trusted. Untrusted components are placed in the domain with the same trust label, in order to create reliable domains. If the domain needed by a component is not in the starting partitioning, it is created with correct labels and added to the partitioning. `partitioning/3` has two main

```

1 partitioning([(S,TData,TChar)|Ss], Partitioning, NewPartitioning) :-
2   partitionCharLabel(TChar,TData,TCD),
3   select( ((TData,TCD), Ds), Partitioning, TmpPartitioning),
4   DNew = ( (TData,TCD), [S|Ds]),
5   partitioning(Ss, [DNew|TmpPartitioning], NewPartitioning).
6 partitioning([(S,TData,TChar)|Ss], Partitioning, NewPartitioning) :-
7   partitionCharLabel(TChar,TData,TCD),
8   \+ member( ((TData,TCD), _), Partitioning),
9   DNew = ( (TData,TCD), [S]),
10  partitioning(Ss, [DNew|Partitioning], NewPartitioning).
11 partitioning([],P,P).

```

Fig. 7: The partitioning/3 predicate.

clauses (lines 1 and 6) plus the empty software list case that leaves the partitioning unmodified (line 11). The first case describes the situation in which a software element can be placed on a domain already created. After determining the labelling of the hosting domain (line 2), the library predicate `select/3` checks if such domain is already created in the partitioning (line 3) and extract it. Then, an updated domain is created by adding the current software component (line 4). Finally, `partitioning/3` recurs on the rest of the software list giving as starting partitioning the old partitioning with the updated domain (`[DNew|TmpPartitioning]` of line 5).

The second clause of the predicate (line 6) describes the situation in which the domain that has to host the software component is not already in the input partitioning. The initial step to determine the hosting domain labelling is the same as the previous clause (line 7). Then, there is an explicit check that such domain is not already in the partitioning (line 8). At this point, the new domain is created (line 9) and it is included in the partitioning during the recursive call (`[DNew | Partitioning]` of line 10).

As mentioned in Sect. 2.2 it is important to reduce the number of domains of a partitioning as much as possible. We emphasise that `SKnife` main predicate

outputs as unique solution the minimal eligible partitioning of the application, if it exists⁶.

3.3 Labelling suggestions

Not all the existing applications are partitionable, precluding the possibility to find an eligible partitioning. To assist application developers in those situations we added the feature to suggest *relaxed labellings* on data or characteristics of the applications. Those suggestions reduce the secrecy or increase the trustability of components relaxing the labelling of an application in order to find an eligible partitioning. This feature is intended to help the review of an application preventing the risk of leaking the confidentiality of data. Because of space limitations, we do not include code snippets of this version of SKnife⁷.

The basic version of SKnife either finds the minimal partitioning or fails if there is a risk of data leak given by untrusted components. To support the suggestions feature, we suitably modified SKnife to individuate the source of a failure and to retry the partitioning after relaxing the labelling of such a source.

As aforementioned, the predicates that check if an application is partitionable are `hardwareOk/1` and `softwareOk/1`. Those predicates are modified in order to return every single component responsible for a failure when the application is not partitionable. Then, the labelling of those components are *relaxed*, i.e. the data labels are decreased or the characteristics labels are increased. After that, the application is labelled with the relaxed labels and a new search for an eligible partitioning of the application starts. This retry mechanism is repeated until both the check predicates do not find a failure. The search eventually finds suggestions for an eligible partitioning. In the worst case scenario, the data labels will be relaxed to the lowest label of the lattice or the characteristics label will be relaxed to the highest label of the lattice.

The main predicate of the refinement of SKnife is `sknife/3`. It has as the first argument the application identifier, as in the base version. The second argument is the list of relaxed labelling, pairs of data/characteristics names and the new relaxed label. The third argument is the eligible partitioning found with the relaxed labelling. Note that this predicate does not compute a unique solution. Indeed, for every query different relaxed labelling with the relative eligible partitioning are computed. This allows SKnife to give to the developer different suggestions.

4 Motivating Example Revisited

In this section, we will solve the partitioning problem of the architecture `iotApp1` of the motivation example⁸ of Sect. 2.2 given a suitable set of labels for every data and characteristic. Then, we will consider the slightly different `iotApp2`

⁶ The proofs indicating that the minimal partitioning is unique and that SKnife finds the minimal partitioning are presented in Appendix

⁷ Full code of the prototype extension at <https://github.com/di-unipi-socc/sk/blob/main/Examples/CloudExample/skplacerRecommend.pl>

⁸ Full example code at <https://github.com/di-unipi-socc/sk/tree/main/Examples/CloudExample>

architecture showing that it is not partitionable and we will apply the *relaxed labelling* feature of SKnife. In both cases the application architecture, the (software and hardware) components and the security lattice can be expressed as per the modelling of Sect. 3. Data and characteristics are labelled as indicated by the letters between brackets in Fig. 1 using the `tag/2` predicate.

4.1 Finding the minimal partitioning

To find the minimal partitioning of `iotApp1` we can simply query the `sKnife/2` predicate as `sKnife(iotApp1, Partitioning)`. Initially, SKnife labels all the application components as depicted in Fig. 8, where is assigned a pair of label for each component, one for data (the τ above the component) and top for its characteristics (the τ below the component). Then, SKnife checks if the application is partitionable. The application is partitionable because the hardware compo-

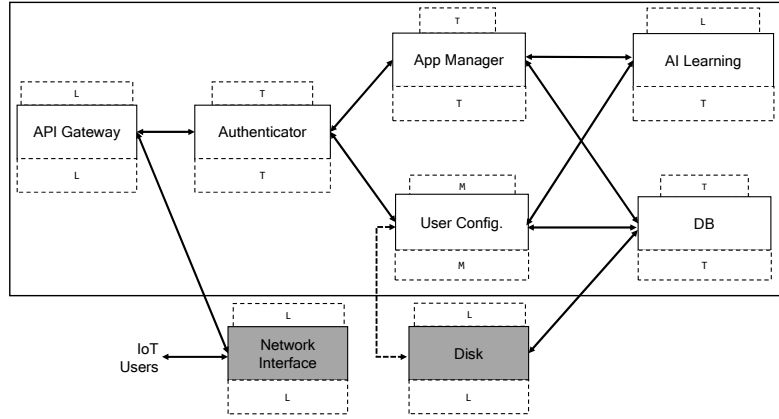


Fig. 8: Labelling of application components.

nents manage only low data and do not exist a path from AI Learning (the only untrusted component) to the hardware components that can leak top or medium data.

Fig 9 summarises the obtained result. The eligible partitioning is composed of 4 domains, 3 with trusted components (D1–D3) and 1 with an untrusted component (D4). It is a minimal partitioning because we have at least 3 software components with different secrecy labels and only 1 untrusted component and it is not possible to divide those components in less than 4 domains that are data consistent and reliable.

As aforementioned, SKnife outputs only a solution because the minimal eligible partitioning is unique, i.e. do not exist a partitioning of the application with a fewer or equal number of data consistent and reliable domains.

4.2 Relaxing the labelling

To show the *relaxing labelling* feature we consider the architecture `iotApp2` with the additional link between the User Configuration and the Disk. This link creates a path from the AI Learning to the Disk that can leak the top data IoT

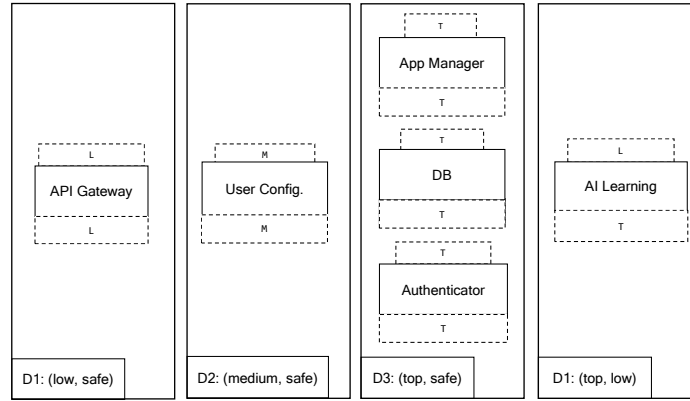


Fig. 9: Minimal eligible partitioning.

measurements and medium data User Preferences, making the application non partitionable. This happens because AI Learning is an untrusted component and can leak its data via its explicit links. The linked component User Configurations has trust label medium and it is not reliable to manage top data, thus IoT measurements can be leaked to the Disk with the newly added link. In this situation, it is not possible to found an eligible partitioning.

To use the *relaxing labelling* feature on application `iotApp2` we can query `sKnife/3` predicate as `sKnife(iotApp2, S, Partitioning)`. As expected, the check performed by `softwareOk/2` finds a path with an external leak and individuate all the components involved in the path, triggering the retry behaviour explained in Sect. 3.3.

For the sake of clarity we show only the results to the query for the suggestion variable `S`, avoiding to display the eligible partitioning generated by applying the suggestions. The obtained results are

```
S=[(iotMeasurements,low),(userPreferences,low)]; S=(aiFramework,top);
S=(dataLibrary,top);                             S=(iotMeasurements,medium);
S=(fromProvider,top);                             S=(iotMeasurements,low).
```

For this specific situation, we can see that the solution is either reduce the security of IoT Measurements and User Preferences managed by AI Learning, cutting the path toward the Disk. When is analysed AI Learning the suggestion is to label low the data. When is analysed the second component of the path – User Configuration –, the suggestion is to reduce IoT Measurements to medium. Finally, when is analysed the last component of the path – Disk –, the suggestion is to reduce IoT Measurements to low. The alternatives increase the trust of each component of the path to cut the possible leak, increasing the characteristics AI Framework, Data Library and From Provider to top.

Those suggestions can support application developers to change the labelling if for instance the secrecy of IoT Measurements can be reduced. Otherwise, the suggestions could lead to changing the characteristics involved in the leak, for instance using a more reliable Data library for the component User Configuration.

Analysing each component of a path for every query can output overlapping solutions. For instance, `IoT Measurements reduce to low` appear two times. This can be avoided easily by searching all the suggestions at once and filtering the results.

5 Related Work

To the best of our knowledge, there are currently no proposals that employ information-flow security to place applications on SKs. Some approaches use information-flow security to address problems that are complementary to our techniques, for example checking the correct labelling of software or monitoring inputs and storage accesses by the applications. Elsayed and Zulkernine [15] propose a framework to deliver Information-Flow-Control-as-a-Service (IFCaaS) in order to protect confidentiality and integrity of the information flow in a SaaS application. The framework works as a trusted party that creates a call graph of an application from the source code and applies information-flow security based on dependence graphs to detect violation of the non-interference policy. At Function-as-a-Service (FaaS) level, Alpernas et al. [9] present an approach for dynamic Information-flow control monitoring the inputs of serverless functions to tag them with suitable security labels, in order to check access to data storage and communication channels to prevent leaks of data managed by the functions. Similarly, Datta et al. [14] propose to monitor serverless functions by starting to learn the information flow of an application, showing the detected flows to the developers and enforcing the selected ones. In the Cloud-IoT continuum scenario, our previous work [12] exploits information-flow security to place FaaS orchestrations on Fog infrastructures. Functions are labelled with security types according to input received and infrastructure nodes are labelled according to user-defined security policies. The placements are considered eligible if every node involved have the security type greater or equal than the security type of all the hosted serverless functions. Developers assign a level of trust to infrastructure providers that concurs to rank the eligible placements. Differently from SKnife, all those proposals but [15] consider the cloud provider reliable, conflicting with the threat model we introduced in Sect. 2. Recently, few proposals have leveraged on information flow analyses to enforce data security in cloud applications when the cloud provider is untrusted. For example, Oak et al. [22] have extended Java with information flow annotations that allow to verify if partitioning an application into components that run inside and outside a SGX enclave violate confidentiality security policies. In this proposal partitioning is decided by the programmer.

Other approaches aim at verifying the data separation and the data flow of SKs [10, 13, 18, 27]. SKnife does not require special assumption on the SK and can be employed on all those SKs.

Similarly to SKnife, declarative techniques have been employed to resolve different Cloud-related problems. There are proposals to manage Cloud resources (e.g. [20]), to improve network usage (e.g. [19]), to assess the security and trust levels of different application placements (e.g. [16]), and to securely place VNF chains and steer traffic across them (e.g. [17]). To the best of our knowledge,

there are currently no declarative proposals tackling our considered partitioning problem.

6 Concluding Remarks

This article introduced a declarative methodology and its prototype, SKnife, to protect the data confidentiality of Cloud applications from external attackers and unreliable cloud providers. Our approach exploits the Separation Kernel technology as Trusted Computing Base. It also employs information-flow security mechanisms to determine the eligible partitioning of a partitionable application to assist the placement of its software components in the Separation Kernel domains. To support the application developers, we extended SKnife with a feature that allows finding the eligible partitioning of non-partitionable applications relaxing the information-flow constraints given by application developers.

Our methodology requires manual labelling of data and characteristics of the application components. This limitation can be tackled using monitoring or automatic techniques to track information flow [11, 14]. At the current stage, the flow of information is static, the data never change its labelling after the developer declaration. In our future work, we plan to support dynamic information flow, based on the application input and the relations between the software components. Moreover, we plan to support the labelling modification of an already partitioned application, correcting only the partitioning of the involved components, avoiding restarting from scratch the partitioning process. For instance, when a bug in a library used by a component is discovered, the label of the library can be reduced and the partitioning of the application can be changed accordingly. Another interesting direction is to extend our methodology to support data integrity. In addition to tackling all aforementioned points, we also plan to apply our methodology to different types of applications and to evaluate the effectiveness and scalability of our approach with experimental results. Finally, we plan to add a second feature to allow eligible partitioning of non-partitionable applications. We intend to use software engineering techniques to suggest modification of the application architecture in order to correct data leakage without changing the overall application behaviour.

Appendix

Unique solution proof

Here we show how is calculated the bound on the minimum number of domains.

Lemma 1 (Minimum partitioning). *The minimum number of domains p of a secure placement is the same as all the labellings l of the software components of an application. The number p can be bound as:*

$$p = l, 1 \leq p \leq \frac{L(L+1)}{2}$$

where L is the number of security types of the security lattice and l is the number of secrecy labellings plus the number of possible labelling of untrusted components.

Proof. To find the minimum number of domains in a secure placement we consider separately two cases for the domains. The first case considers only domains hosting trusted components, represented as:

$$(DataLabel, safe)$$

indicating domains hosting trusted software components with data label equal to *DataLabel*. The characteristic label *safe* is a placeholder for the trust of those domains. It is easy to see that the minimum number of domain p_1 in this case is

$$p_1 = t, 1 \leq t \leq L$$

where t is the number of trusted software components with different secrecy labels. Considering that a domain must be *data consistent*, we have to separate components with different data labels. If we have t different data labels, we need at least t domains to separate them. The lower bound is 1 because we assume that the application has at least one trusted software component. The upper bound is the maximum number of data labels that is L by definition.

The other kind of domains are the ones that host untrusted components, that can be represented as:

$$(DataLabel, CharLabel) :$$

indicating domains hosting untrusted software components with data label equal to *DataLabel* and characteristic equal to *CharLabel*, with *CharLabel* less than *DataLabel*. To make the domains *reliable* we have to isolate untrusted components with same pair of labels. In this case, the minimum number of domain p_2 is

$$p_2 = u, 0 \leq u \leq \frac{L(L-1)}{2}$$

where u is the number of untrusted components with different pairs of labels. Here the lower bound is 0, indicating that is possible to not have untrusted components. The upper bound indicates all the possible configurations of untrusted labelling. To count the configuration number we start considering the highest security label. This label has $L-1$ labels lower than itself, thus there are $L-1$ possible untrusted configurations with the highest data label. The second-highest security label has $L-2$ possible configurations when it is the data label. We can continue counting until the lowest security label that has 0 possible configurations has data label because there are no labels lower than itself. Summing all together, the number of possible untrusted configuration c is

$$c = (L-1) + (L-2) + \dots + (L-(L-1)) + (L-L) = (L-1) + (L-2) + \dots + 1 + 0$$

that is the sum from 1 to $L-1$, that can be expressed as

$$c = \frac{L(L-1)}{2}$$

The overall minimum number of domains p is given by the sum of p_1 and p_2 . The number of different labellings l is the sum of the labellings of the trusted

components t and the labellings of the untrusted componets u . Putting all together we have

$$p_1 + p_2 = t + u, 1 \leq (t + u) \leq L + \frac{L(L-1)}{2} =$$

$$p = l, 1 \leq l \leq \frac{L(L+1)}{2}$$

Minimal partitioning proof

Proof that **SKnife** finds the minimal partitioning of an application.

Proof. As aforementioned, **SKnife** main predicate can fail – answering false – if the application is not partitionable. This is checked by `hardwareOk/1` of Fig. 4 and `softwareOk/2` of Fig. 5. The minimum number of domains is not fixed for all possible applications, because it depends on the number of security types of the lattice and on the labelling of the application to be placed.

To prove that the result of **SKnife** is the minimal partitioning is enough to prove that the predicate `partitioning/3` of Fig. 7 creates the minimal partitioning. A partitioning is a set of domains, here we describe a domain as a triple

$$(d, c, s)$$

where d is the secrecy label, c is the trust label and s is the list of software components hosted by the domain. The predicate `partitioning/3` implements a function

$$partitioning : S \times P \rightarrow P$$

where S is the software components set and P is the partitioning set. The function `partitioning` has the following invariant:

$$\forall (d, safe, s) \in P : \forall sw \in s, \lambda(sw) = (d, c), c \geq d, \phi(sw) = (d, safe) \quad (1)$$

$$\forall (d, c, s) \in P, c \neq safe : \forall sw \in s, \lambda(sw) = (d, c), c \leq d, \phi(sw) = (d, c) \quad (2)$$

$$\forall (d, c, s), (d', c', s') \in P : d = d' \wedge c = c' \implies s = s' \quad (3)$$

$$\forall (d, c, s) \in P : s \neq \emptyset \quad (4)$$

where λ applied to a software component gives its labelling and ϕ gives the labelling of the domain needed by the software component. Equations 1 and 2 indicate the labelling of software components and domains to satisfy the well-formed domain properties. Equation 3 indicates that no domains with the same labelling are admitted. Equation 4 indicates that do not exist empty domains.

We can proof by induction that the function `partitioning` creates the minimal partitioning.

Base Case

$$\text{partitioning}([], P) = P$$

an empty list of software components do not modify the partitioning.

Induction Step

$$P_i = \text{partitioning}([sw_i : sws], P_{i-1})$$

$$P_i = \begin{cases} P_{i-1} \setminus \{(d, c, s)\} \cup \{(d, c, \{s \cup sw_i\})\} & \text{if } \phi(sw_i) = (d, c) \in P_{i-1} \\ P_{i-1} \cup \{(d, c, \{sw_i\})\} & \text{otherwise} \end{cases}$$

We assume, by inductive hypothesis, that the invariant holds up to the i -th step, i.e. for the domain of components $[sw_1, \dots, sw_i]$. It is easy to see that at step $(i + 1)$ the current sw_{i+1} is suitably placed into an existing domain (1st clause of `partitioning/3`) or inserted into a new one (2nd clause of `partitioning/3`) by following the rule above. Assume that after this step, the invariant does not hold, i.e. that the found domain is non-minimal. Then, as sw_{i+1} was correctly sorted, it means that some services in $[sw_1, \dots, sw_i]$ where not. This implies that the invariant did not hold in some of the previous steps, which contradicts the inductive hypothesis.

Basically, the *base case* and the first case of the *induction step* do not modify the number of domains of the input partitioning. The second case of the *induction step* increases by one the number of partitioning if the domain to host a software component is not in the partitioning.

References

1. AMD Secure Encrypted Virtualization (SEV). <https://developer.amd.com/sev/>, Last Accessed: Nov. 2021
2. Arm Confidential Compute Architecture (CCA). <https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture>, Last Accessed: Nov. 2021
3. AWS IoT Greengrass. <https://aws.amazon.com/greengrass/>, Last Accessed: Nov. 2021
4. Azure IoT Edge. <https://azure.microsoft.com/services/iot-edge/>, Last Accessed: Nov. 2021
5. Home Assistant. <https://www.home-assistant.io/>, Last Accessed: Nov. 2021
6. IFTTT. <https://ifttt.com/>, Last Accessed: Nov. 2021
7. Intel Trust Domain Extensions (TDX). <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html>, Last Accessed: Nov. 2021
8. Almorsy, M., Grundy, J.C., Müller, I.: An analysis of the cloud computing security problem. CoRR **abs/1609.01107** (2016)
9. Alpernas, K., Flanagan, C., Fouladi, S., Ryzhyk, L., Sagiv, M., Schmitz, T., Winstein, K.: Secure serverless computing using dynamic information flow control. OOPSLA **2**, 1–26 (2018)
10. Andronick, J.: From a proven correct microkernel to trustworthy large systems. In: FoVeOOS 2010. vol. 6528, pp. 1–9. Springer (2010)

11. Bastys, I., Balliu, M., Sabelfeld, A.: If this then what?: Controlling flows in iot apps. In: ACM SIGSAC CCS 2018. pp. 1102–1119 (2018)
12. Bocci, A., Forti, S., Ferrari, G.L., Brogi, A.: Placing faas in the fog, securely. In: ITASEC 2021. CEUR Workshop Proceedings, vol. 2940, pp. 166–179 (2021)
13. Dam, M., Guanciale, R., Khakpour, N., Nemati, H., Schwarz, O.: Formal verification of information flow security for a simple arm-based separation kernel. In: ACM SIGSAC 2013. pp. 223–234. ACM (2013)
14. Datta, P., Kumar, P., Morris, T., Grace, M., Rahmati, A., Bates, A.: Valve: Securing function workflows on serverless computing platforms. In: WWW. pp. 939–950 (2020)
15. Elsayed, M., Zulkernine, M.: Ifcaas: Information flow control as a service for cloud security. In: ARES 2016. pp. 211–216. IEEE Computer Society (2016)
16. Forti, S., Ferrari, G.L., Brogi, A.: Secure cloud-edge deployments, with trust. *Future Gener. Comput. Syst.* **102**, 775–788 (2020)
17. Forti, S., Paganelli, F., Brogi, A.: Probabilistic QoS-aware Placement of VNF Chains at the Edge. *Theory and Practice of Logic Programming* **22**(1), 1–36 (2022)
18. Heitmeyer, C.L., Archer, M., Leonard, E.I., McLean, J.D.: Formal specification and verification of data separation in a separation kernel for an embedded system. In: ACMCCS 2006. pp. 346–355. ACM (2006)
19. Hinrichs, T.L., Gude, N.S., Casado, M., Mitchell, J.C., Shenker, S.: Practical declarative network management. In: WREN. pp. 1–10 (2009)
20. Kadioglu, S., Colena, M., Sebbah, S.: Heterogeneous resource allocation in Cloud Management. In: NCA 2016. pp. 35–38 (2016)
21. Kaufman, L.M.: Data security in the world of cloud computing. *IEEE Secur. Priv.* **7**(4), 61–64 (2009)
22. Oak, A., Ahmadian, A.M., Balliu, M., Salvaneschi, G.: Language support for secure software development with enclaves. In: IEEE Computer Security Foundations Symposium (CSF 2021) (2021)
23. Rushby, J.M.: Design and verification of secure systems. In: Proceedings of the Eighth Symposium on Operating System Principles, SOSP 1981. pp. 12–21. ACM (1981)
24. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. *IEEE J. Sel. Areas Commun.* **21**(1), 5–19 (2003)
25. Sabelfeld, A., Sands, D.: A per model of secure information flow in sequential programs. *High. Order Symb. Comput.* **14**(1), 59–91 (2001)
26. Sahita, R., Caspi, D., Huntley, B., Scarlata, V., Chaikin, B., Chhabra, S., Aharon, A., Ouziel, I.: Security analysis of confidential-compute instruction set architecture for virtualized workloads. In: (SEED). pp. 121–131. IEEE (2021)
27. Sewell, T., Winwood, S., Gammie, P., Murray, T.C., Andronick, J., Klein, G.: sel4 enforces integrity. In: ITP 2011. vol. 6898, pp. 325–340. Springer (2011)
28. Shaikh, F.B., Haider, S.: Security threats in cloud computing. In: ICITST 2011. pp. 214–219. IEEE (2011)
29. Tianfield, H.: Security issues in cloud computing. In: IEEE SMC 2012. pp. 1082–1089 (2012)