International Journal of Information Security (2023) 22:713–721
https://doi.org/10.1007/s10207-022-00658-8

REGULAR CONTRIBUTION

Multi-cloud applications: data and code fragmentation for improved security

Rudolf Lovrenčić[1] · Dejan Škvorc[1]

Published online: 3 January 2023
© The Author(s), under exclusive licence to Springer-Verlag GmbH, DE 2023

Abstract

When deciding against outsourcing their data to the cloud, organizations often point to security as the primary reason. If cloud is not used as a passive storage only, but rather both the data and the code required for their processing are being outsourced, then the data privacy may get compromised in two ways: (i) in the storage if not being encrypted and (ii) during the processing through various execution-level attacks. Encrypting the data before outsourcing enhances their security while in the storage, but disables their processing in the cloud. On the other hand, if a cloud has the ability to decrypt the data before processing, then they remain vulnerable during the execution. In this paper, we present a paradigm for outsourcing both the data and the code to the cloud in a way that preserves data privacy, while still enabling their processing outside the organization. The paradigm leverages constraint-based data and code fragmentation and deploys these fragments to multiple independent computer clouds. We introduce several architectural patterns for secure computation in a multi-cloud environment, demonstrate the paradigm use, and examine introduced performance penalty on a simple application.

Keywords Distributed applications · Distributed databases · Cloud computing · Security and privacy

1 Introduction

Cloud computing enables on-demand access to a shared pool of computing resources over the network [1]. Acquisition and release of those resources requires minimal management effort from both the service provider and the user which results in great flexibility at a low cost.

Some of the biggest challenges in cloud adoption are related to trust since the users often feel like they are losing control over their data [2]. Numerous data breaches and security vulnerabilities [3,4] do not give users the confidence that the *cloud service provider* (CSP) will keep their data secure. High flexibility of a cloud service makes exhaustive and continuous security revisions expensive or intractable [5].

Recent trends show that the use of multiple cloud providers simultaneously is increasing to achieve higher service availability and reduce damage in the case of malicious insiders on a single CSP [6]. Such *multi-cloud* environments mitigate reliance on a single cloud provider. Initial research of multi-cloud techniques focused on achieving cloud interoperability and overcoming resource restrictions of a single service provider [7,8]. AlZain et al. [6] conclude that the combination of secure cloud architectures and cryptography offers a huge potential going forward.

In this paper, we present a paradigm for the development of secure multi-cloud applications that improves the data privacy both in the storage and during the computation. We rely on fragmentation rather than encryption since it enables higher security levels while preserving data availability—the data remain available in plaintext, but are fragmented and distributed among multiple CSPs. Figure 1 illustrates our approach. Assuming that a and b stored together in a single database reveal critical information (i.e., illness and the person's name), we fragment the database so that a and b may be stored on different CSPs. As a consequence of the data fragmentation, the application's computational logic is fragmented as well in order to avoid a point where both a and b are known together. In other words, the data fragmentation governs the fragmentation of the computational logic. Data and code fragments are then distributed among multiple CSPs. In its fragmented form, the result of the monolithic

✉ Rudolf Lovrenčić
rudolf.lovrencic@fer.hr

Dejan Škvorc
dejan.skvorc@fer.hr

[1] University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia

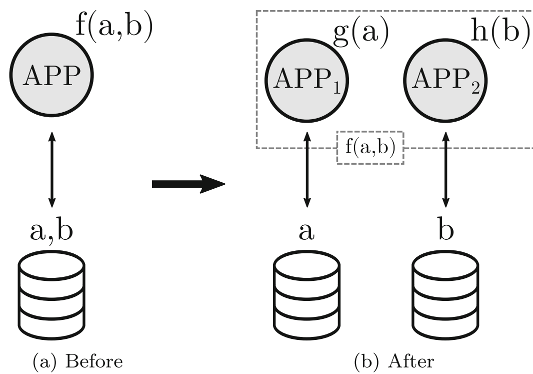


Fig. 1 An application and its database **a** before and **b** after the fragmentation. Both the database and the computational logic are partitioned into fragments and distributed among different CSPs

application functionality $f(a, b)$ must be preserved, but in a way which ensures that each application component knows only either a or b. In other words, application components distributed among different CSPs jointly compute $f(a, b)$ with one component performing $g(a)$ and the other $h(b)$.

Along with providing the paradigm, we demonstrate the way user-provided constraints, which describe how parts of the data must be restricted to specific CSPs, influence the structure of a multi-cloud application. We introduce *multi-cloud computation patterns* which enable computation without violating the provided constraints—plaintext inputs and outputs are only available to the cloud providers that are required to have access to them in order to maintain the application functionality. We describe characteristics of those patterns and explore their performance on a simple application.

The rest of the paper is organized as follows. Section 2 describes previous work related to the privacy of data while outsourced to the cloud. Section 3 introduces secure multi-cloud computation patterns and presents a simple application to exemplify their usage. In Sect. 4, we demonstrate how security requirements influence the architecture of a multi-cloud application in order to get the most optimized application for a given data fragmentation. Performance analysis of the previously introduced application is provided in Sect. 5, while Sect. 6 concludes the paper.

2 Related work

Data fragmentation has received significant attention from the research community as a means for increasing the data privacy [9–11]. Using it in the cloud environment has also been explored to lower the trust assumption toward the system as a whole [12]. The data are fragmented and deployed to the multi-cloud based on the user-provided constraints. Our previous work [13] focused on minimizing the violation of constraints when deploying the multi-cloud application. Assuming that the importance of each data chunk and the trust level for each CSP are provided, we are able to determine a Pareto-optimal set of deployment strategies for a given multi-cloud application. Quantitative factors, such as availability and bandwidth, may be used to estimate the trust level of a CSP [14].

Database fragmentation has been combined with secure multi-party computation to enable the multi-cloud database to execute queries without confidential data pairs having to join at some trusted server [15–18]. Recent work [19] expanded the supported operation set by leveraging *homomorphic encryption*. Proposed system supports all queries from the TPC-H benchmark [20].

With the recent advances in the field of homomorphic encryption, a few programming libraries have emerged [21–24]. While these solutions offer a good research platform, they have not seen significant adoption due to poor performance and accessibility for anyone but experts in the field [25]. Microsoft SEAL [23] is more approachable for non-specialists, but it does not support branching on the encrypted data which is crucial for all but the most simple applications. Additionally, severe security pitfalls arise when this technology is used in practical systems [26], especially when utilized by non-experts. One research [27] argues that even if homomorphic encryption was feasible today, cryptography alone cannot enforce privacy in the cloud. Instead, the user also needs to rely on tools such as distributed computing, complex trust systems, and tamperproof hardware (e.g., Intel SGX and AMD Memory Encryption Technology [28]).

While multi-cloud databases received decent contributions, to the authors' knowledge, no work has explored fragmentation of both the computational logic and the database. We expand on previous work by considering the combination of a multi-cloud database and multi-cloud application code to improve the data security without trusted servers.

3 Secure multi-cloud computation

A trivial application that calculates the payout of company's employees is considered. Payout is calculated by multiplying the number of hours worked with the hourly wage of the employee's workplace. Additionally, if an employee has worked more hours than the workplace requires, a workplace-specific bonus is added to the employee's payout.

The application database consists of two tables: *employee* and *workplace*, as shown in Fig. 2. The table *employee* contains employee ID, workplace ID, and the number of hours worked by the employee. Columns of the *workplace* table define workplace ID, hourly wage, quota, and bonus of the workplace. For simplicity's sake, each employee is assigned

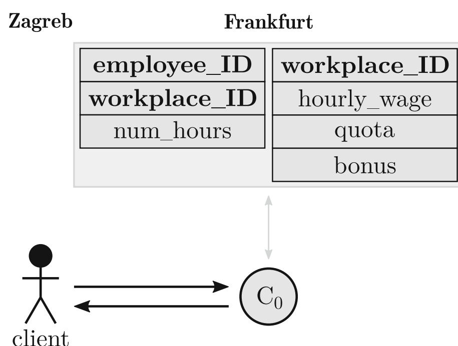


Fig. 2 Monolithic application

Pseudocode 1: Monolithic C_0 functionality.

```
query =
    SELECT num_hours, hourly_wage,
           quota, bonus
    FROM employee JOIN workplace
         ON employee.workplace_ID =
            workplace.workplace_ID
    WHERE employee.employee_ID =
          employee_ID_from_client;
func on_receive_from_client(employee_id)
    num_hours, hourly_wage, quota, bonus
        = execute(query);
    payout = num_hours * hourly_wage;
    if num_hours > quota then
        payout += bonus;
    send({client}, payout);
```

to only one workplace, but workplaces with no employees are allowed to exist in the database.

In a regular single-cloud form, the entire database and all of the necessary code are deployed together to the same cloud, or alternatively, the database may be separated from the code using two different clouds. In this case, the application consists of a single component (C_0 in Fig. 2) that receives employee ID from the client and performs `SELECT` query that fetches number of hours, hourly wage, quota, and bonus for a given employee ID. Pseudocode 1 shows the component procedure which is invoked when employee ID is received from the client. Upon invocation, the query that obtains all the data required for payout calculation is executed. The component then calculates the payout and sends the result to the client.

Multi-cloud variations of this application rely on data fragmentation to ensure confidentiality of sensitive data pairs [9]. Database columns are split apart according to the privacy-related requirements for a given application and deployed to different CSPs. Since no CSP has access to all the data parts necessary to reveal the sensitive information, data in storage are now considered more secure compared to a monolithic database. However, sensitive information may still leak if the data from separate chunks got retrieved for processing in a single point in the system. Therefore, application's computational logic has to be fragmented as well in a way that no party in a multi-cloud environment ever learns all the data that makes a sensitive combination. To enable computation of multi-operand operations where different operands reside in different clouds in such a secure way, we introduce *secure multi-cloud computation patterns*, as shown in Table 1.

The *addition pattern* consists of three components of which the first two obtain the two operands as inputs. The addition is performed in three steps:

1. Component CA_0 obtains input a and generates a random number x. Input a is masked by adding x to it, and the result is sent to CA_1. The random number x is sent to the final component of the pattern CA_2.
2. Component CA_1 obtains the second input b and receives masked first input from CA_0. The masked first input and second input are added together, and the result is sent to CA_2.
3. CA_2 deducts the received random number x from the received masked result to obtain the addition result.

During this process, no component learns both operands. Secure *multiplication pattern* is analogous to the addition pattern, with the difference that multiplication and division are used instead of addition and subtraction. *Comparison pattern* differs slightly since initial component of the pattern CC_0 sends the random number to CC_1. CC_0 and CC_1 then use the same random number to mask their inputs and send those masked inputs to CC_2 for comparison. Since the same random number is used to mask both inputs, masked inputs may be compared to obtain the comparison result.

To present the basic idea in its simplest form, in this paper we intentionally keep the structure of the multi-cloud computation patterns as simple as possible. If for a particular purpose some pattern does not satisfy the required security level (e.g., the range of the data is small, or operand values are deducible from the result of the operation), it may be extended with additional protective elements (e.g., comparison pattern may use additive and multiplicative mask to hide both the difference and the ratio between the two operands). Additionally, in their current form, the patterns assume CSPs are *honest-but-curious* [29]—they do not deviate from the defined protocol but will attempt to learn all possible information from the received messages.

If all operands of a given operation are available in the same cloud, then the operation is computed as usual, using a regular code pattern. However, if the operands are distributed across different clouds, then the operation has to be performed using a secure multi-cloud computation pattern. Since regular pattern is supposed to be far more efficient than the multi-cloud one, we apply the multi-cloud patterns only when this is required by the data fragmentation. This means

Table 1 Regular and secure multi-cloud computation patterns for the arithmetic operations used in example application. Each component of a secure multi-cloud computation pattern is hosted and executed on a separate CSP

Operation	Operands	Regular code pattern	Secure multi-cloud computation pattern
addition	*payout* *bonus*	a + b	a, x = rand() → CA_0 —a+x (1)→ CA_1 (b) —(a+x)+b (2)→ CA_2 —(a+x+b)-x (3)→; CA_0 —x→ CA_2
multiplication	*num_hours* *hourly_wage*	a · b	a, x = rand() → CM_0 —a • x (1)→ CM_1 (b) —(a • x) • b (2)→ CM_2 —(a • x • b)/x (3)→; CM_0 —x→ CM_2
comparison	*num_hours* *quota*	a > b	a, x = rand() → CC_0 —x (1)→ CC_1 (b) —b+x (2)→ CC_2 —(a+x)>(b+x) (3)→; CC_0 —a+x→ CC_2

that the architecture of a multi-cloud application is carefully profiled toward the required data fragmentation in order to optimize its execution.

4 Profiling the architecture of a multi-cloud application

Development of a multi-cloud application starts with the design of a data fragmentation model. The user, the application architect, or the security analyst, defines the security requirements of the application in the form of *security constraints* [13]—sets of columns where at least one column in a set must be deployed to different cloud provider. In other words, all columns in a set may not be available on a single cloud provider. Although such security mechanism does not protect individual columns, sensitive relations between columns are still kept secret. Using this principle, the data remain available for arbitrary computations in the cloud, which would not be the case if encryption was used. For example, if columns *illness* and *name* in some database table are split and deployed to different clouds, neither CSP knows which person suffers from which illness. Some information still leaks (i.e., which illness is the most common), but such leaks are outside the scope of this security mechanism.

A set of security constraints defines how the application logic and the database are split into components and fragments. In other words, the architecture of a multi-cloud application depends on the security constraints provided by the user. Two variations of the multi-cloud application exemplified in Sect. 3 are presented to illustrate this. The first is a *fine split variation* (Sect. 4.1) where the maximum fragmentation of the database is required. The second variation is a *minor split variation* (Sect. 4.2) where only the workplace bonus is required to be separate from the rest of the data. Note that the real-world semantics of security constraints in these examples are not essential for this paper—the idea is to demonstrate how security constraints affect the structure of a multi-cloud application.

4.1 Fine split variation

Fine split variation of the demo application considers maximum possible fragmentation for a given database. Each column, except identifier columns, is constrained with every other column. Identifier columns are constrained with every column, except the columns which they identify (in other words, columns that were in the same table before database fragmentation). This results in a set containing the following 10 security constraints:

$\{employee_ID, \quad hourly_wage\}$
$\{employee_ID, \quad quota\}$
$\{employee_ID, \quad bonus\}$
$\{workplace_ID, \quad num_hours\}$
$\{num_hours, \quad hourly_wage\}$
$\{num_hours, \quad quota\}$
$\{num_hours, \quad bonus\}$
$\{hourly_wage, \quad quota\}$

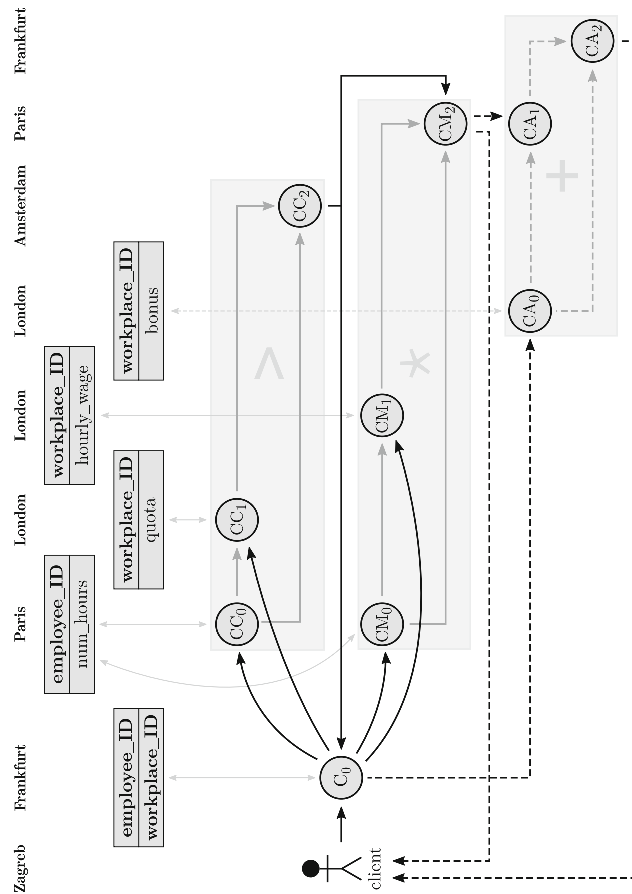


Fig. 3 Fine split variation of multi-cloud application (dashed lines indicate communication that may or may not occur, depending on the control flow). All three arithmetic operations (comparison, multiplication, and addition) are computed using the multi-cloud computation patterns. Eight CSPs are required to run the application since vertically aligned data fragments and application code components may reside in the same cloud

$\{hourly_wage, \quad bonus\}$
$\{quota, \quad bonus\}$

Furthermore, the result of multiplying the number of hours worked by the employee should not be available on the same cloud as the workplace bonus:

$\{bonus, \quad multiplication_result\}$

In total, the set of security constraints for the fine split variation contains 11 constraints.

Assuming that employee and workplace identifiers carry no useful information, those columns may appear together on the same CSP. Furthermore, identifier may appear together with any column from its table. These assumptions are required for preserving the information when fragmenting the database so that the individual fragments may later be joined on those identifiers. In cases where the identifier column carries useful information and occurs in the security constraints, new identifier must be introduced such that it

Pseudocode 2: Fine C_0 functionality.

```
query =
    SELECT workplace_ID
    FROM identifier_table
    WHERE employee.employee_ID =
           employee_ID_from_client;
workplace_id = null;
func on_receive_from_client(employee_id)
    workplace_id = execute(query);
    send({CC0, CM0}, employee_id);
    send({CC1, CM1}, workplace_id);
func on_receive_from_CC2(comparison_result)
    if comparison_result then
        send({CA0}, workplace_id);
```

may appear alongside other columns in its table and identifiers from other tables.

The database is fragmented into 5 data fragments, each containing the identifier column and one column from the original table (Fig. 3). Since the application is maximally constrained, all three operations (multiplication, comparison, and addition) must be performed by leveraging secure multi-cloud computation patterns, as described in Sect. 3.

The comparison result must be dispatched to all components that are affected by the result of an `if` statement. In this case, C_0 must be aware of the comparison result to decide whether or not to send the workplace identifier to CA_0. Additionally, CM_2 needs the comparison result to decide if the multiplication result represents the total payout amount sent to the client or if the workplace bonus must be added to the result, in which case the result is sent into the secure addition pattern.

Component C_0 serves as a connector component between the two original tables since it is the only component that has access to the *identifier_table* (table that contains both identifiers). The component receives employee identifier from the client and fetches the workplace identifier for that employee. Identifiers are then distributed to components that require them: Employee ID is sent to CC_0 and CM_0, while workplace ID is sent to CC_1 and CM_1. C_0 then waits for the result from CC_2 to decide whether or not to send the workplace identifier to CA_0. Pseudocode 2 lists functionality of component C_0 in the fine split variation of the application.

Pseudocode 3 lists the functionality of the initial comparison component CC_0. When the component receives the employee ID from component C_0, the number of hours worked by the employee is retrieved from the database. A random mask is then generated and sent to CC_1, while the sum of the number of hours and the mask is sent to the final comparison component CC_2.

Fine variation of the demo application requires 8 CSPs and a client. In total, 10 application components are shown

Pseudocode 3: Fine CC_0 functionality.

```
query =
    SELECT num_hours
    FROM num_hours_table
    WHERE employee.employee_ID =
          employee_ID_from_C0;
func on_receive_from_C0(employee_id)
    num_hours = execute(query);
    mask = rand();
    send({CC1}, mask);
    send({CC2}, num_hours + mask);
```

in Fig. 3, but there are two pairs of components that reside on the same cloud: $\{CC_0, CM_0\}$ and $\{CM_2, CA_1\}$. Each of these two pairs may be merged into a single executable providing the same functionality as the original pair of components. This reduces the communication cost of the multi-cloud application since the data between those components need not be communicated over the network—it is available right there in memory. Resulting fine split application variation consists of 8 executables which matches the number of required CSPs for this variation.

4.2 Minor split variation

In the minor split example, workplace bonus is split from the rest of the data. A set of security constraints that determines this requirement contains the following constraints:

$\{bonus, \quad hourly_wage\}$
$\{bonus, \quad quota\}$
$\{bonus, \quad num_hours\}$
$\{bonus, \quad employee_ID\}$

Additionally, same as in the fine split variation, the multiplication result is constrained with the workplace bonus:

$\{bonus, \quad multiplication_result\}$

Since only the bonus is now split from the rest of the data, multiplication and comparison are allowed to be computed within the same application component. Therefore, these two operations are calculated using more efficient regular code patterns, instead of performance-costly multi-cloud patterns. The addition operation still requires the use of a secure multi-cloud pattern.

This set of constraints results in two database fragments and four application components, as shown in Fig. 4. Components and databases along the same vertical line (horizontal line in the rotated image) may reside at the same CSP without breaking any security constraints.

Pseudocode 4 outlines the implementation of the component C_0 in the minor split variation. Component C_0 receives the employee identifier from the client and executes the query similar to monolithic application query, but without the bonus in the `SELECT` clause since it is stored in a different database fragment which C_0 cannot access. When the query result is received, C_0 multiplies the number of hours with the hourly wage of the employee's workplace and decides whether or not the bonus should be added to the result. If the number of hours is less or equal to the workplace quota, the multiplication result is sent directly to the client as a final result. Otherwise, the *multi-cloud addition pattern* must be used to add the bonus to the multiplication result without breaking any security constraints. In other words, no CSP should have access to the bonus and either quota, hourly wage, number of hours worked, or employee identifier.

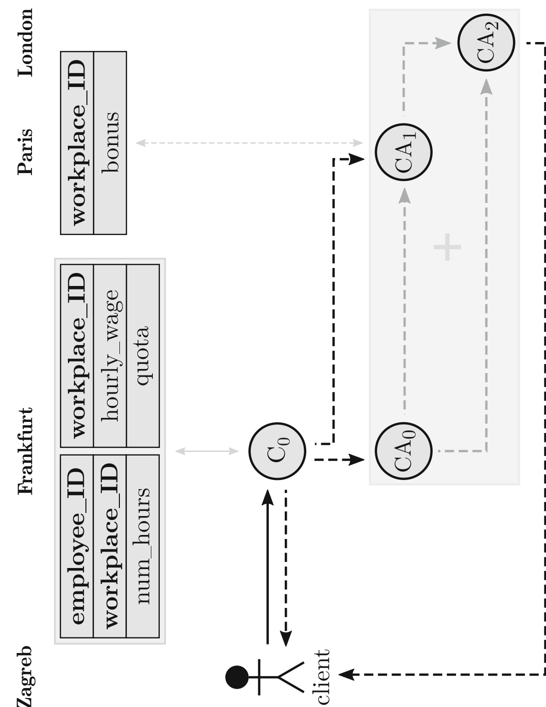
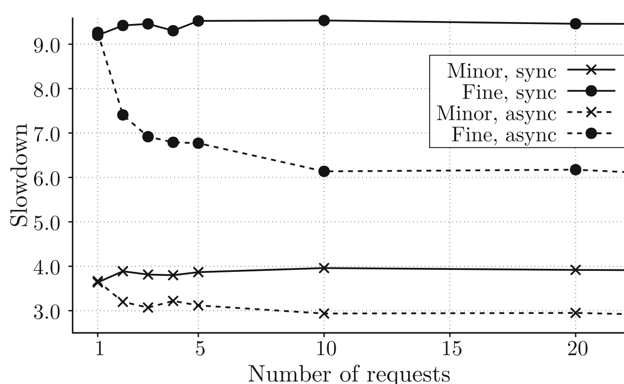


Fig. 4 Minor split variation of multi-cloud application. Comparison and multiplication are computed using the regular code patterns within the C_0. Only the addition is computed using the multi-cloud computation pattern. Three CSPs are enough to run the application since vertically aligned data fragments and application code components may reside in the same cloud

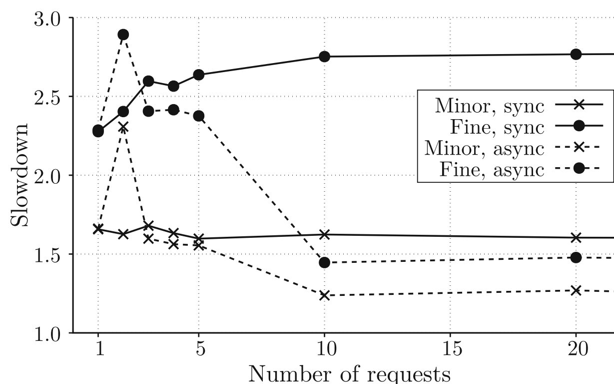
Equivalently to the merging of components in Sect. 4.1, components C_0 and CA_0 may be combined into the same executable to reduce communication costs since they reside on the same cloud. The final structure of the minor split variation results in three executable components and two database fragments. Three CSPs are required for multi-cloud deployment of this application variation.

5 Performance analysis

Monolithic and two variants of multi-cloud application are implemented in C++ using the `Boost Asio` library to facilitate asynchronous communication between applica-



(a) Large data set (50k workplaces, 500k employees)



(b) Small data set (50 workplaces, 500 employees)

Fig. 5 Slowdown of minor and fine variations relative to the monolithic application for **a** large and **b** small data sets

Pseudocode 4: Minor C_0 functionality.

```
query =
    SELECT num_hours, hourly_wage,
           quota, workplace.workplace_ID
    FROM employee JOIN workplace
         ON employee.workplace_ID =
            workplace.workplace_ID
    WHERE employee.employee_ID =
          employee_ID_from_client;
func on_receive_from_client(employee_id)
    num_hours, hourly_wage, quota, workplace_id
        = execute(query);
    payout = num_hours * hourly_wage;
    if num_hours > quota then
        send({client}, payout);
    else
        send({CA_0}, payout);
        send({CA_1}, workplace_id);
```

tion components. Serverless database `SQLite` is used to store the application data. Since the purpose of these tests is solely to gauge the performance impact of the multi-cloud paradigm on the demo application, for convenience, all *virtual machines* (VMs) are hosted by a single cloud service provider—Vultr. VMs offer a single core of a Cascade Lake Intel Xeon CPU clocked at 3.0GHz and 1GB of memory. The client code is running on Intel Core i7-8550U CPU clocked at 4.0 GHz. The client and VMs communicate with bitrate of 550Mbit/s and response time of roughly 40 ms. To simulate a realistic multi-cloud scenario, components that communicate are always hosted in different geographical locations. This results in response times between components of roughly 10 ms and the bitrate of approximately 3Gbit/s. Hosting locations of the application components and the database fragments are marked in the application variation figures (Figs. 2, 3, 4).

The client starts measuring time before sending the first request and stops when the final response has been received. Two types of client behavior are considered: (a) *synchronous*—the client waits for response from the application before sending another request, and (b) *asynchronous*—the client sends the next request even if the response to the previous request has not yet been received. The former behavior measures response times as perceived from the perspective of a particular client. The latter simulates a scenario where multiple clients send requests at the approximately the same time which provides an insight in how the system throughput is affected.

Synchronous client does not allow for *inter-request parallelism*—request computation may not start before the response to the previous request has been received. For example, after C_0 in the fine split variation (Fig. 3) has determined and sent the data to the comparison and multiplication components, the component cannot start handling the next request since the client has not yet sent the next request. The client waits for the response from CM_2 or CA_2 before sending the next request. On the other hand, asynchronous client sends its requests one immediately after another, which means that the next input for C_0 is available as soon as it delivers the output from the previous request to the next components in the workflow. The same holds for other components in the pipeline. In the minor split variation, parallelization is somewhat reduced due to a greater portion of logic contained within a single component (C_0 in Fig. 4), but it still occurs in the multi-cloud addition pattern. In the monolithic application, parallelism is not present since computation is performed by a single component resulting in approximately the same execution times for both versions of the client.

Execution times have been measured on two data sets: one with 50k workplace and 500k employee entries and the other

with 50 and 500 entries, respectively. Figure 5 illustrates the slowdown of minor and fine variations relative to the monolithic application for up to 20 requests. Both client behaviors are considered on both data sets. In the large set scenario, the computation dominates the total execution time. On the other hand, when dealing with the small data set, communication between application components is the dominant factor.

When dealing with large amounts of data, minor and fine variations experience significant slowdown relative to the monolithic application, but they also benefit greatly from the asynchronous client. Furthermore, the fine variation features *intra-request parallelism* since the comparison and multiplication patterns work in parallel on a single request. For a single query, minor variation is up to 4 times slower than the monolithic application. When using the asynchronous client and as the number of requests increases, the slowdown drops to below 3× due to inter-request parallelism. The fine split application variation benefits more from the asynchronous client since the computation pipeline is longer. Slowdown drops from roughly 9.4× for a single request to 6.17× for 20 consecutive requests.

When the small data set is considered, the response of a fine variation takes 2.3 times longer than the monolithic application response, while minor variation response is only 1.66× slower. Since the communication costs now have a greater share in the total execution time, the benefits of the asynchronous client become more apparent with larger numbers of requests.

6 Conclusion

We introduced a novel paradigm for secure multi-cloud application development and demonstrated its use on a simple application. Furthermore, the performance analysis indicates that the paradigm is suitable for small-scale applications or critical parts of larger cloud systems. Although the retrieval of data from multiple database fragments and usage of multi-cloud computation patterns affect the performance of the application to some degree, slight performance drop may still be acceptable if the security of the outsourced sensitive data is increased significantly. What is left outside of the discussion in this paper is how using multiple clouds to run a single application affects the application's operational costs.

The next step in our research is to develop a procedure and a tool for automatic generation of the multi-cloud application components and database fragments based on the provided monolithic application code, monolithic database, and data-related security constraints. This makes the development of the multi-cloud applications transparent to the programmers, but also facilitates regeneration of the multi-cloud application if the security constraints or available number of clouds change over time. We aim to combine this procedure with our previous work [13] to determine the optimal deployment of application components and database fragments to the available CSPs. Another possible avenue of research may explore the preservation of the database functionality as much as possible after the fragmentation and deployment to the multi-cloud environment (e.g., data integrity enforcement). While there is still a lot of research and practical work left to be done before our vision of the multi-cloud paradigm becomes suitable for non-trivial applications, we believe that the distributed trust systems, such as the proposed multi-cloud application model, will eventually prevail.

Acknowledgements This research is co-sponsored by the European Regional Development Fund through a research Grant KK.01.2.1.01.0109. We acknowledge the support of the Ministry of Economy of the Republic of Croatia as well as our research partners OROUNDO Mobile GmbH Austria and OROUNDO Mobile GmbH Subsidiary Croatia.

Data availability The data sets generated and analyzed during the current study as well as the source code used for the experiments are available from the corresponding author on reasonable request.

Declarations

Conflict of interest The authors have no competing interests to declare that are relevant to the content of this article.

Human and animal rights We did not use animals and/or human participants in the study reported in this work.

References

1. Mell, P.M., Grance, T.: The NIST Definition of Cloud Computing. Tech. rep., National Institute of Standards and Technology (2011)
2. Jansen, W.: Cloud Hooks: security and privacy issues in cloud computing. In: 44th Hawaii International Conference on System Sciences, pp. 1–10. IEEE (2011)
3. Modi, C., et al.: A survey on security issues and solutions at different layers of cloud computing. J. Supercomput. **63**(2), 561–592 (2013)
4. Hashizume, K., et al.: An analysis of security issues for cloud computing. J. Internet Serv. Appl. **4**(1), 1–13 (2013)
5. Kelbert, F., et al.: SecureCloud: secure big data processing in untrusted clouds. In: Design, Automation & Test in Europe Conference & Exhibition, pp. 282–285. IEEE (2017)
6. AlZain, M.A., et al.: Cloud computing security: from single to multi-clouds. In: 45th Hawaii International Conference on System Sciences, pp. 5490–5499. IEEE (2012)
7. Bernstein, D., et al.: Blueprint for the intercloud—protocols and formats for cloud computing interoperability. In: 4th International Conference on Internet and Web Applications and Services, pp. 328–336. IEEE (2009)
8. Celesti, A., et al.: How to enhance cloud architectures to enable cross-federation. In: International Conference, pp. 337–345 (2010)
9. Ciriani, V., et al.: Combining fragmentation and encryption to protect privacy in data storage. ACM Trans. Inf. Syst. Secur. **13**(3), 1–33 (2010)
10. Raj, S., Arunkumar, B.: Enhanced encryption for light weight data in a multi-cloud system. In: Distributed and Parallel Databases, pp. 1–10 (2021)

11. Abed, H.N., Mahmood, G.S., Hassoon, N.H.: A secure and efficient data distribution system in a multi-cloud environment. Malays. J. Sci. Adv. Technol. **9**(3), 109–117 (2021)
12. Hudic, A., et al.: Data confidentiality using fragmentation in cloud computing. Int. J. Pervas. Comput. Commun. **9**(1), 37–51 (2012)
13. Lovrencic, R., et al.: Security risk optimization for multi-cloud applications. In: International Conference on the Applications of Evolutionary Computation, pp. 659–669. Springer, Berlin (2020)
14. Alam, B., Fadlullah, Z., Choudhury, S.: A resource allocation model based on trust evaluation in multi-cloud environments. IEEE Access **9**, 105577–105587 (2021)
15. Wu, S., et al.: ServeDB: secure, verifiable, and efficient range queries on outsourced database. In: 35th International Conference on Data Engineering, pp. 626–637. IEEE (2019)
16. Emekci, F., et al.: Dividing secrets to secure data outsourcing. Inf. Sci. **263**, 198–210 (2014)
17. Xue, K., et al.: Two-cloud secure database for numeric-related SQL range queries with privacy preserving. IEEE Trans. Inf. Forensics Secur. **12**(7), 1596–1608 (2017)
18. Xiang, T., et al.: Processing secure, verifiable and efficient SQL over outsourced database. Inf. Sci. **348**, 163–178 (2016)
19. Wang, L., Yang, Z., Song, X.: SHAMC: a secure and highly available database system in multi-cloud environment. Futur. Gen. Comput. Syst. **105**, 873–883 (2020)
20. Poess, M., Nambiar, R.: TPC Benchmark H Standard Specification, tech. rep., Transaction Processing Performance Council (2010)
21. Halevi, S., Shoup, V.: Algorithms in HElib. Advances in Cryptology, pp. 554–571. Springer, Berlin (2014)
22. Chillotti, I., et al.: Faster fully homomorphic encryption: bootstrapping in less than 0.1 seconds. In: Advances in Cryptology, pp. 3–33. Springer, Berlin (2016)
23. Chen, H., Laine, K., Player, R.: Simple encrypted arithmetic library—SEAL v2.1. In: International Conference on Financial Cryptography and Data Security, pp. 3–18. Springer, Berlin (2017)
24. Cheon, J.H., et al.: Homomorphic encryption for arithmetic of approximate numbers. In: Advances in Cryptology, pp. 409–437. Springer, Berlin (2017)
25. Crockett, E., Peikert, C., Sharp, C.: ALCHEMY: a language and compiler for homomorphic encryption made easy. In: Conference on Computer and Communications Security, pp. 1020–1037. ACM (2018)
26. Peng, Z.: Danger of using fully homomorphic encryption: A look at Microsoft SEAL. ArXiv (2019)
27. Van Dijk, M., Juels, A.: On the impossibility of cryptography alone for privacy preserving cloud computing. In: 5th USENIX Conference on Hot Topics in Security, USENIX Association, pp. 1–8 (2010)
28. Mofrad, S., et al.: A comparison study of Intel SGX and AMD memory encryption technology. In: 7th International Workshop on Hardware and Architectural Support for Security and Privacy. Association for Computing Machinery, pp. 1–8 (2018)
29. Paverd, A., Martin, A., Brown, I.: Modelling and automatically analysing privacy properties for honest-but-curious adversaries. Tech. rep., University of Oxford (2014)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.