AT-DIFC⁺: Toward Adaptive and Trust-Aware Decentralized Information Flow Control

CHARILAOS SKANDYLAS, NARGES KHAKPOUR, and JESPER ANDERSSON,
Linnaeus University, Sweden

Modern software systems and their corresponding architectures are increasingly decentralized, distributed, and dynamic. As a consequence, decentralized mechanisms are required to ensure security in such architectures. Decentralized Information Flow Control (DIFC) is a mechanism to control information flow in distributed systems. This article presents and discusses several improvements to an adaptive decentralized information flow approach that incorporates trust for decentralized systems to provide security. Adaptive Trust-Aware Decentralized Information Flow (AT-DIFC⁺) combines decentralized information flow control mechanisms, trust-based methods, and decentralized control architectures to control and enforce information flow in an open, decentralized system. We strengthen our approach against newly discovered attacks and provide additional information about its reconfiguration, decentralized control architectures, and reference implementation. We evaluate the effectiveness and performance of AT-DIFC⁺ on two case studies and perform additional experiments and to gauge the mitigations' effectiveness against the identified attacks.

CCS Concepts: • **Computer systems organization** → **Self-organizing autonomic computing**; • **Security and privacy** → *Formal security models*; **Distributed systems security**;

Additional Key Words and Phrases: Adaptive security, decentralized information flow control, adaptive trust, decentralized feedback loop

ACM Reference format:

Charilaos Skandylas, Narges Khakpour, and Jesper Andersson. 2021. AT-DIFC⁺: Toward Adaptive and Trust-Aware Decentralized Information Flow Control. *ACM Trans. Auton. Adapt. Syst.* 15, 4, Article 13 (December 2021), 35 pages.

<https://doi.org/10.1145/3487292>

13

1 INTRODUCTION

Today, many software systems are characterized by the distribution of behavioral responsibilities on multiple components deployed on connected computational devices [4]. The systems provide for openness, which allows devices and components to join and leave a system at runtime, throughout the system's lifecycle. Openness can lead to situations where a centralized control model is perceived as top-heavy and unfit. Thus, decentralized application control is likely a better fit for open, decentralized systems.

This work was funded by the Swedish Knowledge Foundation (Grant No. 20160186).

Authors' address: C. Skandylas, N. Khakpour, and J. Andersson, Linnaeus University, Växjö, Sweden; emails: charilaos.skandylas@lnu.se, narges.khakpour@lnu.se, jesper.andersson@lnu.se.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2021 Copyright held by the owner/author(s).

1556-4665/2021/12-ART13

<https://doi.org/10.1145/3487292>

An open and decentralized setting elevates security to a primary system concern. Multiple collaborating processes, owned by independent stakeholders and each process's unique security requirements, impel developers to prioritize security. As a consequence, each process is required to define its individual security policies. However, in a decentralized system, processes must trust that no other process violates their policies without explicit approval. Tackling security in a pure, decentralized environment requires mechanisms that manage which processes to trust and the extent of trust for each process.

Trust in decentralized systems has been explored in the past. Current trust-based security solutions focus on allowing or disallowing communication in the system at the granularity of nodes. These approaches are often domain specific and focus on either defending against specific attacks or identifying and/or isolating malicious nodes [1, 11] based on trust metrics [31] or reputation-based trust [1, 21], lacking the capabilities to deal with fine-grained security policies that control the access to, modification and propagation of information between entities. Furthermore, they operate under static enforcement rules [2, 31], even in the presence of dynamic trust and do not feature a dynamic trust architecture. As a result, given the openness of the system and the lack of peer concession, approaches based solely on trust are lacking the required tools to deal with the uncertainty, rapid-change in the system, and the dynamicity of trust.

Consider a microservice architecture that implements a web shop that depends on a third-party payment service. If the payment service is malicious or compromised, then there are no guarantees that the payment or the transaction information will not be tampered with or leaked without the web shop noticing. Information flow control techniques [18] that track and control the flow of the information in a system can provide the critical guarantees. In a decentralized setting, a variant of classical information flow control, decentralized information flow control [15] allows each process to independently specify its policies and provides mechanisms for policy enforcement throughout the system. Current **Decentralized Information Flow Control (DIFC)** mechanisms rely on a globally trusted controller. However, such a controller is impractical as (i) it cannot be reliably agreed upon due to the mutual distrust between the entities of the system and (ii) the system openness and its decentralized nature designate the controller as the single point of failure, should the controller be compromised the entirety of the system is compromised, while if the controller is overloaded, then the system's security enforcement is delayed. Moreover, such mechanisms do not consider trust as a significant factor that affects security properties in a mutually-distrusting-processes setting. Since basic security properties, i.e., confidentiality and integrity are rooted in the trust between the system components, the policy enforcement model must also be dependent on trust.

To address the above challenges, we proposed **Adaptive Trust-Aware Decentralized Information Flow Control (AT-DIFC)** [22], that is, an adaptive approach to facilitate trust-aware decentralized information flow control. AT-DIFC extends DIFC with capabilities for multiple decentralized controllers, each responsible for managing a part of the decentralized system. The controllers' capabilities, controlled entities, and policies are dynamically adapted to reflect changes in the environment, the security of system entities or their trust policies. The system is partitioned into non-communicating subsystems, satisfying the principle of least privilege, and enhancing security since information that needs not be shared to third principals will never be transmitted to them. In AT-DIFC, each entity can specify its policies independently, using a trust-estimate approach based on the information it shares with other entities. The model enforces information flow control policies that prohibit communication that would entail leakage or illegal tampering of information. Its adaptation architecture provides the required structures and behaviors for adapting

controllers to meet different stakeholder goals as well as to handle the uncertainty caused by the openness of the model.

However, AT-DIFC suffers from some security weaknesses that make it susceptible to a number of attacks that take advantage of the system openness and the decentralized nature of its operation. We have identified three categories of attacks: (i) attacks that aim to discredit other system entities by abusing valid system operations, e.g., by taking advantage of the capability to dynamically alter security policies at any time to force DIFC policy violations that result in the defamation of a non-malicious entities, and (ii) attacks that attempt to claim illegitimate ownership of process resources, or perform privileged operations while lacking that privilege. For instance, an attacker could claim to be the owner of a secret piece of information and attempt to coerce the system to declassify that information, i.e., an action that would violate confidentiality. If that information has already been declassified in the system, then, nevertheless, the attacker can attempt to use that information in his/her security policies to deceive the system into rejecting valid DIFC messages, and (iii) attacks that attempt to exploit established countermeasures to starve system resources or stall the system's operations. The security of our model and its implementation is critical, because if the policy enforcement mechanisms are bypassed or compromised, then the security guarantees provided by DIFC can no longer be relied upon. Once the attacker is able to manipulate the trust of system entities at will, since trust is an essential part of the model, the system's security policy enforcement becomes unreliable. Resource starvation or stalling of the system operation can make the system unresponsive and unable to operate normally. In light of these shortcomings, we have enhanced the AT-DIFC model and in this article present **Adaptive Trust-Aware Decentralized Information Flow plus (AT-DIFC⁺)**, an improved version of AT-DIFC that is able to mitigate the identified attacks and is extended with additional operations to further improve its security.

This article extends [22] with the following contributions:

- We extend AT-DIFC to address the newly identified security weaknesses and strengthen the security of its adaptation and information flow control mechanisms.
- We give a detailed description of the improved AT-DIFC⁺ model, which includes: (i) the (formal) model, (ii) the reconfiguration plan generation and reconfiguration operation algorithms, and (iii) an improved planning phase for our hybrid architecture, which has been revised to improve its efficiency.
- We discuss our reference implementation and provide details regarding the concrete system operation and the implemented mitigations.
- We evaluate the updated AT-DIFC⁺ model and its implementation by applying it on two case studies. We conducted several experiments that evaluate the security of our approach in multiple scenarios, including the assessment of the improved AT-DIFC⁺ model. In addition, we conducted experiments to evaluate our mitigations' effectiveness against the newly discovered weaknesses.

The remainder of this article is structured as follows. Section 2 provides an introduction to DIFC. In Section 3, we introduce a running example used in the rest of the article. Section 4 introduces and describes the approach and provides a formal specification of the updated AT-DIFC⁺ model. Section 5 discusses reconfiguration, in particular, reconfiguration operations, reconfiguration plan generation and utility-based evaluation. A reference implementation of the approach is described and discussed in Section 6 and Section 7 elaborates on the design of the decentralized adaptive security architectures. Section 8 details the evaluation of two demonstrator applications. In Section 9, we present related work and contrast our work. Section 10 concludes and presents future research paths.

2 INFORMATION FLOW CONTROL

Information Flow Control (IFC) is a mechanism to control information flow in a system to enforce *confidentiality* and/or *integrity*. Confidentiality is satisfied if confidential information cannot leak to public channels and integrity guarantees that unauthorized users cannot manipulate data [8]. IFC can be enforced with centralized or decentralized mechanisms. In this article, we concentrate on DIFC used to control information flow in a decentralized system context. DIFC ensures system confidentiality and integrity, even with untrusted code or untrusted users in decentralized environments. [15]. It allows the designers to control how data flows between application components and the outside world, by defining policies to be enforced by the system when the data flows to and from an application.

One of the most well-known models of decentralized information flow control is the Flume model [13]. In Flume, tags and labels are used to track how information flows through the system. A label consists of multiple tags. A system consists of a set of processes where each process p is equipped with a label for secrecy, denoted by S_p , or a label for integrity, represented by I_p but rarely both. For instance, a user of an email client c would label his account information private by adding the *password* and *username* tags to c 's secrecy label S_c . That would allow the user to specify policies for using *password* and *username* in processes that interact with the email client.

Any process can create new tags to define decentralized privileges. A privilege is defined based on capabilities, i.e., each tag g is associated with two capabilities g^+ and g^- . Each process p is equipped with a set of capabilities $\mathcal{P}_p$. For example, if $password^+ \in \mathcal{P}_c$, then c can add *password* to its labels, whereas if $password^- \in \mathcal{P}_c$, then c can remove *password* from its labels. When a process is able to add a tag to its labels, that means that it is able to read or modify the data associated with that tag, depending on if the label is associated with secrecy or integrity. Since the *password* tag is associated with a secrecy label, c can read the data associated with the password tag. If a process owns both g^+ and g^- , then it fully controls the appearance of g in its labels and we say it contains a dual privilege on g . The set D_p represents all tags for which p has a dual privilege.

In Flume, a system is secure if and only if all process label changes are *safe* and all messages are *safe*. A safe label change is defined as follows:

$$\{l' - l\}^+ \cup \{l - l'\}^- \subseteq \mathcal{P}_p, \quad (1)$$

where the notation $\{l\}^+$ refers to the set of tags of a label for which a process has addition capabilities, $\{l\}^-$ refers to the set of tags for which removal capabilities exist, and $-$ denotes the set difference operator. Informally this means that to update a label l of a process p to l' , the process must have the capabilities to change the label. Furthermore, a message from a process p to a process q is safe if the following condition holds:

$$S_p - D_p \subseteq S_q \cup D_q \quad \text{and} \quad I_q - D_q \subseteq I_p \cup D_p. \quad (2)$$

Informally, this means that the message receiver must have knowledge of secret information shared by the sender (i.e., $S_p - D_p \subseteq S_q \cup D_q$), while the sender must have permission to modify information transmitted to the receiver (i.e., $I_q - D_q \subseteq I_p \cup D_p$). If, for instance, the mentioned user would like to migrate his e-mail account from one client c to another nc , then $S_c - D_c \subseteq S_{nc} \cup D_{nc}$ which is equivalent to $\{password, username\} \subseteq S_{nc} \cup D_{nc}$ where S_{nc} and D_{nc} show the new client's secrecy and duality labels.

3 RUNNING EXAMPLE

Figure 1 depicts an example item shop microservice architecture. The item shop is composed of seven communicating modules that belong to three different stakeholders. The item shop itself comprises four microservices: Item, Frontend, Account, and Cart. The item shop communicates with

Fig. 1. Itemshop microservice architecture comprising seven modules and four trust contexts. The three bottom-level trust contexts are named after their stakeholders.

a Payment microservice owned by a bank and its shipping is handled by a shipping company using two microservices of Shipping and Logistics. The microservices are grouped into three contexts based on their stakeholder and mutual trust. These three contexts then form a global context to allow communication between the microservices they contain. Therefore, four trust contexts are shown: ItemShop, Bank, Shipping Service, and Global.

In a classical decentralized information flow control setting, each microservice would correspond to a DIFC process that defines a single label for its secrecy or integrity. A label in that context specifies the security policy for the microservices and includes a tag for each piece of security-relevant information. A centralized DIFC controller would enforce information flow control based on Equations (1) and (2). However, classical DIFC approaches suffer from a few limitations when applied to a decentralized system like the one shown in Figure 1. First, they do not utilize the already established trust between the modules in a given trust context. Second, they are unable to express more fine-grained security policies at the interface invocation level, or based on trust between the system entities. For example, an invocation of the `add_to_cart` interface of the `Cart` microservice, does not need to know about the customer's credit card information, even if the `Cart` has to store that information to invoke the `pay` interface of the `Payment` microservice. Lastly, the central controller is both a single point of failure and a bottleneck. These four aspects constitute a research gap we addressed by introducing AT-DIFC and which we improve upon in this work.

To address the first and third issues, our approach forms trust contexts that combine DIFC enforcement with techniques based on trust-estimation to incorporate trust in DIFC. We address the second issue by defining our security policies based on the parameters and return value of the callable interfaces of the modules in the system rather than the data that each module contains, which allows us to express security policies at a higher granularity. The fourth problem is approached by equipping each of our trust contexts with a controller, bound to the trust context and responsible to facilitate information flow control for its direct children, i.e., the context elements. Information is therefore only propagated upwards when modules belonging to different controllers need to communicate. As a result, should a controller fail or be compromised, only part of the system enforcement is compromised, as the rest of the controllers will operate as normal. Furthermore, since enforcement happens at the trust context level, more DIFC messages can be concurrently verified. Finally, information is only shared with the part of the system that is absolutely necessary to perform DIFC enforcement. For instance, an exchange between the `Item` and `Cart` microservices will be handled by the `Itemshop` controller, and thus the `Global` controller will never learn of that interaction. As a result, the principle of least privilege is satisfied, since the security critical information is only shared with as few controllers as strictly needed for enforcing DIFC.

4 ADAPTIVE TRUST-AWARE DIFC

The section presents Adaptive Trust-Aware Decentralized Information Flow (AT-DIFC⁺), a revision of AT-DIFC, a decentralized information flow control model that is able to adapt its architecture to

deal with changes in the system entities or their established trust. It formalizes the basic components, trust computation, security policies, and security goals. In addition, it discusses the security weakness of AT-DIFC, the attacks that could take advantage of them and our mitigations included in AT-DIFC⁺.

4.1 Overview

We model systems as a set of communicating *modules* where a module is an independent computation unit that communicates with other modules via message passing. This system model supports many different types of system architectures, e.g., microservices, multi-agent systems, service-oriented systems, and component-based systems. To enforce security in such a system, we incorporate trust in decentralized information flow control. The new model, AT-DIFC is in principle, an extension of the Flume model with trust, adaptation and decentralized control.

We use the widely accepted definition of trust in Reference [7]: “Trust (symmetrically, distrust) is a particular level of the subjective probability with which an agent will perform a particular action, both before we can monitor such an action (or independently of his capacity of ever being able to monitor it) and in a context in which it affects our own action.” This means that trust is the probability that an entity e will take a beneficial action in regard to a trusting another entity e' . Trust is determined by the confidence level, depending on which, an actor decides whether to give out information or not. The notion of trust can be broadly classified into two categories: *direct trust* and *recommended trust*. Recommended trust can additionally be split into two subcategories: *active recommended trust* and *passive recommended trust* [1]. Direct trust is defined based on the satisfaction of an entity’s direct interaction with another entity. Recommended trust is based on indirect interactions [1]. Active recommendation happens when an entity only takes into account its trusted partners when requesting a recommendation for another untrusted entity. Passive recommendation, however, takes into account the whole set of paths from one entity to another when requesting a recommendation, that includes untrusted entities in the system.

To enforce security in a dynamic environment, we utilize AT-DIFC⁺ to control information flow, where the complete information needs not be distributed to all controllers, and there is no trusted central controlling authority. The trusted set of modules varies from module to module and may not be known in advance.

To this end, we organize modules into *trust contexts* that are formed based on strong trust between their module members. Furthermore, trust contexts are further organized in a hierarchical structure to facilitate communication between their modules. We call such a hierarchy of trust contexts a *trust architecture*. The information flow to/from a trust context is controlled by a *trust context controller* that controls information flow by enforcing the context members’ security policies. When an entity attempts to send a message to another, the immediate context controller interposes the message. Subsequently, the policies of the sender and receiver entities are forwarded to their first common ancestor, who is then able to enforce security policies for the particular message locally. Whenever a set of mutually trusted modules form a trust context, information flow between the context’s modules is enabled. Each context controller acts as a local control point for its sub-hierarchy. Therefore, the overall model comprises “trees” of disconnected trust contexts that permit information to flow between their members but not between each other. The model is composed of a forest of communicating trees, where two contexts communicate with each other via their shared ancestors. Therefore, instead of a single global root, we create multiple localized tree hierarchies, each controlling its descendant sub-trees.

The system entities and their security policies are defined based on the trust evolution over time. Therefore, a trust architecture should be dynamic and adaptable, which requires a self-organizing trust architecture.

Fig. 2. The AT-DIFC⁺ concepts, communication and adaptation dynamics. This figure shows a merge operation followed by an isolate operation and a rejected AT-DIFC⁺ operation.

The goal is to form a trust architecture where the information flow is maximized and security policies are enforced. To that end, the trust contexts must contain modules with the highest mutual trust. Hence, we restructure the architecture to improve the permitted flow. We merge two sibling trust contexts, split a context when its trust deteriorates or isolate a maliciously behaving trust context to re-structure the overall trust architecture.

Figure 2 represents a typical trust architecture. It features an execution of an adaptation plan modifying the architecture from architecture t_0 to architecture t_1 . The plan consists of two reconfiguration actions, a merge of two trust contexts, TC_{12} and TC_{34} into TC_{1234} , and an isolation of M_4 . Finally a failed attempt at an invocation from M_2 to M_3 on interface i is shown where M_2 's i label is propagated to TC_{12} and then to TC_{1234} where DIFC validation happens, while M_3 's i label is forwarded first to TC_{34} and then to TC_{1234} . Since M_2 's security policies are satisfied but M_3 's are not, the invocation validation fails. As a result, M_2 is notified of the failure and subsequently, does not perform the invocation. It is important to note that there are two communication channels in our model. The first channel submits and receives AT-DIFC⁺ messages, which propagate through the trust architecture hierarchy. These messages contain the security relevant information for the invocation but no actual data, and are used to determine whether it is secure to perform an invocation. The second channel is the application specific channel through which the actual invocation occurs.

We support two types of trust contexts: (a) atomic trust contexts, that represent contexts whose elements cannot be separated from each other, and (b) non-atomic trust contexts, the elements of which can be freely relocated. Atomic trust contexts allow us to tradeoff between security and maintaining the system's functionality. In that way, our approach can support a larger class of systems, and provide a more realistic model. An atomic trust context is immune to the operations that we define in AT-DIFC⁺ and is treated as a single module. As a result, while the functionality within the context is always maintained and guaranteed regardless of the trust of its members, if a module belonging to that context is compromised, then the system cannot single it out and isolate it, which entails that the satisfaction of the security policies of the modules that communicate with the atomic context cannot be guaranteed.

4.2 Formalization of Components

Let $\mathcal{C}$ be the set of all context controllers, $\mathcal{M}$ be the set of all modules and $\mathcal{T}$ be the set of all trust contexts where the trust contexts, context controllers and modules are mutually disjoint, i.e., $\mathcal{M} \cap \mathcal{T} = \emptyset$, $\mathcal{M} \cap \mathcal{C} = \emptyset$ and $\mathcal{C} \cap \mathcal{T} = \emptyset$. A *trust context* is defined as a tuple $t = \langle c, \mathcal{E} \rangle$ where c is the context controller and $\mathcal{E} = \{e_0, \dots, e_n\}$ is a set of elements controlled by it with $\mathcal{E} \subseteq \mathcal{M} \cup \mathcal{T}$. A module belongs to only a single context, i.e., for any two arbitrary trust contexts $t = \langle c, \mathcal{E} \rangle$ and

$t' = \langle c', \mathcal{E}', \rangle$ where $t \neq t'$, it holds that $\mathcal{E} \cap \mathcal{E}' = \emptyset$ and $c \neq c'$. A *module* is defined as a tuple $m = \langle I, \lambda, \mathcal{P} \rangle$ where

- $I = \{i_0, \dots, i_n\}$ is a set of interfaces where $i_j = \langle n_j, p_j, r_j, s \rangle$ and n_j, p_j and $r_j, 0 \leq j \leq n$ correspond to the name, parameter set and return value of the interface, respectively. Furthermore, s corresponds to a cryptographically secure token that we use as a version signature for the interface. Each time the interface's label is modified, a new version signature is generated and future invocations must use the new signature to invoke the interface.
- $\lambda : I \rightarrow \mathcal{L}$ is a function that maps each interface to a label in $\mathcal{L}$, where $\mathcal{L}$ is the set of all labels.
- $\mathcal{P}$ is a capability set that enables a module to alter its labels.

The set of atomic trust contexts is denoted by $\mathcal{T}_a$. Elements of an atomic trust context, are treated as a single element from an outsider's perspective, i.e., from any other context's point of view, an atomic context, comprises just a single entity and its controller. A module is critical if its functionality is essential to the system and its messages should never be dropped. The set of critical modules is denoted by $\mathcal{M}_c$.

An *invocation* is a call to an interface of a module by some other module, defined as a tuple $\iota = \langle m_1, i, m_2, s_i \rangle$ where m_1 is the source module, m_2 is the destination module, i is the interface of m_2 being invoked, and s_i corresponds to the sender's known version signature for the invoked interface. The purpose of s_i is to ensure that the sender and receiver are referring to the same version of the interface's security policies, i.e., to ensure that the security policies of the interface have not been altered since m_1 's last invocation of that interface. Every module holds a label for each of its interfaces. The tags of the label correspond to the parameters of the interface, set by the module that exposes the interface (callee), while the tag of the return value, if any, is set by the invoking module (caller). Labels are allowed to flow through trust contexts to allow for AT-DIFC⁺ enforcement but are never stored in contexts. If an interface's security policies have been altered, which entails a change in the corresponding label, then any module invoking that interface must also update its label to receive the interface's new version signature.

A tag is formally defined as a tuple $g = \langle o, id, v, d \rangle$, where (i) o corresponds to the tag's owner, i.e., the module that created the tag and holds complete ownership over it, (ii) id corresponds to a tag ID, an opaque unique token that identifies the tag, (iii) v corresponds to a real value $\in [0, 1]$ that represents how much an entity has to be trusted in order for the tag to be allowed to be part of that entity's labels, and (iv) d corresponds to the maximum distance (i.e., the number of visited context controllers to reach its destination) that the specific tag is allowed to flow away from the module that has created it. A label is defined as a set of tags, i.e., $l = \{g_0, \dots, g_n\}$ associated with a specific security property. Typically the security property is secrecy, though labels can be expressed for integrity or more complicated properties. We define capabilities that allow to add or remove a tag from a label. A capability allowing the tag g_j to be added to a label l is represented as g_j^+ , whereas one allowing removal of g_j from l is represented by g_j^- . A capability set refers to a set of capabilities of the module represented by $\mathcal{P}$. The values related to trust value and distance are selected based on how security relevant the information represented by each tag is. That is, if the particular piece of information is critical, then the trust required by its handler will be very high, closer to 1. Distance represents a measure of indirect trust, it is especially useful when privacy or spread of information is of concern. As an example, consider a user who wants to directly allow his/her payment module to share his/her credit card details with the bank, but would not like to allow the bank to share that information further. In that case a very small value of distance is suggested, e.g., 1. Note that distance limits the spread of information regardless of what the proximity of the modules at certain points in time is. This means that a module might be highly trusted and therefore, gain a tag at

Table 1. The Payment and Shipping Modules, Including Interfaces, Labels, Tags, and Capability Sets

Module	Payment
Interface	$i_{pay} = \langle pay, \{amount, user\}, \{paid\}, s_{pay} \rangle$
Labels	$l_{pay} = \{paid, amount, user\}$
Tags	$amount = \langle Cart, 0.8, amount, 1 \rangle$ $user = \langle Item, 0.5, user, 5 \rangle$ $paid = \langle Payment, 0.5, paid, 1 \rangle$
Capabilities	$\mathcal{P}_{Payment} = \{amount^+, user^+, amount^-, user^-\}$
Module	Shipping
Interface	$i_{ship} = \langle ship, \{user, destination\}, \{shipped\}, s_{ship} \rangle$
Labels	$l_{ship} = \{shipped, user, destination\}$
Tags	$shipped = \langle Shipping, 0.5, shipped, 1 \rangle$ $user = \langle Item, 0.5, user, 5 \rangle$ $destination = \langle Payment, 0.9, destination, 1 \rangle$
Capabilities	$\mathcal{P}_{Payment} = \{user^+, destination^+, user^-, destination^-\}$

some point time, and become untrusted later. Even though the tag is now known to that module, since the distance constraint cannot be met, any message that contains the tag will violate DIFC policies. That implies that even if a module behaves non-maliciously to gain the required tags and then tries to use the tags maliciously, it will not be able to do so.

Example 1. The Payment microservice of the system in Figure 1 exposes a pay interface that has two parameters user and amount and returns paid. A label l_{pay} associates a label to the pay interface that describes the policies associated with each parameter as a tag. For instance, amount is considered highly secret and as a result, it is assigned a trust value of 0.8 (i.e., $v = 0.8$) and a maximum distance of 1 (i.e., $d = 1$). That means that any entity in the system that receives or needs to add the amount parameter to its labels has to be at least trusted by 80% and be at most one trust context away from Payment. The parameter tag user is considered less secret and is assigned $v = 0.4$ and $d = 5$, allowing the tag to “flow” much further away from Payment and be added to labels of less trusted modules. Table 1 depicts the Payment and Shipping modules, each containing a single interface that takes two arguments and a label associated with that interface. Each of the labels is associated with three tags: two of them are created and fully regulated by the module via its capability set $\mathcal{P}$ that correspond to the interface’s arguments, and the third is controlled by the caller and corresponds to the interface’s return value.

4.3 Formalization of Trust

We define the partial function $\text{Trust} : \mathcal{M} \cup \mathcal{T} \times \mathcal{M} \cup \mathcal{T} \rightarrow [0, 1]$ that returns the trust between two entities, a value in the interval $[0, 1]$. The function $\text{Trust}(e, e')$ shows how much e trusts e' . We define trust between two entities as a combination of direct and recommended trust. We opt to use the number of messages successfully exchanged as a basis for trust. We define direct trust as the number of messages successfully transmitted divided by the number of messages sent in total. Intuitively, since messages are dropped by a trust context controller when AT-DIFC⁺’s security policies are violated, the higher the ratio of accepted to total messages, the more reliable the entity. We define recommended trust in terms of deferring to the closest entity that can offer a judgment, in the case of a module that is the parent trust context. In the case of a trust context, we defer to the trust established with the trust context’s parent. To enhance the reliability of the trust

establishment, we set a minimum number of messages as a threshold before which we presume that direct trust cannot be trusted. In this way, we guard against malicious modules that would be considered trusted in the initial stages of the system, as they might behave non-maliciously for a small number of messages.

Let $\gamma_a(e_0, e_1)$ return the number of messages successfully sent from e_0 to e_1 , i.e., the messages that have not been rejected by the context controllers. Let the function $\gamma_s(e_0, e_1)$ return the number of all messages sent from e_0 to e_1 , including those that have been rejected by the context controller. Furthermore, γ_T is a message threshold and the function $\text{Cntxt}(e)$ returns the immediate trust context to which the entity e (i.e., a module or a context) belongs.

The trust function is then defined as follows:

$$\text{Trust}(e_0, e_1) = \begin{cases} \frac{\gamma_a(e_0, e_1)}{\gamma_s(e_0, e_1)} & \{e_0, e_1\} \subseteq \mathcal{M}, \gamma_s(e_0, e_1) \geq \gamma_T \\ \text{Trust}(\text{Cntxt}(e_0), e_1) & \{e_0, e_1\} \subseteq \mathcal{M}, \gamma_s(e_0, e_1) < \gamma_T \\ \sum_{e_i \in \mathcal{C}_0} \frac{\gamma_a(e_0, e_i)}{\gamma_s(e_0, e_i)} \frac{1}{|\mathcal{C}_0|} & e_0 \in \mathcal{T}, e_i \in \mathcal{M} \cup \mathcal{T}, \gamma_s(e_0, \text{Cntxt}(e_1)) \geq \gamma_T \\ \text{Trust}(e_0, \text{Cntxt}(e_1)) & e_0 \in \mathcal{T}, e_1 \in \mathcal{E} \cup \mathcal{T}, \gamma_s(e_0, \text{Cntxt}(e_1)) < \gamma_T \\ \text{undefined} & \text{Otherwise} \end{cases} \quad (3)$$

In some cases, for instance, when there is only a single module that is vital for the system's operation but that component behaves maliciously, e.g., in the case of a compromised safety-critical component such as a mission control module in a flight craft or when the confidentiality or integrity of an application data are not of high importance and thus functionality takes precedence over the application's security, a choice needs to be made between breaking the system's functionality or tolerating a security impact to keep the system functional. In the latter case, the trust between entities needs to be modified to satisfy the minimum criteria for functionality. Therefore, the trust between entities is calculated according to our trust function and the tag trust values and context threshold values are updated as normal, but the system operates slightly differently. The trust values v of tags participating in messages to and from critical modules are modified to meet the minimum values required for messages to succeed. If the system is updated with trustworthy modules that meet the same functionality as the critical ones, then the malicious nodes can be supplanted by trusted ones, improving the system security. Selecting functionality over security should be decided carefully, as the misbehaving critical components are able to exfiltrate or modify data passing through them without security policies being enforceable. Since the messages to and from critical components are never dropped, there is no way to block messages that could lead to inferring information implicitly.

As the system execution progresses, trust in the system evolves depending on each entity's behavior. When the policies of modules are respected, their mutual trust increases, while when the policies are not respected, the mutual trust decreases. Each trust context is responsible to ensure a reasonable level of trust among its elements. We define two notions, *inner* and *outer* trust, which correspond to the trust within a context and trust toward entities not in the context, respectively.

We define inner trust as the average trust of each context's members to each other, i.e.,

$$\mathcal{I}(t) = \frac{\sum_{e_i, e_j \in \mathcal{E}, e_i \neq e_j} \text{Trust}(e_i, e_j)}{|\mathcal{E}|}, \quad (4)$$

where $t = \langle c, \mathcal{E} \rangle$ and $|\mathcal{E}|$ shows the number of elements in $\mathcal{E}$. Inner trust serves as a measure of how tightly the entities in a trust context trust each other. Since the trust contexts are formed with this goal in mind, inner trust is used to evaluate the suitability of a trust architecture in our adaptation decisions (see Section 5.3).

Outer trust is defined as the average trust of each context's elements to entities or contexts of the other contexts, i.e.,

$$O(t) = \frac{\sum_{e_i \in \mathcal{E}, e_j \notin \mathcal{E}} \text{Trust}(e_i, e_j)}{|\mathcal{E}|}. \quad (5)$$

We define the average trust for each trust context that serves as an indicator of how well a trust context performs in terms of complying to AT-DIFC⁺'s security goals. Average trust is defined as

$$\mathcal{A}(t) = \frac{I(t) + O(t)}{2}. \quad (6)$$

Entities in the same or sibling trust contexts have the lowest communication overhead and the highest amount of mutual trust. Elements further away from each other have a higher communication cost and are expected to have lower mutual trust, as they are not expected to communicate with each other often. A high degree of overall inner trust entails confidence in the system's often occurring and tightly grouped interactions, where a high amount of trust between the participants is expected. A high degree of outer trust entails a high degree of confidence in the rarely occurring or lowly trusted interactions where there is a low trust or uncertainty toward the particular amount of trust to be placed on the interaction.

4.4 Security Policies

The basic security policies enforced by the system are defined in terms of *safe label alteration* and *safe message passing*. Tags incorporate the notion of trust and have a maximum travel distance, thus the safe operations have to take these aspects into account. Moreover, since no central tag authority capable of controlling tag creation, deletion and update exists in our model, an additional operation is required to revoke a tag that is no longer valid. A tag can be invalidated due to either (i) a security policy change that results in the tag's secret information being declassified or (ii) due to the tag's owner leaving the system. We call the operation that revokes a tag, i.e., prohibits the usage of a tag in the system, a *tag revocation* operation.

Label changes are required to enable communication and facilitate endorsement or declassification. When a module needs to send some piece of its secret information, the tag corresponding to that information should be added to its labels. After a label change succeeds, the label's corresponding interface version signature is also communicated with the module updating its known tags. Therefore, version signature synchronization after an interface's policy changes is achieved by label changes for all communicating modules that invoke an interface. We say a label change from l to l' is *safe* for a module $m \in \mathcal{M}$ iff

- $\{l - l'\}^+ \cup \{l - l'\}^- \in \mathcal{P}_m$,
- $\forall \langle o, id, v, d \rangle \in l', v \leq \text{Trust}(m', m)$ and $\text{Dist}(m', m) \leq d$,

where $\mathcal{P}_m$ is m 's capability set, m' is the tag's owner, and Dist is a function that returns the distance between two system entities (number of visited context controllers to reach each other).

Informally, this states that a label change is only permitted *iff* a module has the capabilities to make the required changes, and for each tag in the new label, the module is trusted by the tag's owner and the tag has not flowed too far away. The tag's owner, i.e., the module that created the tag is the only actor that is allowed to assign capabilities related to that tag.

Example 2. In order for the Frontend microservice to be able to invoke the pay interface of the Payment microservice successfully, either amount is already in its labels, or a label changing operation is required to add it. Formally, this is stated as $\text{amount} \in \mathcal{P}_{\text{Frontend}}$, i.e., Frontend has the capability to add amount to its labels, $0.8 \leq \text{Trust}(\text{Payment}, \text{Frontend})$ and $\text{Dist}(\text{Payment}, \text{Frontend}) \leq 1$.

In our model, there exists no concept of an isolated message, rather the modules communicate via invocations. An Invocation can be expressed either as a single or as two messages depending on whether the invoked interface has a return value or not. If an interface has a return value, then the caller is also able to specify the security properties of the expected return value by adding that tag to its labels.

The second condition for a secure system is that all invocations between modules of contexts are *safe*, i.e., the interface version signature of an invocation matches the version signature of the called interface and all messages that belong to invocations are *safe*. Informally, a message is *safe* when the sender already knows about the tags of the receiver, and furthermore, for each tag in l the receiver is trusted enough by the sender and the tag has not flowed too far. A message sent by a module or context m_i with the label l_{m_i} to another module or context m_j on interface i with label l_{m_j} is safe iff:

- (1) $l_{m_i} \subseteq l_{m_j}$, and
- (2) $\forall \langle o, id, v, d \rangle \in l_{m_j}, v \leq \text{Trust}(m_i, m_j)$ and $\text{Dist}(o, m_j) \leq d$.

Example 3. In our running example, an invocation from Item to Payment calling the $\text{pay}(\text{user}, \text{amount})$ interface that returns paid, considering that Item knows $s_{\text{Item_pay}}$ to be the version signature of pay, will be permitted, iff:

- $s_{\text{pay}} = s_{\text{Item_pay}}$
- $\text{user} \in l_{\text{pay}}, 0.4 \leq \text{Trust}(\text{Item}, \text{Payment}), \text{Dist}(\text{Item}, \text{Payment}) \leq 1$,
- $\text{amount} \in l_{\text{pay}}, 0.8 \leq \text{Trust}(\text{Item}, \text{Payment}), \text{Dist}(\text{Item}, \text{Payment}) \leq 1$, and
- $\text{paid} \in l_{\text{pay}}, \text{value}(\text{paid}) \leq \text{Trust}(\text{Payment}, \text{Item}), \text{Dist}(\text{Payment}, \text{Item}) \leq 1$.

That informally states that in order for an invocation from Item to Payment invoking the pay interface to be safe, Item and Payment should trust each other, they should agree on which version of the interface is being invoked and the tags in their pay label must have their constraints satisfied.

To perform tag revocation, the direct context controller of the module that owns the tag sends a *tag revoked* special DIFC message to all direct context controllers of modules that hold any capability over the tag to be revoked. Each such controller, in turn, transmits a *tag revoked* message to its controlled modules that hold any capabilities over the revoked tag. The same controllers are additionally responsible to enforce that no messages with the revoked tag will be accepted. Therefore, each controller holds a list of revoked tags, that is checked whenever a message is sent from or verified by the controller, and on each label change.

4.5 Security Weaknesses and Mitigations

The AT-DIFC model presented in Reference [22] was susceptible to a few attacks that take advantage of the decentralized and open nature of the system. These attacks can be categorized into three attack types.

The first type of attacks aimed to exploit the loose definition of tag ownership in our previous model. Since tag ownership was not explicitly encoded in our model, any module could claim ownership over any tag, including those it did not create and therefore should not have control over. Since tag ownership grants a module the privilege to assign capabilities related to the owned tag to any other module, the malicious module could gain partial control of how the particular tag was used in security policies throughout the system. Note that complete control would be difficult

due to the fact that the malicious entity would be competing with the legitimate tag owner module in terms of setting the tag values. Moreover, since we did not consider the tag lifetime in our previous version of the model, a module could claim ownership of tags that belonged to modules that have left the system and therefore should have been invalidated. The illegitimate ownership of tags could then be exploited twofold. (i) If the tag was valid, i.e., the entity that created it was still in the system, then the malicious module could assign any capabilities to itself, thus defeating DIFC mechanism, as it would be able to learn the secret information associated with a secrecy-related tag or compromise the integrity of critical information associated with an integrity-related tag, by modifying it without permission. (ii) If the tag was invalid, since some modules would still hold a reference to that tag in their labels, then the attacker could add that tag to the labels of modules s/he controlled and use it in communication that would result in DIFC policy violations. The DIFC violations entail a decrease in the trust of modules that would call the interfaces using the expired tag in their labels.

To mitigate attacks related to tag ownership, we first introduce and enforce a new DIFC operation that we call tag revocation and was explained in Section 4.4, and, second, add the owner information as an explicit part of the model's tags. As a result, any capability assignment or revocation attempt by a module that does not own a particular tag results in an illegal operation that is penalized by our model. To further strengthen our model, we implemented the opaque tokens that represent owners and tag ids as cryptographically secure hashes to eliminate the possibility of attacks that rely on guessing them.

The second and third type of attacks are connected and revolve around exploiting valid system operations to either affect the trust of modules that invoke interfaces of a malicious process, or generate a large number of control message traffic in an attempt to slow down DIFC enforcement or starve critical system resources. Defaming attacks would attempt to alter the label of an interface while a message that is part of an invocation to that interface was in the process of being transmitted. This would result in the violation of the callee policies, and consequently, lowering the caller's trust. To mitigate this type of attack, we introduce and update a version signature for each interface. When an invocation's version signature does not match an interface's version signature, a special *retry* DIFC message is sent to the caller, which forces a label change and the message is retried once the version signature has been updated. In continuation of this attack, the malicious process could attempt to continue to repeatedly change its interface labels, aiming to flood the system with *retry* messages, i.e., this leads to either starving the system resources or degrading the performance of DIFC mechanism. To mitigate this type of attack, we introduce a maximum acceptable rate of label changes and penalize modules that exceed it.

4.6 Security Goals

Trust between entities is formed based on their history of conformance to security policies. As a result, trust is dynamic and depends on the behavior of modules. If the trust contexts remain fixed, then the rate of permitted information flowing through the system may drop significantly, as highly trusted entities might be in far away contexts as opposed to within the required distance for their messages to be accepted. Furthermore, since tags have a limited scope on how far they can flow, valid invocations of the system will be blocked. Therefore, as trust in the system evolves, the architecture of the trust contexts is required to adapt accordingly to allow more secure information flow, hence, the trust architecture becomes dynamic itself.

Our main security goal is to enhance the permitted flow of information in the system, while upholding decentralized information flow control and enforcing security. To achieve the most permissible system, we need to dynamically restructure the trust architecture to maximize the average trust, as far as we can. Therefore, we define the system's security goals as the following:

- **Goal 1.** The system should maximize the overall average trust, formally:

$$\max \sum_{t_i \in \mathcal{T}} \mathcal{A}(t_i). \quad (7)$$

- **Goal 2.** The system must be able to enforce decentralized information control flow and minimize the system's blocked functionality.
- **Goal 3.** The system must be able to identify and handle malicious modules or trust contexts.

Our first goal aims to improve both the system's permissiveness and the performance of DIFC evaluation in the trust architecture. Average trust is a composite metric that is affected by how well an entity adheres to the AT-DIFC⁺ policies of entities within its context (inner trust) and to the policies of entities outside its context, with which it communicates with (outer trust). Thus, average trust represents how well a context's elements summarily comply to AT-DIFC⁺, and is therefore a reasonable candidate for optimization to maximize AT-DIFC⁺ enforcement.

As the average trust in the system increases, entities that share high amounts of trust with each other are moved closer to each other, that leads in reducing the distance between them, and allowing more functionality to be available to them. Thus, the permissiveness of DIFC enforcement improves as the modules come closer to each other, and consequently it becomes more likely that the distance values of tags in labels to be within the range. This allows more of their interfaces to be invocable that can increase the functionality available to modules in the system. The throughput also can increase, as the DIFC messages need to traverse a smaller part of the system to reach the common ancestor that will do the enforcement. For example, if the entities belong to the same context, then that context controller can perform DIFC enforcement directly. When the average trust decreases, then the distance between elements that do not trust each other increases, as they are placed further away from each other. As a result, sensitive information cannot be propagated to untrusted entities as the distance criteria of tags in entities' labels will not be met. That entails an improvement to the system security, as fewer entities interfaces are invocable by malicious or low-trusted modules.

The first part of our second goal is guaranteed by the AT-DIFC⁺ model, i.e., it can be guaranteed that no messages that do not meet the security policies of entities will be transmitted. The second part of the goal, which corresponds to the minimization of the blocked functionality is achieved based on the underlying trust system. If an entity behaves non-maliciously, then it will be placed closer to entities that it communicates with and also behave in a trustworthy manner. Since the closer two entities are, the more functionality becomes available to them, the blocked functionality becomes less as the average trust in the system increases and entities are dynamically moved closer to one another when they have a degree of mutual trust.

We also depend on the trust system to meet our third goal, i.e., to identify and handle malicious entities. Since average trust represents the extent to which AT-DIFC⁺ is respected by an entity, that implies that entities with a low average trust do not regularly conform to AT-DIFC⁺ policies. Therefore, such entities are placed further away from other entities limiting the functionality exposed to them. Moreover, we make sure to isolate malicious entities to ensure that critical system information is not leaked to untrusted actors.

5 RECONFIGURATION IN AT-DIFC⁺

The system's principle security goal is to organize the trust context hierarchy, so that the information flow in the system is maximized, whilst prohibiting information flow to/from untrusted or malicious actors. Information flow in the system is intrinsically linked to trust, as information is only permitted to flow toward trusted modules. As a result, we may define our approach's goal:

to increase the trust between the system's modules and contexts as much as possible. To achieve that, we require a reconfiguration mechanism. We elaborate on the reconfiguration strategy we employ below.

5.1 Trust Architecture Reconfiguration

To achieve Goal 1 and Goal 2, we restructure the trust architecture in a way that increases the information flow in the system. To this end, we identified three reconfiguration actions:

- *Context split* that will split a context into two parts: a part in which there is a high amount of trust within its members, and a part where the trust is lower.
- *Contexts merge* that enables two contexts that have a high mutual trust relation to fuse into one.
- *Context or module isolation* that segregates a malicious entity from the system and prohibits messages that carry tags to it. Messages from the entity are not blocked, as they will be dropped if they do not respect the receiver's policies.

Every reconfiguration action is associated with one or more threshold variables that are tuned dynamically to achieve our security goals. We denote the merge threshold by Tr_m and the split threshold by Tr_s . Isolation is dependent on three thresholds that include the DIFC violation threshold Tr_v , the illegal operation threshold Tr_i , and the potentially malicious operation threshold Tr_p . When the merge condition of two sibling contexts is met, i.e., $\mathcal{A}(t_1, t_2) \geq Tr_m$, then t_1 and t_2 are merged. Conversely, the condition $\mathcal{I}(t) \leq Tr_s$ entails a context split for the context. An isolation is performed in three cases: (i) when the average message success history is consistently lower than Tr_v , i.e., when an entity consistently sends messages that violate AT-DIFC⁺ policies; (ii) when the frequency of its security policy changes is too high, that is, when an interface alters its labels more frequently than the threshold Tr_p ; and (iii) when the number of attempted illegal operations surpasses Tr_i . An operation is illegal if a module has no rights to perform it. Illegal operations comprise tag revocation or capability assignment related to a tag that the module does not own.

In effect, thresholds act as the conditions where action has to be taken to improve the system state. That can happen either when some contexts act in a more trustworthy way than initially expected, resulting in them being merged, or the opposite that results in a context split at first and then an isolation. To be able to evaluate whether each threshold is violated, we store the required information in the context controllers. This information, for each entity e , includes (i) its trust to each entity it communicates with; (ii) the ratio of successful to unsuccessful messages, which we denote as $history(e)$; and (iii) the number of illegal operations attempted, which we denote as $illegal(e)$, and the frequency of policy changes, which we denote by $policy_change(e)$. Additionally, we define $trusted(e)$ to be a function that returns the list of entities an entity e has communicated with.

5.2 Reconfiguration Plan Generation

Algorithm 1 describes reconfiguration plan generation. A plan consists of three sets of operations: merge, split and isolate. The operations are planned based on the order by which they are scheduled, denoted by m_o , s_o , and i_o respectively. A module or trust context can only be part of one reconfiguration operation. Enforcing this principle, allows the presented algorithm to generate a deterministic plan where the planned operations are guaranteed to be executed as planned. To achieve that, we maintain the set of the entities that have already been planned for, i.e., scheduled for a reconfiguration operation and an entity is only a candidate for an operation if it is not an element of that set. In the algorithm, the value of op_order shows the current operation being planned according to the order of m_o , s_o , and i_o . On each loop, the thresholds for

the current reconfiguration operation are checked for all context elements, and if the thresholds are violated, then the entity is placed in the corresponding set of planned operations and the set of scheduled entities. When all reconfiguration operations with that order have been considered, *op_order* is incremented and the next operation is considered.

Given the different possible orderings of the reconfiguration operations and the order in which the trust context elements are examined, Algorithm. 1 can generate different plans. We run Algorithm 1 multiple times, providing all combinations of values for m_o , s_o , and i_o as input, to obtain all possible reconfiguration plans. The two constraints described above, i.e., enforcing that an entity can take part in a single reconfiguration action in a plan and forcing an ordering on the reconfiguration actions allow us to keep the number of plans manageable and additionally allow us to generate deterministic plans. The number of plans depends on the number of entities that satisfy the threshold for each operation but will not, in the worst case, exceed the possible combinations of assigning each entity to every unique reconfiguration action. Such a large number of plans, however, is not too likely since it would require that every entity be eligible for every reconfiguration operation at the same time. Merging occurs amongst more trusted entities, while splitting happens amongst entities with low trust. This will make it less likely that an entity will be a candidate for both a merge and a split operation in the same plan. Further, an entity is isolated when it is consistently misbehaving over a time period and thus its trust will be lowered, which makes it less likely to be eligible for a merge operation at the same time. To select among the possible plans, we propose a utility-based approach that is discussed in Section 5.3.

5.2.1 Context Split. A context split is performed when there is a low amount of trust among its members, leading to separation between malicious or misbehaving members and trusted parts of the context, thus improving the flow of information. When the average inner trust of a non-atomic module drops below Tr_s , a context split is required.

To separate a trust context into two partitions with a high amount of inner trust, we apply a modified version of the Fiduccia–Mattheyses graph partitioning algorithm [6]. In each iteration, the Fiduccia–Mattheyses algorithm selects a node to move from one partition to the other by finding a more optimal cut of the graph. That is achieved by calculating a heuristic gain and maintaining the partition balance. The gain is typically calculated as the difference between the moving force, i.e., the number of edges what will be kept if the node moves from one partition to the other, and the retention force, i.e., the number of edges that will be removed when the node is moved. A balancing condition, e.g., that both partitions should have approximately the same number of nodes, is introduced to ensure that all nodes are not placed in one partition. The selected node is the node with the maximum gain that does not break the balance between partitions if moved. Moving a node to a partition fixes it in place, meaning that the node cannot be moved anymore. Once all nodes have been fixed, the algorithm terminates. The context split is performed by Algorithm 2.

The context splitting process is performed in five steps:

- (1) *Graph creation:* An undirected weighted graph is generated from the context's elements. Each element corresponds to a node and edges correspond to the trust relationship between two nodes. The weight of an edge (e_i, e_j) is defined as $\mathcal{W}(e_i, e_j) = \frac{\text{Trust}(e_i, e_j) + \text{Trust}(e_j, e_i)}{2}$.
- (2) *Graph partitioning:* We employ the Fiduccia–Mattheyses algorithm with two minor changes: we consider $\mathcal{W}(e_i, e_j)$ in gain computation and aim to maximize the gain, as the edges now correspond to trust between entities. Our gain function *gain* is computed as the average inner trust of the remaining nodes in one partition, once a node is moved to the other partition and the average inner trust of the nodes in the other partition once the node is moved to it. Note that $\mathcal{I} = \frac{\sum \mathcal{W}(e_i, e_j)}{|\mathcal{E}|}$. The balancing condition is also simplified to be the

ALGORITHM 1: Reconfiguration Plan Generation Algorithm

Input: The trust context $t = \langle c, \mathcal{E} \rangle$ to be reconfigured, m_o, s_o, i_o , the order of reconfiguration operations

Output: A plan consisting of sets of merge, split and isolation actions (mergeAc, splitAc, isolateAc)

mergeAc = $\emptyset$, splitAc = $\emptyset$, isolateAc = $\emptyset$;

// scheduled is a set that contains entities already part of the plan

scheduled = $\emptyset$;

// op_order shows the order of the current reconfiguration action

op_order = 0;

while op_order < 3 **do**

for $e_i \in \mathcal{E}$ **do**

if op_order = $m_o \wedge e_i \in \mathcal{T} \wedge e_i \notin \mathcal{T}_a \wedge e_i \notin \text{scheduled}$ **then**

for $e_j \in \mathcal{E}$ **do**

if $\mathcal{A}(t_1, t_2) \geq \text{Tr}_m \wedge \mathcal{A}(t_2, t_1) \geq \text{Tr}_m \wedge e_j \notin \mathcal{T}_a \wedge e_j \notin \text{scheduled}$ **then**

 mergeAc = mergeAc $\cup \{(e_i, e_j)\}$;

 scheduled = scheduled $\cup \{e_i\} \cup \{e_j\}$;

end

end

end

if op_order = $s_o \wedge e_i \in \mathcal{T} \wedge e_i \notin \mathcal{T}_a \wedge e_i \notin \text{scheduled}$ **then**

if $I(e_i) < \text{Tr}_s$ **then**

 splitAc = splitAc $\cup \{e_i\}$;

 scheduled = scheduled $\cup \{e_i\}$;

end

end

if op_order = $i_o \wedge e_i \notin \mathcal{T}_a \wedge e_i \notin \text{scheduled}$ **then**

if $\text{history}(e_i) \leq \text{Tr}_v \vee \text{illegal}(e_i) \geq \text{Tr}_i \vee \text{policy_change}(e_i) \geq \text{Tr}_p$ **then**

 isolateAc = isolateAc $\cup \{e_i\}$;

 scheduled = scheduled $\cup \{e_i\}$;

end

end

end

 op_order = op_order + 1;

end

difference between the number of nodes of each partition. The balancing criterion is met when both partitions are of exactly equal size, or their size differs by exactly 1. The resulting two partitions correspond to two partitions that are more strongly connected in terms of trust between their entities.

- (3) *New context creation:* We create a new trust context t' and move the elements of the partition with the lower trust to t' .
- (4) *Connectivity repair:* t' is connected with t 's parent and all messages that are targeting entities that are now residing in t' are redirected to it.
- (5) *Trust recalculation:* The trust of t and t' in addition to their parent's trust are recalculated.

Example 4. Let us consider splitting the ItemShop context of our running example. After computing $\mathcal{W}$, the resulting graph is shown in Figure 3(a). We will call the two graph partitions *partA* and *partB*. Initially the whole graph can be considered to be in *partA*. In each iteration, the gain of each node is computed by removing that node from a partition and computing the average

Fig. 3. Split Algorithm Example.

of the remaining trust weights. The results for the first iteration are as follows: $gain(A) = 51.6$, $gain(I) = 55$, $gain(F) = 55$, $gain(C) = 56.6$. As a result, the Cart module is selected to be moved and fixed to partB. The resulting partitions are shown in Figure 3(b). Calculating the balancing criterion yields $|\mathcal{E}_1| - |\mathcal{E}_2| = 3$, and since the criterion is not met, we continue iterating. In the second iteration, we fix A as it has yields the maximum gain. After checking balancing criterion again, partA and partB are balanced and therefore our algorithm terminates. The final state is shown in the rightmost part of Figure 3.

After a split, we expect the system's average trust to increase, as there now exist two contexts, the components of which are tightly trusted, instead of one context containing loosely trusted elements. Furthermore, since the less trusted elements have been separated from the highly trusted ones in a separate context, they can be merged with contexts they are highly trusted by.

5.2.2 Context Merge. A merge is performed when the members of two sibling contexts are highly trusted. In that way, the rate of information flow in the system increases, as one less step is required for the communication of the merged contexts' elements. Furthermore, previously prohibited interactions with tags that were out of reach become available.

Two sibling contexts, i.e., contexts with the same controller, are merged when their mutual trust exceeds the trust threshold, formally: $\mathcal{A}(t_1, t_2) \geq Tr_m \wedge \mathcal{A}(t_2, t_1) \geq Tr_m$. The merging process consists of three steps: (1) all elements of t_2 are moved to t_1 , (2) the trust between all elements that now reside in t_1 is recalculated, and (3) the common parent is updated to redirect messages to elements that formerly belonged to t_2 toward t_1 . Finally t_2 is discarded.

Example 5. To merge the Bank and Shipping Service trust contexts, the Shipping and Logistics modules are moved to the Bank context. Then, the Shipping Service context is deleted and all messages to it are redirected to the Bank. The global context would therefore contain two contexts: the ItemShop and the Bank.

5.2.3 Module or Context Isolation. Isolation ensures that AT-DIFC⁺'s security policies are enforced, as secret information cannot escape the system or be leaked to untrusted actors in the system. The malicious behavior is segregated and cannot spread to the rest of the system. When the rate of accepted messages sent by an entity drops below Tr_v , the frequency of its interface label changes is greater than Tr_p , or the number of its illegal operations surpasses Tr_i , that entity is marked for isolation. If a module marked for isolation belongs to an atomic context, then the whole context is isolated. Critical modules, however, cannot be isolated. When an entity is isolated, messages sent to it are dropped or replaced with safe defaults, while the messages sent from that entity are handled normally. Allowing messages from a malicious entity does not affect

ALGORITHM 2: Trust Context Split Algorithm

```

Input:  $t = \langle c, \mathcal{E} \rangle$  a trust context to be split
Output:  $t'$  the new trust context that was split from  $t$ 
 $p = \text{context}(t)$ ;
// Step1: Calculate the trust weights and build the graph G
 $W = \emptyset$ ;
for  $e_i \in \mathcal{E}$  do
    for  $e_j \in \mathcal{E}$  do
        if  $e_i \neq e_j$  then
             $\mathcal{W}(e_i, e_j) = \frac{\text{Trust}(e_i, e_j) + \text{Trust}(e_j, e_i)}{2}$ ;
             $W = W \cup \{\mathcal{W}(e_i, e_j)\}$ ;
        end
    end
end
 $G = \text{create\_graph}(\mathcal{E}, W)$ ;
// Step2: Partition G into two parts partA and partB
// Initialize the array of fixed nodes
 $\text{fixed}[] = \text{init\_fixed}(G, \text{num\_fixed})$ ;
 $\text{partA} = G$ ;
 $\text{partB} = \text{empty\_graph}()$ ;
while  $\text{!balanced}(\text{partA}, \text{partB})$  do
    // n is the node with max gain to be moved and fixed
     $n = \text{calculate\_max\_gain}(\text{partA}, \text{partB}, \text{fixed})$ ;
     $\text{move\_node}(\text{partA}, \text{partB}, n)$ ;
     $\text{update\_weights}(\text{partA}, \text{partB})$ ;
    // fix n, cannot be moved again.
     $\text{fixed}[n] = \top$ 
end
// Step3: Create a new context from partB, remove the elements from t
 $t' = \text{create\_context}(\text{partB})$ ;
 $\text{remove\_elements}(t, \text{partB})$ ;
// Step4:
 $\text{redirect}(p, \text{partB}, t')$ ;
// Step5: Update trusts
 $\text{update\_trust}(t)$ ;
 $\text{update\_trust}(t')$ ;
 $\text{update\_trust}(p)$ ;
return  $t'$ 

```

the correctness of the system, as all messages that do not conform to the DIFC rules will be dropped by the corresponding controllers.

Isolation can be too restrictive, as well-behaving entities that share a trust context with malicious ones might be isolated, even though their behavior is non-malicious. To improve the system's permissiveness, these entities can be migrated to trust-contexts by which they are highly trusted. We call this process migration and incorporate it in the isolation reconfiguration operation. It should be noted that migration can be enabled or disabled based on user preferences.

The Isolation-Migration algorithm is shown in Algorithm 3. If migration is enabled and the entity to be isolated is a context, then the algorithm walks through each entity in that context and

attempts to migrate it to one of the contexts that this entity highly trusts. If the highly trusted entity is a module, then the algorithm checks whether that module's context trusts the particular entity enough by checking against the merge threshold, and if so, the element is added to that context. If the trusted entity is a context, then the check is performed against the context directly. If the entity in question is not trusted highly enough by any of its trusted entities, then the entity is isolated along with the context itself. The rest of the algorithm follows the isolation process, the entity is isolated and the messages sent to it are dropped.

ALGORITHM 3: Module Isolation-Migration Algorithm

```

Input: e a module or context to be isolated
Output: Nothing is returned
p=context(e);
if migration_enabled  $\wedge$  !isModule(e) then
  // since t is a context,  $\mathcal{E}$  is the set of its elements
  for  $e_i \in \mathcal{E}$  do
    for  $e_j \in \text{trusted}(e_i)$  do
      if isModule( $e_j$ )  $\wedge$  Trust( $e_i, \text{context}(e_j)$ )  $> \text{Tr}_m$   $\wedge$  Trust( $\text{context}(e_j), e_i$ )  $> \text{Tr}_m$  then
         $p_j = \text{context}(e_j)$ ;
        add_element( $p_j, e_i$ );
        break;
      end
      else if Trust( $e_i, e_j$ )  $> \text{Tr}_m$   $\wedge$  Trust( $e_j, e_i$ )  $> \text{Tr}_m$  then
        add_element( $e_j, e_i$ );
        break;
      end
    end
  end
end
isolate(p,e);
drop_messages_to(p,e);

```

5.3 Utility-based Reconfiguration Evaluation

There exist multiple plans that can steer the system toward maximizing trust, each of which may lead the system to a different configuration state. Since the behavior of the entities in our system is dynamic and the system is open, the best plan is very difficult to know in advance. Moreover, the system can only converge to an optimal average trust if the entities in the system remain constant and behave consistently. Given the uncertainty due to the system openness and the dynamicity in the behavior of its entities, we believe that a utility based approach is a viable strategy to address the problem of plan generation. Hence, we follow a utility-based approach to decide which of the available plans is more suitable.

We define two utility functions that guide the selection process by choosing a reconfiguration plan that leads to the maximum gain by determining how much the average trust of the system improves if an operation is performed. The local utility function only takes into account the local state of the context in addition to the immediately affected contexts by the change, i.e., it only considers a small part of the overall system state. On the contrary, the global utility evaluation inspects the whole system state.

Local Utility Evaluation. The function δ_m defines the gain of the merging two contexts, i.e., how much the average trust of a system improves locally, if we merge the two contexts t_1 and t_2 and obtain t , then

$$\delta_m(t_1, t_2) = \mathcal{A}(t) - \frac{\mathcal{A}(t_1) + \mathcal{A}(t_2)}{2}. \quad (8)$$

If the result is positive, then that means that merging t_1 and t_2 into t has a positive local effect on the trust, as the average trust improves. The function δ_s defines the gain of splitting a context, i.e., how much the trust of the system improves locally, if the context t is replaced by t_1 and t_2 as a consequence of the split operation defined in Section 5.1:

$$\delta_s(t_1, t_2) = \frac{\mathcal{A}(t_1) + \mathcal{A}(t_2)}{2} - \mathcal{A}(t). \quad (9)$$

As above, if δ_s is positive, then that means that splitting t into t_1 and t_2 has a positive effect to the system, as replacing t with t_1 and t_2 improves the overall average trust. Finally, the function δ_i is defined as the rate of blocked messages in the network if the processes or contexts of the proposed plan are isolated. This is always a positive effect to the system, as the entities transmitting the messages are untrustworthy or malicious and isolation reduces verification effort.

A local utility function $\mathcal{U}_{\mathcal{L}}$ only takes into account the trust context in which the selected adaptation plan executes. We assume that the adaptation actions are not conflicting, i.e., a context is only involved in one of the adaptations of a plan. Let $\mathcal{T}_m$ be the set of context pairs to be merged, $\mathcal{T}_s$ be the set of contexts to be split and $\mathcal{T}_i$ be the set of contexts to be isolated. We define the local utility function as Equation (10) where w_0 , w_1 , and w_2 are weights placed on each part of the plan based on the perceived importance of each of the merge, split and isolation actions, respectively. Weight assignment is domain specific and decided by the user.

$$\mathcal{U}_{\mathcal{L}} = w_0 \cdot \sum_{(t, t') \in \mathcal{T}_m} \delta_m(t, t') + w_1 \cdot \sum_{t \in \mathcal{T}_s} \delta_s(t) + w_2 \cdot \sum_{t \in \mathcal{T}_i} \delta_i(t). \quad (10)$$

Local Utility Evaluation. The global utility function $\mathcal{U}_{\mathcal{G}}$ takes the whole system's trust into account where $\mathcal{T}'$ is the set of trust contexts of the system after reconfiguration:

$$\mathcal{U}_{\mathcal{G}} = \sum_{t \in \mathcal{T}'} \mathcal{A}(t) - \sum_{t \in \mathcal{T}} \mathcal{A}(t) \quad (11)$$

In contrast to the local evaluation, global evaluation provides more accurate results but entails a higher computation time. Depending on the number of entities and the number of available plans, global evaluation might have a prohibitive cost. However, if security is of the utmost importance, then local evaluation is not suitable, as it is incapable of seeing the whole picture with regard to system security. The best decision in terms of selecting the most applicable utility function of the two is application specific, as it depends on the balance between system response-time and the security prioritization of the application. The global utility function caters to applications that require high security guarantees. On the contrary, an application that does not require a high level of security but prioritizes throughput would opt to use the local utility evaluation function.

6 IMPLEMENTATION

This section details our AT-DIFC⁺ reference implementation that can be found at: <https://github.com/LNU-Prosses/ATDIFC>.

6.1 Architecture

Figure 4 shows the architecture of an example system that we will use to describe our reference implementation. Each module communicates with a shim, a short piece of code that interposes

Fig. 4. Reference Implementation Architecture for AT-DIFC, depicted: Modules, Shims, and Trust Controllers.

message communication and is responsible to facilitate all AT-DIFC⁺ interactions with a module. The shim communicates with the module's immediate trust controller over TCP to facilitate all operations that require communication with other modules. Each trust controller, in turn, communicates over TCP with its children and parent context controllers. Every module and its shim have to be co-located on the same host machine so that the shim can have access to the module's operations. That is required for the shim to be able to interpose and drop messages that do not conform to AT-DIFC⁺ policies.

6.1.1 Shim API. The shim provides a uniform domain agnostic interface with the rest of the AT-DIFC⁺ architecture. It provides AT-DIFC⁺ capabilities to the underlying module infrastructure independently of the technology used to implement the decentralized system. However, the connection and interaction between the shim and the module remains domain specific. There are two parts that need to be implemented for each type of decentralized system to facilitate AT-DIFC⁺, message interposition and a short stub that allows the module to invoke the shim's API. The shim API features a set of functions that are exposed to each module to perform AT-DIFC⁺ operations including `create_tag`, `assign_cap`, `create_label`, `label_change`, and `send_message`.

6.1.2 Trust Controller API. Every AT-DIFC⁺ controller is associated with a MAPE-K Loop [9] as shown in Figure 5. Each controller is responsible to both enforce AT-DIFC⁺ and improve the trust of the underlying elements it controls. The functionality can be described by an API that can be split into the three parts shown in the figure. The first part of the API is responsible for all with DIFC enforcement operations and is situated on the monitoring and analysis phases of the MAPE-K loop. This part is responsible to receive, route and verify messages, facilitate label changes, and revoke tags. The second part of the API deals with trust calculation and maximization, it is situated in the analysis phase and is responsible to update the history and trust of modules, and to calculate the various types of trust of trust contexts. The final part part of the API deals with reconfiguration, it is responsible for reconfiguration planning and execution of reconfiguration operations. It is situated in the planning and execution phases of the MAPE-K loop.

6.2 AT-DIFC⁺ Initialization

On system initialization, the modules are grouped based on their owner preferences, and AT-DIFC⁺ controllers are formed based on the owner or stakeholder specification. Each module is accompanied by a set of security policies, which form interface labels and a list of trust values for the modules it initially communicates with. The threshold values for each trust context are initially

Fig. 5. The Trust Controller APIs mapped to a MAPE-K architecture.

set by the owner or stakeholder and updated dynamically after each reconfiguration operation. When a new module arrives at the system, either it is claimed by a stakeholder, in which case it is placed in the trust context where the stakeholder's modules reside, or a new trust context is created for it. Beyond their initial context placement, there is no difference between the entities claimed by stakeholders and those that are not; all entities are treated equally by the system based on their behavior in terms of respecting AT-DIFC⁺

The initial values for each threshold are inherently domain specific. The threshold values depend on the nature of the data exchanged within the context and the extent to which it is acceptable for invocations to be limited if a context split occurs. For instance, thresholds that belong to a trust context that handles highly confidential data such as credit card or patient record information, would be assigned high values for merging and splitting. Furthermore, the tags of the modules in that context would be assigned very small distance values. Such an assignment of thresholds and distance values indicates a preference of protecting the confidentiality of the data over providing functionality that might cause leaks. With such an initial value placement, splits will preserve the confidentiality of data but will affect the system functionality, as modules that end up in different contexts after the split might not be able to send messages to each other. Contexts with a less strict policy with regards to data confidentiality or integrity, would set lower thresholds that shows a preference for achieving a higher rate of message exchange over limiting the risk of leaks.

6.3 Message Handling

Messages are initiated by the AT-DIFC⁺ shims and propagated through the AT-DIFC⁺ controllers until the first common ancestor of both the sender and the receiver is reached. To achieve that, the direct trust context controller that controls the sender invokes `has_module`. If `has_module` returns true, then the corresponding labels are fetched by `fetch_label` and `verify_message` is invoked on both the sender and receiver labels. Otherwise, the message is propagated upward to the current controller's parent controller until `fetch_label` succeeds or no parent controller exists, in which case the message is dropped and a DIFC violation is flagged. `fetch_label` has the additional responsibility of checking if the returned label contains any of the revoked tags; if that is the case, then the label is considered as invalid and is not returned.

6.4 Secure Hashes

A number of elements in our system are required to be cryptographically secure, i.e., they should be improbable to guess by an adversary, to ensure that a malicious module cannot impersonate another or be able to modify the policies of an interface or label that it does not own. These elements include, the tag owner, the interface version and the tag ID. We use the SHA-3 algorithm [5] to generate secure hashes for those elements. For the tag ID and interface version elements,

Fig. 6. The AT-DIFC⁺ decentralized control loop operations per MAPE-K phase.

once the hashes have been created, they can be shared in the network and used to identify the corresponding piece of information directly. We store a map from a generated module hash to each module in that module's direct context controller. In this way, we avoid leaking the ownership of tags to intermediate controllers or non-owner modules, which do not need know about modules they do not control or communicate with. When label or tag information has to be retrieved, the direct controller looks at the module hash map and communicates with the corresponding module. In all other instances of controller to controller or controller to module communication, the shared information always corresponds to the module hash that does not reveal any information about its origin or functionality.

7 AT-DIFC⁺ AND DECENTRALIZED CONTROL

In the context of self-adaptive systems, decentralization refers to the distribution of the mechanism that controls self-adaptation decisions. There are several patterns for decentralized adaptation with different characteristics that match various adaptation goal types and system decompositions [27].

Figure 6 depicts AT-DIFC⁺ activities mapped on a MAPE-K conceptual architecture. An AT-DIFC⁺ adaptation controller manages the trust architecture, enforces AT-DIFC⁺'s security policies and forwards messages. The knowledge base holds information about the trust and message history for all members in a controller's current context, including the thresholds for splitting, merging and isolation, and the context's current inner and outer trust values. The monitoring mechanism intercepts messages (1) and decides if they should be locally processed or delegated to the parent context, i.e., (2). The analysis enforces AT-DIFC⁺ rules (3), and updates the trust (4) and historical data (5) in the knowledge base. Planning (6)–(9) and the corresponding execution (10) and (11) vary depending on the feedback loop architecture. This is covered in detail below for each of the demonstrator architectures. In general, planning identifies need and plans the required structure changes while the execution enacts the plan on the current context and sub-contexts.

7.1 Decentralized Feedback Loop Demonstrators

Hierarchical. The hierarchical architecture, depicted in Figure 7, assigns each trust context in the hierarchy the full responsibility for the adaptation of all context elements that it encapsulates. The architecture forms a tree of trust contexts, in which each context controller controls its sub-contexts. The adaptation strategy is that each level optimizes the flow of trust in the layer below. Besides, each level initiates its MAPE-K operations once the layer below has completed. This strategy allows the controller to have a complete view of the state of the controlled sub-hierarchy. The

Fig. 7. The hierarchical demonstrator feedback loop.

adaptation strategy requires bottom-up coordination, that is, each feedback loop completes its operations and then updates its parent, which triggers adaptation at the parent level. Once all controlled elements in a context have completed, the top-level context controller runs its adaptation loop. This process continues until all top-level contexts have finished their execution.

Adaptation planning in a hierarchical structure works as follows. Each context attempts to (i) identify if merge, isolation, or split operations are required. The identification is described in detail in Section 5.1. (ii) We combine the identified operations into candidate plans. Each candidate is (iii) evaluated by one of the two utility functions (local or global) defined in Equation (10) and Equation (11). Finally, in (iv), the planner forwards the best fit the plan for enactment to ⑨.

Hybrid planner. The second demonstrator is a hybrid architecture, which we depict in Figure 8. The hybrid architecture combines the regional planner and master–slave patterns.

The master–slave pattern creates a hierarchical relationship between a centralized master component that is responsible for the analysis and planning and multiple slave components that are responsible for monitoring and execution. The implication is that the analysis and adaptation logic (planning) are decided globally, while monitoring and adaptation enactment are preformed locally. It is common to structure systems in loosely coupled subsystems, where each subsystem has local goals. The regional planner pattern mirrors this structure for the adaptation control structure. The weakly coupled components (regions) perform adaptations with little or no impact on other regions. However, the regional control mechanisms benefit from coordination and information exchange to contribute to systemwide adaptation goals. The hybrid planner architecture is often less effective in trust maximization but offers lower response time.

The top level is a single planner that selects a region—a set of trust contexts—to adapt, plans, and adapts the region. The planner uses the trust contexts in the region below that manages the modules. The levels form several master–slave planners where one analysis and one planning component handles each region.

Planning is somewhat more complicated in a hybrid setting due to the two levels at which the planner operates: First, a region is chosen to be adapted, and then region-specific adaptation decisions are made to optimize its average trust. Furthermore, this planner is capable of exchanging the trust contexts between regions when there is a high degree of interactions between these contexts. The top level selects which contexts master–slave loop to trigger, while the master–slave level evaluates the current region’s trust and optimizes it using a utility evaluation method. The region selection process is based on the activity of each part of the underlying architecture. The activity is measured in terms of messages that have been exchanged through a part of the architecture. The trust contexts that make up the region to be adapted are selected based on their *hotness*, i.e., how many messages have been propagated through them. A hotness threshold is set

Fig. 8. The hybrid demonstrator feedback loop.

and all contexts that meet that threshold are moved to the selected region. As a further optimization step, we traverse through the selected contexts and move *cold* contexts, i.e., contexts belonging to hot contexts that themselves do not meet the hotness threshold, out of the region.

One particular case at the top level is communication that crosses regions, i.e., when the planner can identify trust contexts that belong to different regions, but nonetheless feature a high-enough degree of interaction to warrant them being moved to the same region and eventually merged if they are trustworthy enough. Bringing those contexts closer, improves the performance of DIFC enforcement, as fewer intermediate controllers are involved, and increases the permissiveness of DIFC, as interfaces whose policies contain tags with smaller distance values are now accessible. When such situations occur, the planner rearranges the regions^⑧. The rearranging operation is based on the same concept of hotness as region selection. Trust contexts that are consistently hot are identified, hinting to the fact that they are part of the same message chain, i.e., they form a sequence of messages that facilitates some part of the system’s functionality. Those contexts are brought closer together by performing a merge reconfiguration operation. The dual operation is also performed if a context is consistently cold while all of its siblings are hot, and that context is moved out of the particular parent context by performing a migration operation, if possible.

The master–slave part of the planner behaves similarly to the hierarchical planner with one exception. The plan refers to the whole region, not a specific lower-level context. The execution either modifies the trust contexts^⑨ or modifies the regions^⑩.

8 EVALUATION

We conducted experiments on two systems to evaluate our approach. The first system is a randomly generated decentralized architecture, while the second system is an extension of our running example, representing a more realistic microservice architecture. We performed experiments using all four combinations of decentralized architectures and utility evaluation functions as described in more detail below.

8.1 Experiment Set up

For the randomly generated system, we evaluated each setting and experiment type with 100, 1,000, and 10,000 modules. Initial trust is set randomly. Trust contexts are then formed by grouping entities with high mutual trust. The initial hierarchy is constructed as a balanced tree. For the case studies of 1,000 and 10,000 modules, we extended the hierarchy with one and two more layers, respectively.

The microservice system is an extension of our running example, depicted in Figure 1. The ItemShop is extended with a Cart microservice, the Bank is extended with Account and Billing microservices, and we extend the ShippingCo with a Delivery microservice. We further scale the initial case study to produce the 100 microservices and 1,000 microservices case studies, which contain 10 and 100 item shops, banks, and shipping companies, respectively. The initial trust assignment is based on our running example and microservices are grouped according to their stakeholders.

8.2 Attacker Model

We assume that the attacker cannot intercept, read and modify DIFC messages aimed at modules that s/he does not control, s/he cannot access, decipher or guide the controller's operations and s/he cannot control the adaptation process. An attacker is assumed to be any party that controls at least a single module and aims to subvert or exploit the modules s/he does not control to compromise the confidentiality or integrity of the decentralized system. The attacker aims to infer information that s/he should have no access to or tamper with the validity of the information in the system. While the DIFC messages themselves cannot be tampered with, the attacker can legally modify the information and, therefore, impact the integrity of the information in the system by gaining the required labels. The attacker can affect confidentiality by gaining ownership of tags that are not lawfully owned by modules that s/he controls.

One strategy to obtain ownership is to send a combination of valid DIFC messages that aim to gain the trust of target modules and then send carefully selected malicious DIFC messages, aiming to learn the desired information by abusing the newly gained trust. In our newly discovered attacks, the attacker aims to either flood the system with invalid DIFC messages to negatively influence the trust of the modules that reside in shared or communicating contexts or disrupt the system operation. The attacker may attempt to (i) coerce the system into revealing tags with a low trust value, (ii) claim ownership of tags of modules that have left the system to disrupt the system operation by using those tags in his/her modules' labels, and (iii) attempt to gain capabilities over tags by claiming ownership over them. Once the attacker controls a tag's capabilities, he/she can partially control its usage in the system's security policies.

8.3 Experiment Results and Discussion

8.3.1 Average Trust—Running Time. Figure 9 depicts the resulting average trust values for both case studies while Table 2 and Table 3 depict total time in milliseconds for 100 adaptation runs. The X axis corresponds to the current adaptation step. The goal of this type of experiments is to identify the stable behavior of our DIFC architectures, i.e., how the average trust is affected by each combination of architecture and utility function. In our analysis, we observe that both types of systems perform similarly in terms of trust maximization and adaptation time. In particular, the hierarchical architecture provides a higher degree of average trust, while the *hybrid* architecture provides a shorter adaptation time. In other words, we can identify a tradeoff between the time required to adapt and the amount of achievable average trust. In either setting, the global utility evaluation provides a higher degree of trust, albeit at a slower adaptation time in comparison to the local utility function. We should also note that even though the hashing and the added communication required to enforce the interface versioning scheme have an impact on the system throughput, that impact has a negligible effect on adaptation time. Moreover, it should be noted that our updated region selection process in the hybrid decentralized architecture has improved its performance in terms of average trust by about 20% when compared to the previous version of our experiments, for all experiments performed in this evaluation.

Fig. 9. Average Trust.

Depending on the behavior of the malicious actors in the model, the effectiveness of local utility evaluation varies. When comparing the microservices and random case studies, we can see that local utility evaluation performs worse in the random case study as the malicious modules are allowed to invoke any interface of any module in the system, which is a consequence of crossing trust context boundaries and even region boundaries in the hybrid-planner architecture. In the microservices scenario, the malicious actors are only allowed to invoke interfaces in a restricted set of the modules, as a consequence, the local utility evaluation has a more accurate view of the adaptation, since fewer cross-trust context, or cross-region invocations are possible. Furthermore, as shown in Figure 9, the number of modules affects the number of adaptations required for the maximal trust to be established. We expect that trust stabilizes when all malicious actors in the system have been isolated, as from that point on, no DIFC message will be blocked, since in this type of experiments no modules enter or exit the system. Note that as long as at least one malicious actor exists, trust can never reach 100%.

8.3.2 Average Trust under Increasing Attacker Presence. Figure 10 shows the average trust in the system after 100 adaptations when the attacker controls an increasing percentage of the system's

Table 2. Adaptation Time for 100 Adaptations (ms)—Random

Modules	HierGlob	HierLoc	HybrGlob	HybrLoc
100	2,978	630	313	72
1,000	853,730	16,924	8,321	1,489
10,000	3,577,923	712,935	331,720	50,192

Table 3. Adaptation Time for 100 Adaptations (ms)—Microservices

Microservices	HierGlob	HierLoc	HybrGlob	HybrLoc
10	713	243	72	16
100	6,417	813	436	143
1,000	11,958	23,180	12,364	2,153

Fig. 10. Average Trust with 10%, 20%, and 30% of attacker controlled modules.

modules, noted in the legend. As the attacker controls a larger part of the system, the overall maximum possible established trust drops. For 10% of attacker controlled modules, the maximum possible trust is 90%; for 20% and 30% attacker presence the percentages adjust accordingly to 80% and 70%. In both case studies, the average trust stabilizes very close the percentage of the modules not controlled by the attacker in the hierarchical settings. The hybrid architectures also stabilize eventually, albeit at a slower rate.

8.3.3 Functionality Versus Security. Figure 11 depicts the average trust after 100 adaptations, when in addition to the attacker controlling a larger part of the system, 20% of the modules are deemed critical to the functionality and are thus never isolated. As shown in the figure, in both case studies, the average trust of the system can be noticeably lower depending on the part of

Fig. 11. Average Trust after 100 adaptations when 20% of modules are critical.

the system that the attacker controls. This happens because all directly communicating modules to a trusted critical component will be eventually compromised. In the random case study, since the initial trust is allocated randomly and on average each module communicates with a smaller percentage of the total modules, average trust is slightly higher but still noticeably lower when compared to the non-critical modules experiment in Section 8.3.1.

8.3.4 Average Trust in a Dynamic System. Figure 12 shows the average trust of the system after 100 adaptations, when a number of modules randomly enter or exit the system, between each adaptation. The figure's legend shows the percentage of change that happens in terms of module addition or removal. A high rate of change leads to higher uncertainty, and a result, for both case studies, lower average trust when comparing to the experiments in Section 8.3.1, where the number of modules is constant and no modules enter or exit the system. We can also notice that in this type of experiments, the hybrid planner compares more equally to the hierarchical architecture, as it performs adaptations in a higher rate and is better able to cope with the increasing change in the system. It should also be noted that uncertainty in terms of modules entering and exiting the system does not affect each case study equally, with the microservices case study being more affected than its random counterpart.

8.3.5 Average Trust with Varying Operation Thresholds. We performed a set of experiments to identify the effects of varying merge, split and isolation thresholds to the average trust of the system. The results show that the system can reach three distinct states. In the first state, when the initial threshold values are too strict, no actual progress can be made, and while the security policies are always enforced, the rate of information flowing through the system is limited. On the opposite side, when the thresholds are too permissive, the attacker has sufficient time to compromise the system and a large part of the system will be isolated, as a result. In the case where the

Fig. 12. Average Trust when 5%, 10%, and 20% of the system changes per adaptation.

thresholds are set somewhere in between, the system will operate normally and eventually the thresholds will converge to their optimal values.

8.3.6 Average Trust under Frequent Policy Changes. We have performed a number of experiments where the malicious modules in the system explicitly attempt to defame the entities calling their interfaces as per the defamation attack discussed in Section 4.5. The results show that while the message throughput in the system is slightly affected, due to a number of messages being retried, the system trust remains mostly unaffected by the attack. The rate to which the maximum trust can be reached in this case is slower since fewer messages are successful or considered malicious. If the malicious modules, nevertheless, continue to generate version signatures, as often as they are able, in an attempt to flood the system as discussed in the resource starvation attack in Section 4.5, then once the threshold Tr_p is surpassed, each of those modules is isolated. All such modules in the system have successfully been identified and isolated in this experiment.

8.3.7 Impersonation Attacks. Our final set of experiments evaluates the effectiveness of our mitigations against tag ownership attacks discussed in Section 4.5. In our experiments, all such attacks were identified and the modules attempting to illegitimately claim tag ownership have been isolated once they have attempted to perform more illegal operations than allowed by the Tr_i threshold.

8.4 Threats To Validity

The small total number of experiments is a threat to the validity of this study. More experiments and, specifically, experiments on different decentralized system architectures are needed. We additionally acknowledge that messages in a real setting may be dropped for other reasons than maliciousness, for instance, due to network errors. However, we define the evaluation in terms of

trustworthiness in addition to secrecy, thus, we believe that the network unreliability should be considered an availability concern.

We believe that the randomized experiment does not affect the validity of the results. The experiment results are commensurate to the microservice case study and conform with our expectations that trust in the system stabilizes once malicious actors are identified and are isolated. Moreover, we have not performed experiments where the attacker controls the majority of the modules that participate in the architecture. However, we believe that in that case, the system will rapidly collapse into a segregated state of disconnected modules controlled by the attacker. As a result, several context hierarchies will be unable to establish the minimum trust required to operate, and therefore, drop all DIFC messages. From a security point of view, no information will be leaked or tampered with, however, a large part of the system though will become non-functional. We acknowledge that we have only taken the security of information flowing through the system into account, and do not tackle problems related to secure data communication or medium tampering, which we consider out of scope. Secure communication can be achieved via encryption. Encryption schemes used in messaging platforms or applications can be used to facilitate secure multi-party transmission of messages. Medium tampering is completely out of scope, since we separate the communication channel employed for controlling information flow from the channel through which the modules communicate with each other.

9 RELATED WORK

To the best of our knowledge, no other approach that combines trust, decentralized information flow control and decentralized control to enforce security in open systems has been proposed in the literature. Therefore, we cannot provide a quantitative comparison with other state of the art research that addresses the exact same problem. In the related work that follows, we compare and contrast each work with the corresponding aspect of our approach qualitatively.

Decentralized/Dynamic Information Flow. Several application domains use DIFC to enforce security, e.g., operating systems [12, 17, 20, 30], databases [20], Android applications [16] and SaaS [24]. In operating systems, DIFC controls information flow among communicating OS-level processes. A user-space library enables the processes to specify security policies and a kernel module to verify such policies. Roy et al. [17] combine this approach with lexically scoped security regions and a Java Virtual Machine implementation. In Reference [20], the system tracks sensitive information as it flows through the database system and the applications. The authors in Reference [16] proposes a context-sensitive DIFC enforcement to enforce security in Android applications via lazy polyinstantiation and secure network exporting through domain declassification. Tingting and Yong [24] present a DIFC for SaaS, which allows untrusted services to use private data, while trusted services control the dissemination of that data. All approaches above assume a central, trusted component to validate and verify system boundary control and DIFC policies. This component requires mutual agreement with each entity in the system to act as a trusted arbiter. In essence, the trusted component is both a bottleneck and a single point of failure. In terms of throughput, all valid communication that respects DIFC needs to be validated by the trusted component, which can increase the system latency, and in case the component is compromised, the whole system is compromised, as DIFC cannot be enforced in any part of the system. In contrast, our approach has no centralized controller, making it more suitable for open systems. If some system controllers are compromised, then only the parts of the system managed by the compromised controllers are affected. The throughput in our system is higher as messages only need to reach the first common ancestor in the architecture for DIFC verification. Thus it provides provisioning for a higher degree of concurrency in the message exchange. In addition, extending the DIFC

model with trust allows our modules to define their trustworthiness and select how much information they share with other parts of the system, even with controllers independently. We also incorporate the approach in an adaptive architecture, which facilitates self-adaptive security.

Decentralized and Adaptive Trust. Several trust-based approaches have been proposed to ensure security in distributed systems. Khan et al. [11] attempt to isolate malicious nodes in MANET systems. When the trust of a node in the network drops below a specific threshold, the node is considered malicious, and is isolated. Zhang et al. [31] propose several distributed, adaptive trust metrics for mobile ad hoc networks. These metrics include direct trust, communication, energy trust. They then derive recommendation trust, which uses reliability, familiarity, and integrated direct trust, which considers direct trust and recommendation trust. Ahamed et al. [2] propose an evolutionary model based on historical behavior, behavior feedback, and the total number of actions for each kind. The model is essential for the proposed trust management system that provides protection mechanisms against attacks. Shehada et al. [21] propose an adaptive trust and reputation model, which is tolerant to witnesses' behavioral changes in mobile agent systems. The model uses direct and reputation values in the evaluation and assesses the witnesses' honesty. Ahamed [1] presents a context-aware reputation-based trust model, which has dynamic initial trust modeling capabilities for unknown devices. A model for separating malicious nodes is incorporated. The above works feature a trust model and aim to identify or isolate malicious nodes. They target specific types of networks and defend against specific classes of attacks. In contrast to the above methods, our approach is system-agnostic and applies to any message-passing system. Further, we focus on enforcing message-level security policies that provide fine-grained control of the system's information flow. Moreover, our approach is dynamic, as the module security policies, behavior, and trust may change during the system runtime, which allows for flexible security. In our system, an entity that stops behaving maliciously can gradually regain its trusted status by transmitting enough AT-DIFC⁺ respecting messages to regain its peers trust. We believe the trust-based approaches presented above are not as flexible in terms of the granularity of policy specification and enforcement, their ability to handle dynamic trust updates and their ability to handle system openness.

Decentralized Feedback Loops. Weyns et al. [27], describe patterns for decentralized control of multiple MAPE loops. They also highlight decisions to decentralize MAPE-K architectures. Furthermore, Weyns, Malek, and Andersson [26] describe the critical attributes of decentralized self-adaptive systems using several case studies on decentralized self-adaptation. Sylla et al. propose a design framework for reliable self-adaptive systems used to compose multiple feedback loop managers [23]. Building on generic autonomic loops that combine automata-based control and transactional execution, they propose patterns for autonomic loop composition. In this work, we use the hierarchical pattern [27] and compose the regional planner and master-slave patterns [27] in the hybrid planner.

Adaptive Security and Self-Protection. Yuan et al. [28], propose a taxonomy for self-protecting systems and provide an architecture-based approach to self-protection [29]. They provide architecture adaptation patterns to detect and mitigate security weaknesses in web applications. The authors of Reference [19] compose tactics to mitigate DoS attacks by applying utility-based reasoning to select strategies to apply in the system. Amad and Roudies [3] utilize a MAPE-K architecture to negotiate and deploy security policies in mobile devices, dynamically. Dynamic software product lines techniques are used to perform adaptations at runtime. Lamprecht [14] provides a methodology for runtime security adaptation in non-adaptive systems, uses offline analysis, and dynamically creates policies to tradeoff performance and security. Tun et al. [25]

propose an approach to specify adaptive security under partial environment knowledge that is dynamically updated, as more information becomes available at runtime. Khakpour et al. [10] provide a method to perform security risk assessment on adaptations that considers dependencies of components vulnerabilities. Compared to the current work, (i) we combine information flow control and trust-based methods to enforce security that is a novel approach not present in any of the discussed work, (ii) we target distributed open systems with multiple feedback loops, where modules can specify their policies individually, and (iii) instead of adapting the underlying system, we adapt the structure and behavior of the superimposed trust architecture, which enables us to support a broader range of systems compared to the approaches above.

10 CONCLUSION

In this article, we presented AT-DIFC⁺, a successor of AT-DIFC, a DIFC model extended with trust and adaptation capabilities that enables (i) security policy enforcement in a distributed manner and (ii) dynamic trust architecture adaptation to ensure security. The model utilizes a combination of decentralized information flow control mechanisms, trust-based methods and decentralized control architectures for self-adaptive systems. We have updated the AT-DIFC model and its implementation to mitigate recently identified attacks and thus provided for extended security capabilities. We applied our approach to two case studies and conducted several experiments as part of its evaluation. Our experiments show that the mechanism effectively protects the system against the identified attacks at a small cost to the system's performance.

As future work, we plan to prove the formal guarantees of our approach and scale our approach to other types of decentralized systems. In that work, we plan to incorporate further adaptation operations and explore more decentralized control architectures. We also plan to investigate advanced strategies to generate plans, for example, via SMT or constraint solving. We will also extend the evaluation with more experiments that consider additional decentralized systems types and eventually evaluate the mechanism on commercial systems.

REFERENCES

- [1] S. I. Ahamed, M. M. Haque, Md. E. Hoque, F. Rahman, and N. Talukder. 2010. Design, analysis, and deployment of omnipresent formal trust model (FTM) with trust bootstrapping for pervasive environments. *J. Syst. Softw.* 83 (2010), 253–270.
- [2] F. Almenázar, A. Marín, D. Díaz, A. Cortés, C. Campo, and C. García-Rubio. 2011. Trust management for multimedia P2P applications in autonomic networking. *Ad Hoc Networks* 9, 4 (June, 2011), 687–697.
- [3] M. Amoud and O. Roudies. 2017. Dynamic adaptation and reconfiguration of security in mobile devices. In *Proceedings of the International Conference on Cyber Incident Response, Coordination, Containment Control (Cyber Incident'17)*. IEEE, 1–6.
- [4] K. Bellman, S. Tomforde, and R. P. Würtz. 2014. Interwoven systems: Self-Improving systems integration. In *Proceedings of the IEEE 8th International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. IEEE, 123–127.
- [5] M. Dworkin. 2015. FIPS PUB 202, Federal Information Processing Standards. SHA-3 standard: Permutation-based hash and extendable-output functions. NIST, 1–37.
- [6] C. M. Fiduccia and R. M. Mattheyses. 1982. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*. IEEE Press, 175–181.
- [7] D. Gambetta. 1988. *Can We Trust Trust?* Blackwell, Oxford, UK, 213–237.
- [8] D. Hedin and A. Sabelfeld. 2012. A perspective on information-flow control. In *Software Safety and Security*. IOS Press, 1–29.
- [9] J. O. Kephart and D. M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.
- [10] N. Khakpour, C. Skandylas, G. S. Nariman, and D. Weyns. 2019. Towards secure architecture-based adaptations. In *Proceedings of the IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'19)*. IEEE, 114–125.
- [11] M. S. Khan, D. Midi, S. Malik, M. I. Khan, N. Javaid, and E. Bertino. 2017. Isolating misbehaving nodes in MANETs with an adaptive trust threshold strategy. *Mobile Netw. Appl.* 22, 3 (2017), 493–509.

- [12] M. Krohn and E. Tromer. 2009. Noninterference for a practical DIFC-based operating system. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*. IEEE, 61–76.
- [13] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. 2007. Information flow control for standard os abstractions. *SIGOPS Oper. Syst. Rev.* 41, 6 (2007), 321–334.
- [14] C. J. Lamprecht. 2012. *Adaptive Security*. Ph.D. Dissertation. Newcastle University.
- [15] A. C. Myers and B. Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.* 9, 4 (2000), 410–442.
- [16] A. Nadkarni, B. Andow, W. Enck, and S. Jha. 2016. Practical DIFC enforcement on android. In *Proceedings of the 25th USENIX Conference on Security Symposium*. USENIX Association, 1119–1136.
- [17] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. 2009. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, 63–74.
- [18] A. Sabelfeld and A. C. Myers. 2006. Language-based information-flow security. *IEEE J. Sel. A. Commun.* 21, 1 (2006), 5–19.
- [19] B. Schmerl, J. Cámara, J. Gennari, D. Garlan, P. Casanova, G. A. Moreno, T. J. Glazier, and J. M. Barnes. 2014. Architecture-based self-protection: Composing and reasoning about denial-of-service mitigations. In *Proceedings of the Symposium and Bootcamp on the Science of Security*. Association for Computing Machinery, New York, NY, 1–12.
- [20] D. Schultz and B. Liskov. 2013. IFDB: Decentralized information flow control for databases. In *Proceedings of the 8th ACM European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, 43–56.
- [21] D. Shehada, C. Y. Yeun, M. J. Zemerly, M. Al-Qutayri, Y. Al-Hammadi, and J. Hu. 2018. A new adaptive trust and reputation model for mobile agent systems. *J. Netw. Comput. Appl.* 124 (2018), 33 – 43.
- [22] C. Skandylas, N. Khakpour, and J. Andersson. 2020. Adaptive trust-aware decentralized information flow control. In *Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS'20)*. IEEE, Washington, DC, USA, 92–101.
- [23] A. N. Sylla, M. Louvel, E. Rutten, and G. Delaval. 2017. Design framework for reliable multiple autonomic loops in smart environments. In *Proceedings of the International Conference on Cloud and Autonomic Computing (ICCAC'17)*. IEEE, 131–142.
- [24] L. Tingting and Z. Yong. 2013. A decentralized information flow model for saas applications security. In *Proceedings of the 3rd International Conference on Intelligent System Design and Engineering Applications*. IEEE, 40–43.
- [25] T. T. Tun, M. Yang, A. Bandara, Y. Yu, A. Nhlabatsi, N. Khan, K. Khan, and B. Nuseibeh. 2018. Requirements and specifications for adaptive security: Concepts and analysis. In *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS@ICSE'18)*. IEEE, Gothenburg, Sweden, 161–171.
- [26] D. Weyns, S. Malek, and J. Andersson. 2010. On decentralized self-adaptation: Lessons from the trenches and challenges for the future. In *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. Association for Computing Machinery, New York, NY, 84–93.
- [27] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka. 2013. *On Patterns for Decentralized Control in Self-Adaptive Systems*. Springer, Berlin, 76–107.
- [28] E. Yuan, N. Esfahani, and S. Malek. 2014. A systematic survey of self-protecting software systems. *ACM Trans. Auton. Adapt. Syst.* 8, 4 (2014), 17–41.
- [29] E. Yuan, S. Malek, B. Schmerl, D. Garlan, and J. Gennari. 2013. Architecture-based self-protecting software systems. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*. ACM, New York, NY, 33–42.
- [30] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. 2006. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, Volume 7*. USENIX Association, Berkeley, CA, 263–278.
- [31] D. Zhang, J. Gao, X. Liu, T. Zhang, and D. Zhao. 2019. Novel approach of distributed adaptive trust metrics for MANET. *Wireless Netw.* 25 (2019), 3587–3603.

Received August 2021; accepted September 2021