Journal of Cybersecurity and Privacy

MDPI

Article

Recursive Augmented Fernet (RAF) Token: Alleviating the Pain of Stolen Tokens

Reza Rahaeimehr [1,*] and Marten van Dijk [2,3,4]

[1] School of Computer and Cyber Sciences, Augusta University, Augusta, GA 30912, USA
[2] Centrum Wiskunde & Informatica (CWI), 1098 XG Amsterdam, The Netherlands; mevd@cwi.nl
[3] Department of Computer Science, Faculty of Science, Vrije Universiteit Amsterdam, 1081 HV Amsterdam, The Netherlands
[4] Department of Electrical and Computer Engineering, University of Connecticut, Storrs, CT 06269, USA
* Correspondence: rrahaeimehr@augusta.edu

Abstract

A robust authentication and authorization mechanism is imperative in modular system development, where modularity and modular thinking are pivotal. Traditional systems often employ identity modules responsible for authentication and token issuance. Tokens, representing user credentials, offer advantages such as reduced reliance on passwords, limited lifespan, and scoped access. Despite these benefits, the "bearer token" problem persists, leaving systems vulnerable to abuse if tokens are compromised. We propose a token-based authentication mechanism addressing the critical bearer token problem in modular systems. The proposed mechanism includes a novel RAF (Recursive Augmented Fernet) token, a blacklist component, and a policy enforcer component. RAF tokens are one-time-use tokens, like tickets. They carry commands, and the receiver of an RAF token can issue new tokens using the received RAF token. The blacklist component guarantees an RAF token cannot be validated more than once, and the policy enforcer checks the compatibility of commands carried by an RAF token. We introduce two variations of RAF tokens: user-tied RAF, offering simplicity and compatibility, and fully-tied RAF, providing enhanced security through service-specific secret keys. We thoroughly discuss the security guarantees, technical definitions, and construction of RAF tokens backed by game-based proofs. We demonstrate a proof of concept in the context of OpenStack, involving modifications to Keystone and the creation of an RAFT library. The experimental results reveal minimal overhead in typical scenarios, establishing the practicality and effectiveness of RAF. Our experiments show that the RAF mechanism outperforms the use of short-life Fernet tokens while providing much better security.

Keywords: authentication; OpenStack; token-based authentication; distributed system; modular design security

Academic Editors: Yangguang Tian and Danda B. Rawat

Received: 17 March 2026
Revised: 28 April 2026
Accepted: 7 May 2026
Published: 13 May 2026

1. Introduction

Modularity, modular thinking, and modular development models are essential for building large-scale systems. For a modular system, a proper authentication and authorization mechanism is essential. Usually, a modular system includes an identity module that is in charge of authentication and authorization. Users present their credentials (usually a username and password) to the identity module and are granted tokens in exchange. Other modules only serve users who have valid tokens. Tokens are either a random bit string like a UUID that points to an entry in an authentication and authorization database or a crafted

piece of data that carries some authentication and authorization information like Fernet and JWT tokens (see Section 2.2).

Using a token instead of a username and a password has three main advantages:

- Users do not need to use their passwords for every system access. Consequently, the probability of leaking passwords gets lower.
- Every token has an expiration time. Hence, a stolen token can only be abused for a limited time.
- It is possible to limit the scope of a token and let the user ask for a less privileged token. In this case, if an adversary steals the token, they will have limited access to the resources of the token owner.

The above features allow a user to get a short-life-scoped token from the identity module and delegate the authority to a third trusted party to accomplish a complicated task within the scope.

Problem: Although the ability to use a token and make requests on behalf of a user (authorization delegation) is a vital feature for a highly modular system, it is the source of many vulnerabilities [1]. Adversaries can find bugs in large-scale systems. With the current token mechanisms [2–4], if an adversary exploits some of these bugs and finds a way to obtain user tokens, they can do whatever the owners of the tokens can do. This problem is known as the "bearer token" problem [1].

Scoping tokens down and reducing their lifetime are two ways used to reduce the impact of the bearer token problem [5]. However, all the typical token types like UUID, PKI, Fernet, JWT [4], and ticketing systems like Kerberos [6] have two main shortcomings:

- First, in current mechanisms, the immediate impact of shortening the lifetime of tokens or reducing the scope of tokens is that users need to renew their tokens more often. In practice, if we remarkably shorten the lifetime of tokens and reduce the scope of tokens, the identity module becomes a bottleneck.
- Second, no matter how much a system shortens the lifetime of tokens or limits the scope of tokens, a stolen token can be very dangerous. This is because there is no relation between tokens and commands. For example, in OpenStack, if you get a one-second life token for deleting a volume in a project and the token is leaked to an adversary, the adversary can delete all the volumes in the project during that one second.

Considering this problem, system designers make a trade-off between system performance and security. For example, in OpenStack [7], the default token lifetime is one hour, and the smallest scope is a project [8]. In summary, despite shortening the lifetime or reducing the scope of a token, an adversary who can corrupt a module and observe a token can do tremendous damage. Having a bug-free system is not realistic, but reducing the effects of buggy code is possible. Therefore, it is desirable to generalize the modularity security requirement proposed by Maleki et al. [1] as follows:

> For a modular system, if an adversary corrupts a module, other modules just do whatever requested to do by the users, nothing more.

For instance, in OpenStack, it is desirable that, in the event an adversary compromises the networking module, they should be unable to create or delete a VM.

Our work: This work addresses the modularity security requirement for modular and distributed systems by developing the following components:

1. A novel token that we call the Recursive Augmented Fernet (RAF) token, which allows commands to be added to tokens without requiring additional identity module interaction.
2. A blacklist structure that prevents replay attacks.

3. A policy enforcer component that prevents malicious command execution.

We introduce two subtypes: *user-tied RAF (Ut-RAF)* which is the best option for systems currently using Fernet tokens and seeking more secure solutions without the need for significant changes, and *fully-tied RAF (Ft-RAF)*, which provides better security guarantees. Ft-RAF requires each module to have a secret key shared with the identity module. Hence, it needs a key distribution/renewal mechanism, which makes it harder to deploy compared to Ut-RAF. A summary of the security analysis of Ut-RAF and Ft-RAF is given in Section 3, and the details can be found in Appendix A.

We explain our proposal in the context of OpenStack to show its usability and performance in practice. OpenStack is one of the world's most successful open-source software projects, developed by tens of thousands of contributors around the world. It offers a robust and smooth infrastructure-as-a-service (IaaS) cloud platform and has been utilized by companies like Red Hat, American Express, IBM, AT&T, Adobe, and Best Buy [9]. Therefore, we firmly believe that if RAF tokens function effectively in OpenStack, they will easily integrate with other systems.

Our proposal has two important strengths:

First, it allows for the reduction of the lifetime or scope of tokens without interacting with the identity module. It ties tokens to commands and, in fact, limits the usage of tokens to the commands issued by users and thereby minimizes the impact of stolen tokens.

Second, it offers third-party Fernet-token-based system add-ons the choice to continue using Fernet and gradually upgrade to RAF (if the system policy continues accepting Fernet tokens). Note that, in our solution, the identity module still issues Fernet tokens.

We have implemented an RAF library that allows for the issuance of RAF tokens based on either Fernet or on other RAF tokens. Also, we have modified the identity module of OpenStack to accept and validate RAF tokens. The experimental results show that our mechanism adds less than 1 percent overhead to the validation time of a token, i.e., less than a microsecond. Also, our experiment showed that generating an RAF token from a long-life Fernet token is significantly (more than 88 times in our experiments) faster compared to getting a new short-life Fernet token from the identity module.

Threat Model: A service (In OpenStack, each module is referred to as a service. Therefore, in this article, we use the terms 'module' and 'service' interchangeably.) can be corrupted in two ways:

- Partial corruption: Where a service leaks tokens to the adversary but functions as intended.
- Full corruption: Where a service is under the control of the adversary, enabling the adversary to do whatever they want within the service.

We consider a strong adversary who can fully corrupt some services and observe all network communications except User-Identity module communications. Hence, user-identity module communications must be confidential and authenticated. In OpenStack, it is possible to secure user-service communication by enabling TLS and having a certificate for each service. However, since service-to-service communication is not secure, we also assume user-service communication is not secure. This allows us to cover a wider range of applications and provide stronger security.

Comparison with Existing Mitigation Strategies: Existing mitigation strategies for bearer tokens primarily focus on limiting the duration or scope of token usage. While these approaches reduce the attack surface, they do not fundamentally prevent the misuse of a compromised token within its valid lifetime or scope.

RAF differs from these approaches by introducing command-bound, single-use tokens. Instead of only restricting *when* or *where* a token can be used, RAF constrains *how* a token can be used by binding it to a specific command and enforcing execution consistency across

services. This design ensures that even if a token is leaked, its effect is strictly limited to a predefined operation and cannot be reused or extended arbitrarily.

In practical modular environments such as OpenStack, where services interact on behalf of users, these limitations become more pronounced. A compromised or buggy service may still misuse a valid token within its allowed scope, since existing approaches do not constrain how tokens are used once accepted. This gap motivates the need for mechanisms such as RAF that enforce strict usage semantics in addition to validity constraints.

Table 1 summarizes the key differences between RAF and commonly used token-based mitigation strategies.

Table 1. Comparison of the RAF with existing token-based mitigation strategies.

Property	Short-Lived	Scoped	Delegation	RAF
Limits validity duration	Yes	Yes	Yes	Yes
Restricts access scope	No	Yes	Yes	Yes
Prevents token reuse	No	No	No	Yes
Binds token to command	No	No	No	Yes
Enforces execution consistency	No	No	No	Yes
Mitigates misuse after leakage	Partial	Partial	Partial	Strong

Organization: We commence with an overview of the history and structure of OpenStack, elucidating the four popular token formats in Section 2.1. Section 3 provides details of our tokening mechanism, while Section 4 delves into the security analysis of RAF. In Section 5, we present a proof of concept for RAF and describe some of the results obtained from using RAF in our experiments. We conclude our work with a summary in Section 6.

2. Background

2.1. OpenStack

In 2010, Rackspace wanted to redesign its cloud server offering infrastructure, while NASA was interested in a similar project. Negotiation between the two teams led to a shared program called OpenStack. The OpenStack Foundation was established in September 2012 and has been in charge of OpenStack since then. As of 2025, the OpenStack community continued to grow globally, with contributors and users from more than 180 countries collaborating on one of the world's largest open-source cloud computing platforms [10].

OpenStack is a modular system divided into projects (services) at the highest level. Every project utilizes third-party plugins that must be open-source as well. Usually, for a use case, OpenStack supports several plugins that can be used interchangeably. The following projects constitute the core of OpenStack:
Keystone implements the Identity API and is in charge of authentication and authorization.
Nova is the heart of OpenStack and manages computation resources.
Cinder provides block storage (volumes) mainly used by virtual machines.
Glance is the imaging service, which maintains data assets like operating systems.
Neutron enables networking services.
Swift provides efficient block storage and can be used independently. Hence, some companies take advantage of Swift for storing data.
Horizon resembles the dashboard of OpenStack and provides a web interface for clients. Although Horizon is not the only way for interacting with OpenStack, it is the simplest way for clients if they want to manage the system manually.

In addition to Horizon, users can send their request to OpenStack via cURL [11] or the OpenStack Client library. cURL allows a person to send an HTTP/HTTPS request.

OpenStack Client library is a tool for developers that enables them to convert users' high-level commands to corresponding cURL commands.

Each service implements some Representational State Transfer (REST) APIs, which are the standard way to communicate with a service in OpenStack. A REST API is a web service that is callable by HTTP/HTTPS requests. The OpenStack REST APIs accept parameters in JSON format. To access OpenStack, first, a user must get a token from Keystone by presenting their username and password. Each token has an expiration time, and the user needs to refresh their token periodically. The user must put the token in the header of all their requests to other modules. Whenever a service receives a request, first of all, it checks the presence of a token in the header. Then, the service validates the token with Keystone. Sometimes a service needs to make a request to other services on behalf of the user using the received token. For example, whenever a user wants to create a virtual machine, in addition to the virtual machine specification (like the amount of RAM and number of virtual CPUs), they must specify a network. In this case, the user sends his request to Nova. Nova builds the VM, but it needs to get in touch with Neutron to establish the connection between the VM and the network. Figure 1 shows the interaction between a user and some OpenStack modules as described above.

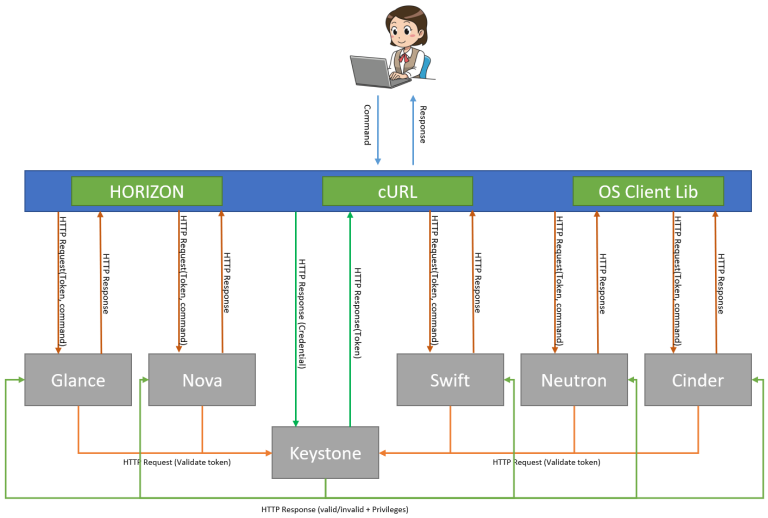


Figure 1. The interaction between a user and the main OpenStack modules.

2.2. Token Formats

The four most common types of tokens that have also been used in OpenStack are as follows:

2.2.1. UUID

A UUID token is nothing more than a random string (32 characters in OpenStack) and carries no information. Consequently, to validate the issued tokens, the system needs to store them and relative metadata in a database. Implementing the UUID token mechanism is straightforward. Its main disadvantage is the token database. In a practical deployment, a UUID token database grows fast, and after a while, inserting and retrieving data from the database turns into a bottleneck for high-throughput systems. In addition, the complexity of establishing multiple identity nodes and scalability are other concerns that convinced the OpenStack community to look for alternative token mechanisms.

2.2.2. PKI and PKIz

The PKI token was the first attempt to utilize public key infrastructure in OpenStack. A PKI token carries some meaningful data, like the owner's specification, privileges, issue

time, and expiration time of the token, and a catalog list signed by the private key of the identity service. Therefore, PKI tokens are very long and in practice can exceed 8k bytes, which does not fit in an HTTP header by default. The motivation was to minimize the need for database access in Keystone. At first glance, Keystone does not need to store PKI tokens. A PKI token can be verified by anyone who has the public key of Keystone. OpenStack services do not need to get in touch with Keystone to discover the privileges of a PKI token. However, in practice, this is not accurate: A service still needs to check a token with Keystone to see if it has not been revoked; Keystone has to keep the revoked tokens in a database (or similar structure).

PKIz is the compressed version of PKI, which was an effort to make PKI tokens small. Nevertheless, this technique did not help much, and PKIz tokens were only 10 percent smaller than PKI tokens in real usage. Because of all the disadvantages described above, the OpenStack community abandoned the PKI and PKIz tokens.

2.2.3. Fernet

In 2013, Heroku introduced its secure message transmission method, which was called Fernet. They encrypted a message and concatenated the ciphertext with the HMAC of the ciphertext and called the resultant a Fernet token. In 2015, the Keystone core team adapted the Fernet token for OpenStack. OpenStack Kilo is the first release of OpenStack that supports Fernet tokens, and nowadays, it is the most used token format.

In OpenStack, Fernet tokens contain a payload that is encrypted and symmetrically signed by Keystone's secret key. The payload usually includes the expiration time of the token, the ID of the owner, an audit ID, and the scope of the token. Any Keystone node that has the secret key can validate a Fernet token and extract the payload. The main advantage of Fernet tokens is that there is no need for a database (backend) for storing and validating them. These features made the Fernet format the most successful token format in OpenStack.

2.2.4. JWS

The OpenStack Stein adapts the JSON Web Signature (JWS) token format that is a subtype of JSON Web Token (JWT) [4]. Each JWS token contains a header, a payload, and the signature of the token. OpenStack uses the ES256 JSON Web Algorithm with a private key for signing JWS tokens. Hence, whoever has the corresponding public key can verify a JWS token. We can say that the JWS format is the updated version of the PKI format. Unlike PKI tokens, JWS tokens contain little data, and the payload of a JWS token is not encrypted. Consequently, a JWS token is much shorter than a PKI token. However, it is still about twice as large as a Fernet token. Like Fernet tokens, JWS tokens are ephemeral; this means we do not need to store them in a token database and replicate the database across all Keystone nodes. There is no apparent preference between Fernet and JWS.

2.2.5. Bearer Token Mitigation Approaches

In addition to traditional bearer tokens, several approaches have been proposed to mitigate token theft and misuse. These approaches can be broadly categorized into two groups. The first group introduces proof-of-possession mechanisms, such as DPoP [12] and mutual TLS (mTLS) sender-constrained tokens [13], which bind tokens to a client-held private key, making them unusable by unauthorized parties that do not possess the corresponding key.

The second group focuses on reducing token exposure through architectural or operational techniques, including short-lived tokens, refresh-token mechanisms, and gateway-based patterns such as phantom tokens [14] or split tokens.

While these approaches significantly reduce the risk of token misuse, they either require additional infrastructure and key management (in the case of proof-of-possession mechanisms) or do not fundamentally restrict the actions that can be performed once a token is accepted as valid. In contrast, RAF binds each token to a specific command and enforces consistency across derived tokens, thereby limiting misuse even in the presence of a compromised service.

2.3. Fernet Tokens

The Fernet format is the base of our solution, and understanding how it works is essential. Hence, we describe Fernet in more detail here. Keystone needs a Fernet key to generate Fernet tokens. A Fernet key is a base64 URL-safe string composed of a signing key and an encryption key. If we decode a Fernet key, we will have a 32-byte array, of which the first half (128 bits) is the signing key, and the second half is the encryption key. Note that it is possible to use larger key. Since OpenStack uses a fresh key every 1 hour, the OpenStack community believes that a 128-bit key for encryption and 128-bit key for HMAC provide enough security. Fernet utilizes symmetric-key cryptography as follows:

1. SHA256 HMAC for signing.
2. AES 128 in CBC mode with a 128-bit Initialization Vector (IV) for encrypting the payload of a token

If we have a close look at a Fernet token, we can distinguish five parts:

Version: The first byte of every Fernet token shows the version of the token. The first version is denoted by 128 (0x80), and since there is only one version of Fernet that has been introduced, we can say every Fernet token starts with the number 128.

Timestamp: The second part of a Fernet token is a 64-bit unsigned big-endian integer that represents the creation timestamp of the token in Linux time format.

IV: IV is a 128-bit random string that is used as the initial vector of AES encryption.

Ciphertext: OpenStack Stein has ten different payload types, including unscoped, domain scoped, project scoped, trust scoped, federated unscoped, federated project scoped, federated domain scoped, OAuth scoped, and system scoped. Depending on the type, the data in the payload of a token varies. The payload of a token at least contains the type, the owner, the method that was used for authenticating the token owner, the expiration time, and the audit ID of the token. The length of a payload must be a multiple of 128 bits. Hence, OpenStack uses the PKCS #7 v1.5 technique [15] to pad the payload. Then, OpenStack encrypts the payload.

HMAC: The last part of a Fernet token is the HMAC of all four previous parts, which is calculated by using SHA256 HMAC. This part is known as the signature of a Fernet token.

2.4. Comparison with Existing Token Mechanisms

Existing token-based authentication mechanisms, such as JWT, OAuth bearer tokens, and Fernet tokens, primarily follow a bearer model, where possession of a valid token is sufficient to access system resources. While these mechanisms provide authentication and integrity guarantees, they do not restrict how a token can be used once issued.

In contrast, RAF introduces several key differences:

- **Command Binding:** Unlike JWT, OAuth, and Fernet tokens, which grant general access privileges, RAF tokens are bound to specific commands. This ensures that a token can only be used for its intended purpose.
- **Single-Use Semantics:** RAF tokens are designed to be used at most once, enforced through the blacklist mechanism. In contrast, traditional bearer tokens remain valid until expiration or revocation.

- **Recursive Derivation:** RAF enables controlled delegation by allowing tokens to be derived from parent tokens while preserving the command chain. Existing mechanisms typically rely on separate delegation frameworks without enforcing such chaining.
- **Misuse Limitation:** In RAF, even if a token is leaked, the adversary is restricted to the commands embedded in the token. In contrast, leaked bearer tokens allow arbitrary actions within their scope.
- **Integration with System Policies:** RAF incorporates a policy enforcer to ensure that derived commands follow valid execution flows, providing an additional layer of protection beyond cryptographic guarantees.

Overall, the RAF extends traditional token-based authentication by combining cryptographic guarantees with system-level enforcement to limit token misuse and improve security in modular and distributed environments.

3. Recursive Augmented Fernet (RAF) Mechanism

As mentioned earlier, in the current authentication mechanisms, whenever an adversary captures a user token, they can do all the things that the owner of the token can do (the bearer token problem). The Recursive Augmented Fernet (RAF) mechanism is our solution for the bearer of this problem. The solution includes an RAF, a policy enforcer (PE) component, and a blacklist. This section explains the solution. The main idea is as follows:

- Each user is granted a Fernet token from the identity module, much like the common OpenStack authentication process, i.e., the user presents their credentials to Keystone, and Keystone returns a Fernet token after validating the credentials.
- The user keeps the Fernet token secret and does not pass the token to other modules. For every API request, the user generates a new RAF token using the Fernet token. The RAF token contains enough information about the API and the user request, and it is unforgeable. The user puts the RAF token in the header of the API request instead of the Fernet token.
- Whenever a service receives an API request, it takes the RAF token and validates the RAF with the identity module. Compared to the Fernet mechanism, if the RAF were valid, the identity module returns extra information about the target API and the user request. The PE verifies if the API request matches the information returned by the identity module.
- Whenever a service needs to call another service on behalf of the user, it derives a new RAF token from the token it has received and uses the new token for its request.
- Each RAF token carries enough information about a user request. Using this information and a blacklist, the identity module only verifies an RAF token pointing to the user request *once* for each service.

3.1. RAF Tokens

The core of our solution is the RAF token. To explain the RAF structure, first, we define the following terms:

Root token: This is a Fernet token that a user obtains from Keystone and uses to generate RAFs. T_0 represents a root token.
Base-RAF token: This is an RAF token derived from a root token by users. T_1 represents a base-RAF token.
Service-RAF token: This is an RAF token that is derived from a base-RAF or another service-RAF. Service-RAFs are generated by the internal services and denoted by T_i where $i > 1$.
Parent token: If a token T_i is derived from T_{i-1}, we say T_{i-1} is the parent of T_i.
Child token: If a token T_i is derived from another token T_{i-1}, we say T_i is a child of T_{i-1}.

RAF digest: In a big picture, a Fernet or RAF token T_i is a message m_i concatenated with the HMAC of the message under a certain key. Hence, we can represent a token $T_i = m_i \parallel HMAC(m_i)$. In our solution, the $HMAC(m_i)$ is called the RAF digest of a token and denoted by H_i. Hence, we may write $T_i = m_i \parallel H_i$ where $H_i = HMAC(m_i)$.
Parent Message: If $T_i = m_i \parallel H_i$ was derived from the token $T_{i-1} = m_{i-1} \parallel H_{i-1}$, then m_{i-1} is the parent message of T_i.

3.1.1. Structure

We define the following structure for an RAF token:

$$V \parallel L^m_{i-1} \parallel m_{i-1} \parallel E_i \parallel R_i \parallel C_i \parallel H_i \tag{1}$$

where:

Version (V): A byte shows the version of the token. We identify the first version by 0x91. This field enables a system to support multiple versions of RAFs in the future.
Length of parent message (L^m_{i-1}): To extract the parent message of a token, we need to know its length. This field of a short integer (2 bytes) determines the length of the parent message in bytes.
Parent message (m_{i-1}): The authentication module needs to extract the parent message of a token to verify the token.
Expiration time (E_i): A 64-bit unsigned big-endian integer, which shows the expiration time of the token. It is calculated based on the number of seconds past 1 January 1970.
Randomizer (R_i): A 64-bit random number used to randomize an RAF token. Without this field, although it is unlikely, two separate RAF tokens can be identical if the same user simultaneously issues them using a single Fernet token for the same purpose with the same parameters.
Command (C_i): This is the command that we want to execute using the token. The command field is variable length. Its length is computed from the total decoded token length after subtracting the lengths of all other fields:

$$|C_i| = |T_i| - \big(|V| + |L_{m_{i-1}}| + |m_{i-1}| + |E_i| + |R_i| + |H_i|\big).$$

In our implementation, C_i is encoded as a byte string representing the requested API operation and its parameters. Since H_i is fixed at 256 bits, the parser identifies C_i as all remaining bytes between R_i and H_i.
HMAC (H_i): This field is a 256-bit SHA256 HMAC that is used for authenticating the content of a token. The signing key of the HMAC and the fields included in the HMAC vary based on the token type, as explained in the following section.

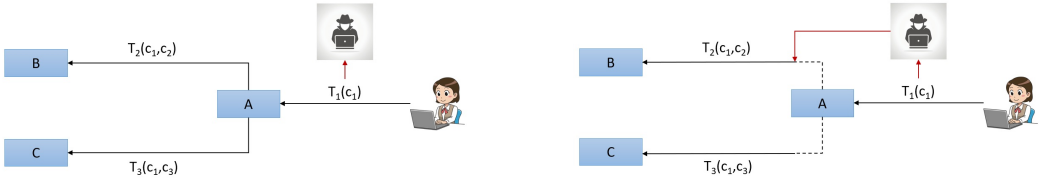
3.1.2. Variations

We introduce two types of RAFT: (1) user-tied RAFT (UT-RAFT) and (2) fully-tied RAFT (FT-RAFT). The structure of both tokens is the same. However, they use different signing keys and include different fields in HMAC as follows:

User-tied RAF: In this type, to derive a token from a parent token (received token), we take the first 128 bits of the parent key and use it for signing the new token. The HMAC field is the HMAC of all the fields prior to it in the token. This approach guarantees the authenticity of the user command carried in an RAF token.
Fully-tied RAF: In this type, the process of issuing a new RAF token for a user is the same as the user-tied version. However, each service has a long-term symmetric key shared with the identity module and uses this key for signing tokens. The HMAC field is the HMAC of all the fields prior to it in the token, plus the parent key of the token. Consequently,

the RAF token is tied to the command and the service, and the identity module can verify the authenticity of the user and services that have contributed to an RAF token.

Figure 2 shows the effect of a leaked token in two types of RAF with an example. In the example, the user issues the token T_1 for her command (request) C_1 and sends T_1 to the service A. To accomplish C_1, A needs to send the C_2 and C_3 commands to B and C, respectively. As shown in the picture, T_1 is leaked to the adversary. In the fully-tied RAF, the adversary cannot forge any valid RAF token using T_1, but in the user-tied RAF, the adversary can forge several valid RAF tokens. However, the point is that all of them carry C_1, and if the adversary puts a command that does not align with C_1, the PE component (described later in this paper) will reject the token.



(**a**) Fully-tied RAF. (**b**) User-tied RAF.

Figure 2. The effect of a leaked token in two types of RAF.

The advantage of the fully-tied token over the user-tied token is that:

- It guarantees stronger security (see Section 4).
- It is not (computationally) possible to forge an RAF token on behalf of a module unless the module was corrupted.

However, since in a fully-tied RAF, each module must have a secret key shared with the identity module, we need to adopt a proper key distribution and key renewal mechanism, which in turn induces extra efforts. While we possess the knowledge of how to perform key renewal [16–18], the challenge lies in determining an appropriate key renewal and management strategy for large communities like OpenStack, which entails reaching a consensus within the OpenStack community regarding the security posture that the key renewal methodology should adopt. The user-tied RAF does not require such discussion. The fully-tied RAF needs an adaptation/extension of the current key renewal practice.

3.1.3. Generation

Algorithms 1 and 2 show the steps for issuing a user-tied RAF token, and Algorithms 1 and 3 represent the steps for issuing a fully-tied RAF token. In both RAF variants, users run the same algorithm. As shown in the algorithms, given a parent token $Token = Pm \parallel Pkey$, a command Cmd, the lifetime $LifeTime$, and the service keys $KeySet$ (for fully-tied RAF), we follow the steps below to issue an RAF token:

1. Unpack the parent token and get the parent message and parent key.
2. Choose the proper signing key.
3. Put the value 0x91 in the version field.
4. Set the length of the parent message field.
5. Set the parent message.
6. Add the token lifetime to the current time and put the result in the expiration-time field.
7. Choose an eight-byte random number for the randomizer field.
8. Set the command.
9. For a user-tied token, compute the HMAC of all the above fields using the parent key.
10. For a fully-tied token, compute the HMAC of all the above fields plus the parent key using the secret key of the service.

11. Base64url encode the entire token.

Algorithm 1 Pseudocode of Issuing a user-tied/fully-tied RAF token by a User

Function *UserIssue(Key, Pm, Cmd, LifeTime)*
$SignKey = HMAC(Key, Pm)$
$V = 0x91$
$L = Length(Pm)$
$E = GetCurrentTime() + LifeTime$
$R \leftarrow Gen(1^{128})$
$Payload = V \parallel L \parallel Pm \parallel E \parallel R \parallel Cmd$
$Signature = HMAC(SignKey, Payload)$
$Token = base64.Encode(Payload \parallel Signature)$
return *Token*
End Function

Algorithm 2 Pseudocode of Issuing a user-tied RAF token by Services

Function *ServiceIssue(Key, Token, Cmd, LifeTime)*
if $Verify(Key, Token) = 1$ **then**
$(Pm, PKey) = Unpack(Token)$
$V = 0x91$
$L = Length(Pm)$
$E = GetCurrentTime() + LifeTime$
$R \leftarrow Gen(1^{128})$
$Payload = V \parallel L \parallel Pm \parallel E \parallel R \parallel Cmd$
$Signature = HMAC(PKey, Payload)$
$Token = base64.Encode(Payload \parallel Signature)$
return *Token*
else
return *nothing*
end
End Function

Algorithm 3 Pseudocode of Issuing a fully-tied RAF token by services

Function *ServiceIssue(i,KeySet, Token, Cmd, LifeTime)*
if *Verify*($k_0 \in KeySet$,*Token*)=*1* **then**
$Pm, PKey \leftarrow Unpack(Token)$
$SignKey = k_i \in KeySet$
$V = 0x91$
$L = Length(Pm)$
$E = GetCurrentTime() + LifeTime$
$R \leftarrow Gen(1^{128})$
$Payload = V \parallel L \parallel Pm \parallel E \parallel R \parallel Cmd$
$Signature = HMAC(SignKey, Payload \parallel PKey)$
$Token = base64.Encode(Payload \parallel Signature)$
return *Token*
else
return *nothing*
end
End Function

In Section 5, we show the implementation of Algorithms 1 and 2 in Python 3.1 code.

3.1.4. Verification

Algorithm 4 shows the pseudocode of a user-tied RAF token verification. Given the secret key of Keystone $k_{keystone}$ and a token, to verify that the token is a valid user-tied

token and recover all the commands embedded in the token, perform the following steps in order:

1. Base64url decode the token.
2. If the first byte of the token is not 0x91, then the token is not valid. Therefore, raise an exception error.
3. Unpack the token.
4. Check the expiration time of the token in m_i.
5. Using the length-of-parent-message field (second and third bytes), retrieve m_{i-1}. If m_{i-1} is an RAF message (i.e., its first byte is 0x91), check the expiration time of m_{i-1} and retrieve m_{i-2} from m_{i-1}. Recursively continue this step until you get m_0 , which must be a Fernet message (i.e., its first byte must be 0x80).
6. Calculate $key_0 = HMAC(k_{keystone}, m_0)$.
7. Having key_0, backtrack step 5 and calculate all parent keys $\{key_1, \cdots key_{i-1}\}$ computing $key_j = HMAC(key_{j-1}, m_j)$.
8. Having key_{i-1}, recalculate the HMAC of the given token. If the calculated HMAC is equal to the HMAC of the given token, then verify the original Fernet token $m_0 \parallel key_0$.
9. Extract the Fernet message from m_0 and all commands from $\{m_1, \cdots, m_i\}$ and return them.

Verification of a fully-tied RAF token is slightly different from verification of a user-tied token. In a fully-tied RAF, each service also has a secret key, and the service uses its secret key to sign tokens. Algorithm 5 is the pseudocode for a fully-tied token verification. Given all the secret keys of the authentication module and other services $KeySet = \{k_0, \cdots, k_n\}$ and a token, to verify that the token is a valid, fully-tied token and recover all the commands embedded in the token, perform the following steps in order:

1. Base64url decode the token.
2. If the first byte of the token is not 0x91, then the token is not valid. Therefore, raise an exception error.
3. Unpack the token.
4. Check the expiration time of the token in m_i.
5. Using the length-of-parent-message field (second and third bytes), retrieve m_{i-1}. If m_{i-1} is an RAF message (i.e., its first byte is 0x91), check the expiration time of m_{i-1} and retrieve m_{i-2} from m_{i-1}. Recursively continue this step until you get m_0 , which must be a Fernet message (i.e., its first byte must be 0x80).
6. Calculate $tag_0 = HMAC(k_0, m_0)$, where k_0 is the secret key of Keystone.
7. Having tag_0, backtrack step 5 and calculate all the tags of ancestor tokens $\{tag_1, \cdots tag_{i-1}\}$ computing $tag_j = HMAC(k_j, m_j \parallel tag_{j-1})$.
8. Having tag_{i-1}, recalculate the HMAC of the given token. If $HMAC(k_i, m_i \parallel tag_{i-1}) = tag_i$, then verify the original Fernet token $m_0 \parallel tag_0$.
9. Extract the Fernet message from m_0 and all commands from $\{m_1, \cdots, m_i\}$ and return them.

Trade-Offs Between User-Tied and Fully-Tied RAF: As shown in Table 2, user-tied and fully-tied RAF tokens offer different trade-offs in terms of usability, scalability, and security.

User-tied RAF tokens are easier to deploy and integrate, as they do not require services to maintain or share secret keys with the identity module. This makes them more suitable for large-scale or loosely coupled systems, where minimizing key management complexity is important. However, their security guarantees are weaker, as a compromised module may still derive new tokens within the constraints of the original command chain.

In contrast, fully-tied RAF tokens provide stronger security guarantees by binding each token to both the command and the issuing service. This prevents adversaries from forging tokens on behalf of other services unless those services are compromised. However, this approach requires maintaining service-specific secret keys and introduces ad-

ditional complexity in key distribution and renewal, which may impact scalability and operational overhead.

Overall, user-tied RAF offers better usability and scalability, while fully-tied RAF provides stronger security guarantees at the cost of higher system complexity.

Table 2. Comparison between user-tied and fully-tied RAF tokens.

	Usability	**Scalability**	**Security**
User-tied RAF	High	High	Moderate
Fully-tied RAF	Moderate	Moderate	High

Algorithm 4 Pseudocode of a user-tied RAF token Verification

Function *Verify(Key, Token)*
  $Token = base64.decode(Token)$
  $V, L, Pm, E, R, Cmd, Tag \leftarrow Unpack(Token)$
  if $V = 0x91$ **then**
    $CheckExpirationTime(E)$
    $Cmds, PKey = ValidateParent(Key, Pm)$
    $AccomulateCommands(Cmds, Cmd)$
    if $HMAC(PKey, V \parallel L \parallel Pm \parallel E \parallel R \parallel Cmd) = Tag$ **then**
      return $Valid, Cmds$
    else
      return $Invalid$
    end
  else
    return Invalid;
  end
End Function
Function *ValidateParent(Key,Message)*
  if *First byte of Message is* $0x91$ **then**
    $(V, L, Pm, E, R, Cmd \leftarrow Unpack(Message)$
    $CheckExpirationTime(E)$
    $Cmds, PKey = ValidateParent(Key, Pm)$
    $AccomulateCommands(Cmds, Cmd)$
    $Tag = HMAC(PKey, V \parallel L \parallel Pm \parallel E \parallel R \parallel Cmd)$
    return $Cmds, Tag$
  else
    return $"", HMAC(Key, Message)$
  end
End Function

3.2. Blacklist

The blacklist not only prevents replay attacks but also guarantees that each module serves at most once for every user command. Every RAF token is either a base-RAF token or has been derived from a base-RAF token. Hence, we can say every valid RAF token has a unique base-RAF token. We want each base-RAF token to be a permit for only one task in the system. For example, if a user generates a base-RAF token for accessing an image, we should guarantee that the image will be accessed at most once using this user-RAF. To do so, we propose to implement a blacklist mechanism in the token validation process. Each entry in the blacklist is a tuple (B, S, E) that is added to the list upon a successful token validation, where

S is the service that asked the authentication module to validate the token T.
B is the base-RAF token of T.
E is the expiration time of B.

Algorithm 5 Pseudocode of a fully-tied RAF token verification

Function *Verify(KeySet, Token)*
  $Token = base64.decode(Token)$
  $(V \parallel L \parallel Pm \parallel E \parallel R \parallel Cmd \parallel Tag) \leftarrow Unpack(Token)$
  if $V = 0x91$ **then**
    $CheckExpirationTime(E)$
    $k_i = FindServiceKey(Cmd)$
    $Cmds, PTag = ValidateParent(KeySet, Pm)$
    $AccomulateCommands(Cmds, Cmd)$
    if $HMAC(k_i, V \parallel L \parallel Pm \parallel E \parallel R \parallel Cmd \parallel PTag) = Tag$ **then**
      return $Valid, Cmds$
    else
      return $Invalid$
    end
  else
    return Invalid;
  end
End Function
Function *ValidateParent(KeySet,Message)*
  if *First byte of Message is* $0x91$ **then**
    $(V \parallel L \parallel Pm \parallel E \parallel R \parallel Cmd) \leftarrow Unpack(Message)$
    $CheckExpirationTime(E)$
    $k_i = FindServiceKey(Cmd)$
    $Cmds, PTag = ValidateParent(KeySet, Pm)$
    $AccomulateCommands(Cmds, Cmd)$
    $Tag = HMAC(k_i, V \parallel L \parallel Pm \parallel E \parallel R \parallel Cmd \parallel PTag)$
    return $Cmds, Tag$

  else
    return "", $HMAC(k_0, Message)$
  end
End Function

When S asks the authentication module to validate T, it extracts B out of T. It also extracts the expiration time E out of B. Since for executing user commands each module is needed at most once, Keystone only validates a token when its corresponding tuple (B, S, E) was not in the blacklist. Keystone periodically cleans up the blacklist and keeps only the tuples that have not expired. A tuple (B, S, E) is expired when E is less than the system time. Since the lifetime of an RAF token is usually very short, a lightweight memcache structure works fine.

Scalability Considerations: The blacklist component is designed to operate efficiently under typical workloads due to the short lifetime of RAF tokens. Since each token is intended for a single operation, its lifetime is expected to be short. As a result, blacklist entries expire quickly, allowing the system to maintain a bounded memory footprint through periodic cleanup.

In distributed environments, the blacklist can be implemented using shared in-memory data stores (e.g., memcache or Redis), which support high-throughput lookups with low latency. Since each validation requires only a constant-time membership check, the overhead introduced by the blacklist remains minimal in practice.

Nevertheless, at very high request volumes, the blacklist may become a performance bottleneck if not properly provisioned. This can be mitigated through standard techniques such as sharding, replication, or partitioning the blacklist across multiple nodes. Additionally, because RAF tokens are short-lived, the number of active entries remains limited, reducing both memory usage and long-term storage overhead.

Overall, the blacklist introduces modest overhead while providing strong protection against replay attacks and enforcing the single-use property of RAF tokens.

3.3. Policy Enforcer (PE)

If the adversary obtains a user-tied RAF token, they can produce children of the token. Also, in a fully-tied RAF, if the adversary knows the secret key of a module (corrupts the module), they can generate children for the tokens sent to the module. RAF (either user-tied or fully-tied) excludes an adversary from producing a token without knowledge of one of its parents/ancestors. The PE is used to make sure that the command embedded in an RAF token is consistent and obeys a proper execution flow (specified by a policy). This means that even though an adversary can create child tokens, they can only do so using commands that fit the PE. Figure 3 demonstrates the idea using OpenStack. Here, the user creates a token T for the "list image" command and sends it to Glance, which is compromised by the adversary. Hence, the adversary can create a child token like T'. If the adversary adds an arbitrary command like "delete a server" to the new token and sends the token to Nova, since the "delete server" command does not relate to the "list image" command, the PE of Nova will reject the request.

The PE completes our solution.

Policy Definition and Enforcement: The PE operates based on predefined rules that specify valid relationships between commands in a request chain. These policies can be derived from the system's API semantics or workflow definitions, ensuring that only logically consistent sequences of operations are allowed.

During execution, when an RAF token is validated and its embedded command sequence is extracted, the PE checks whether the requested command is consistent with the preceding commands in the chain. If the command does not conform to the policy (e.g., an unrelated or unauthorized operation), the request is rejected.

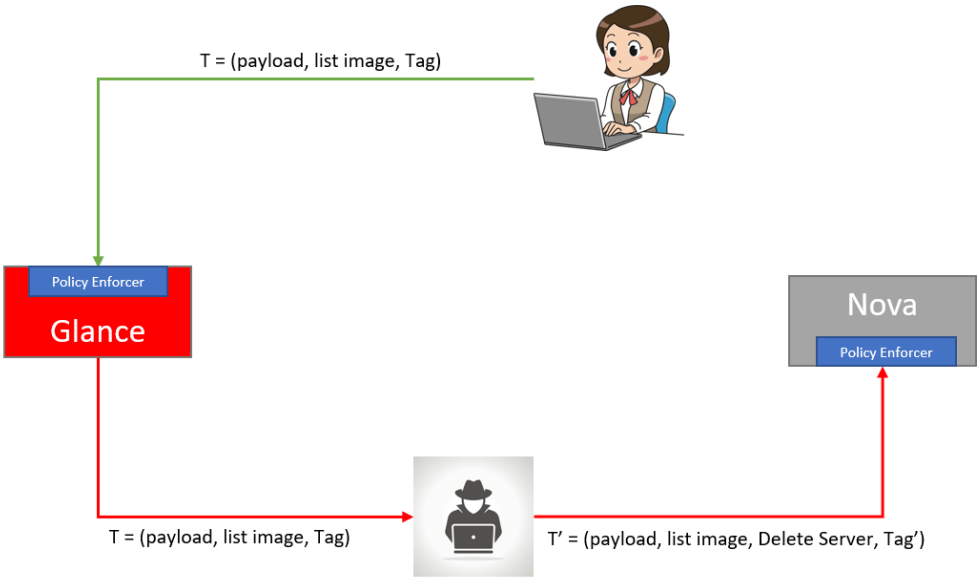


Figure 3. Demonstration of the usage of policy enforcer.

Policies are maintained at the system level and can be updated as part of normal system configuration or API evolution. The PE does not require changes to the cryptographic structure of RAF tokens, allowing flexible policy updates without affecting token generation or validation.

Finally, conflicts between commands are handled implicitly by the policy rules. Any command that violates the defined execution flow is treated as invalid and rejected, ensuring that adversarial attempts to inject or modify commands do not succeed.

3.4. RAF Token Lifecycle

The lifecycle of an RAF token consists of four main stages: generation, propagation, validation, and consumption.

Generation: A user initially obtains a Fernet token from the identity module. For each API request, the user generates an RAF token by embedding the intended command and deriving it from the Fernet token. When a service needs to invoke another service on behalf of the user, it derives a new RAF token from the token it has received.

Propagation: RAF tokens are passed between services as part of API requests. Each derived token preserves the command chain through its recursive structure, ensuring that all downstream requests remain linked to the original user intent.

Validation: Upon receiving an RAF token, a service forwards it to the identity module for verification. The identity module validates the token recursively, checks its integrity and expiration, and extracts the sequence of embedded commands. As part of this process, the system checks the blacklist to ensure that the token has not been used previously. If the token has already been used, the request is rejected.

Consumption: If the token is valid and has not been used before, the associated command is evaluated by the PE of the service. If the command is not a legitimate request from that service, the request is rejected. Otherwise, the command is executed.

Overall, this lifecycle ensures that each token is tightly bound to a specific request, can only be used once, and preserves the integrity of the command chain across multiple services.

4. Security Guarantees

The aim of RAF tokens is to prevent an adversary from sending a request to a modular system on behalf of the user. At a high level, a RAF token is an algorithm that gets a message as input and outputs a token. Security is formulated by requiring that no adversary can generate a valid token on any new message. For a user-tied RAF token, a new message is considered to be any message that the message itself or its ancestors were not previously sent by a user or other modules. For a fully-tied RAF, only the message itself should not be previously sent by the user or other modules.

In this section, we want to define the security that each type of RAF mechanism provides under the threat model defined in Section 1. In order to do this, we take the following steps:

- First, we present the technical definition of what a user/fully-tied RAF is.
- Next, we define an experiment as a game between the RAF mechanism and the adversary. The experiment shows how an adversary interacts with the mechanism and in which circumstances the adversary may successfully forge an RAF token and win the game.
- Using this game and the technical definition of the RAF mechanism, we provide the definition of the RAF token security. The definition states that no PPT adversary should win the game with non-negligible probability.
- Finally, we construct a secure user/fully-tied RAF using the algorithms defined in Section 3.1.3 and define a lemma that indicates that our construction is secure and unforgeable against a chosen-message attack.

The RAF token comes in two types: a user-tied and a fully-tied RAF token. In Section 4.1 we argue about the security guarantees of a user-tied RAF mechanism, while in Section 4.2 we focus on the security of fully-tied RAF.

4.1. The Security Analysis of User-Tied RAF

A user-tied RAF token prevents the adversary from impersonating a legitimate user. Before defining the security features of this mechanism, we first provide a technical definition of user-tied RAF. Technically, a user-tied RAF is made up of four algorithms *Gen*, *UserIssue*, *ServiceIssue*, and *Ver*. The algorithm *Gen* generates a secret key; we assume that upon input of the security parameter λ, the algorithm *Gen* outputs a uniformly distributed string of length λ. The algorithms *UserIssue* and *ServiceIssue* receive a message/token (p/r), a command c and a key k as input and generate a token. *UserIssue* is used by the user dashboard, while *ServiceIssue* is used by the other services. Finally, the algorithm *Ver* receives a token r and a key k and outputs either 1 (meaning valid) or 0 (meaning invalid). The formal definition is as follows:

Definition 1. *A user-tied RAF token is a quadruple* $\Pi = (Gen, UserIssue, ServiceIssue, Ver)$ *of ppt algorithms where:*

- $k \leftarrow Gen(1^\lambda)$ *, where Gen generates a key k with security parameter* λ
- $r_0 = (m_0, t_0) \leftarrow UserIssue(k, p, c_0)$*, where UserIssue generates a token* r_0 *for the root payload p and the command* c_0 *with the key k.*
- $r_j = (m_j, t_j) \leftarrow ServiceIssue(k, r_{j-1}, c_j)$*, where if* r_{j-1} *is internally verified by* $Ver(k, r_{j-1})$ *then ServiceIssue generates a token* r_j *for the input token* r_{j-1} *and the command* c_j *with the key k; Otherwise it returns nothing.*
- $b \leftarrow Ver(k, r_n = (m_n, t_n))$*, where Ver takes the key k and a token* r_n*. It outputs b, with* $b = 1$ *meaning the token is valid and* $b = 0$ *meaning the token is not valid.*
- *It is called correct if for every k output by* $Gen(1^\lambda)$ *and every token r generated by UserIssue or ServiceIssue; it holds* $Ver(k, r) = 1$*.*

A user-tied RAF token must provide the following properties:

- Represents a user command.
- Enables a module to extend a token with a command.
- Prevents an adversary from forging any token except some descendants of the eavesdropped tokens.

A descendant of a token contains all the commands in the token. If the adversary adds an irrelevant command to a child token of an eavesdropped token, it will be rejected by the policy enforcer (see Section 3.3) of other modules. Hence, the ability to forge a child token is tolerated.

Now, we define the following game for a PPT adversary $\mathcal{A}$, the security parameter λ, and a user-tied RAF token $\Pi = (Gen, UserIssue, ServiceIssue, Ver)$:

Game 1. ***Forge a User-tied RAF token*** $ForgeUTT_{\mathcal{A},\Pi}(\lambda)$***:***

1. $Gen(1^\lambda)$ *generates a key k*
2. *The adversary* $\mathcal{A}$ *is given input* 1^λ *and oracle access to* $Ver(k, .)$*,* $UserIssue(k, ., .)$*, and* $ServiceIssue(k, ., .)$ *and outputs* $r = (m, t)$*. Let* Φ *represent the set of all tokens that* $\mathcal{A}$ *has queried.*
3. $\mathcal{A}$ *wins if and only if (1)* $Ver(k, r = (m, t)) = 1$ *and (2) r and any of its ancestors are not in* Φ*.*
4. *The experiment returns 1 if the adversary wins the game; otherwise, it returns 0.*

The definition of Game 1 covers our goal. It ties a token to the user command and allows a service to extend the token with a new command. Since the adversary has oracle access to *UserIssue* and *ServiceIssue*, it covers the situation that the adversary eavesdrops on some tokens. It states that no PPT adversary should be able to generate a valid user-tied RAF token unless they has seen at least one of its ancestors. This guarantees that the adversary cannot forge a token with a fake user command.

We know a function $f(x) : \mathbb{N} \to \mathbb{R}$ is negligible if for every positive polynomial $poly(\cdot)$, there exists an integer $N_{poly} > 0$ such that for all $x > N_{poly}$, it holds that $|f(x)| < \frac{1}{\text{poly}(x)}$.

Definition 2. *A user-tied token $\Pi = (Gen, UserIssue, ServiceIssue, Ver)$ is unforgeable under a chosen-message attack, or just secure, if for all PPT adversaries $\mathcal{A}$ there exists a negligible function **negl** such that:*

$$\Pr[ForgeUTT_{\mathcal{A},\Pi}(\lambda) = 1] \leq negl(\lambda). \tag{2}$$

Using Definition 1 and Algorithms 1, 2 and 4, we construct a user-tied RAF token. We then prove that our construction is a secure user-tied RAF.

Definition 3. *We say HMAC is secure and indistinguishable from a random function, if for all probabilistic polynomial-time distinguishers D, there exists a negligible function α such that:*

$$|\Pr[D^{HMAC(k,.)}(\lambda) = 1] - \Pr[D^{f(.)}(\lambda) = 1]| \leq \alpha(\lambda) \tag{3}$$

where $f(.)$ is a random function and $\alpha(\lambda)$ is a negligible function in λ.

Lemma 1. *Assume that the HMAC used in Construction 1 is secure and indistinguishable from a random function. Then, Construction 1 is a secure user-tied token that is unforgable under chosen-message attacks.*

Construction 1. ***User-Tied RAF***
Let HMAC be indistinguishable from a random number. Define

$$RAFT = (Gen, UserIssue, ServiceIssue, Ver)$$

as follows:

- *$Gen(1^\lambda)$: Upon input 1^λ, choose $k \leftarrow \{0,1\}^\lambda$*
- *$UserIssue(k, p, c)$: Upon input key k, payload p and the command c compute r by running Algorithm 1.*
- *$ServiceIssue(k, r', c')$: Upon input key $k \in \{0,1\}^\lambda$, token r' and the command c' compute r by running Algorithm 2.*
- *$Ver(k, r = (m, t))$: Upon input key $k \in \{0,1\}^\lambda$, token r compute b by running Algorithm 4.*

In this section, we only present an informal sketch of the proof of this lemma. We defer to Appendix A for the full proof.

Proof sketch. Our argument proceeds via a reduction approach; that is, we show how an adversary breaking Construction 1 can be used as a subroutine to violate the assumption that "*HMAC is secure and indistinguishable from a random function.*"

To be more precise, we first assume, by contradiction, that a polynomial-time adversary $\mathcal{A}$ manages to forge a user-tied *RAF* (breaks Construction 1), i.e., with a non-negligible

probability ϵ, the user is able to win Game 1. Next, we build a new scheme, RAF' to be the same as user-tied RAF except that instead of using the $HMAC$ for key k in its algorithms, i.e., in $UserIssue$ and Ver, it uses a random function. We show that this implies the existence of a polynomial-time algorithm, called distinguisher $\mathcal{D}$, that can distinguish the $HMAC$ from a random one with advantage $O(\varepsilon)$. This will then imply that ε must be negligible; otherwise, it will contradict the assumption that $HMAC$ is secure.

More precisely, we show that the user-tied RAF is secure, and the upper bound on the probability of forging RAF is about a factor n times ϵ, where n is the number of commands added in the forged RAFT token. This upper bound makes sense: Let us consider that a token can at most contain n commands; we can imagine these commands leading to an HMAC chain. In the proof methodology, the attacker $\mathcal{A}$ can be successful in impersonating any spot in the chain by using the command c_i. This means that there are n possibilities, and thus the upper bound is a factor of n. By observing that $n = poly(\lambda)$ for a PPT adversary $\mathcal{A}$, the proof is completed. □

Discussion. Lets assume that the HMAC, which is used in the implementation has $\alpha(\lambda) = \frac{1}{2^\lambda}$, then the lemma gives a concrete security statement with $\epsilon(\lambda) \leq (n+1) \cdot \frac{1}{2^\lambda} + \frac{1}{2^\lambda - C}$, where n is the number of modules in the system and $C = poly(\lambda)$ is the adversary's maximum number of queries. Note that because of the blacklist component, n cannot be more than the number of modules in a system.

We introduce the notation $\alpha_C(\lambda) = \frac{1}{2^\lambda - C}$. This implies that $\alpha(\lambda) < \alpha_C(\lambda)$. In the case that the adversary has made C queries to the HMAC(k,.) vs f(.) oracle, then we can say, $\epsilon(\lambda) \leq (n+2) \cdot \alpha_C(\lambda) = (n+2) \cdot \frac{1}{2^\lambda - C}$. If key k does not change for a long time, this implies that the adversary can make more queries to the oracle $\mathcal{O}$ (C gets too large) and therefore, ϵ gets too large, i.e., the adversary's chance of winning the game increases. In order to keep ϵ low, we must implement a key renewal strategy in Keystone. In fact, OpenStack has this strategy implemented, and by default, Keystone's key is changed every one hour.

Statement: The implementation of the discussed and proven secure RAF mechanism provided in Section 5, is a one-time token.

In order to have a one-time RAF token, we have considered and implemented an additional component: a blacklist. The blacklist ensures that each module cannot request a validation for the same token more than one time.The blacklist component must be added to the authentication module and functions as follows: Each time the authentication module receives a token, first, it checks if the token is valid, i.e., the Ver function outputs 1, if so, then it checks if the pair (base-RAF, service ID) is in the blacklist. If not, then the authentication module returns valid; otherwise, it returns invalid (for more detail on blacklist functionality, see Section 3.2).

This indicates that each token with the same base-RAF token can only be validated once, which implies that the RAF token implementation is a one-time token.

4.2. The Security Analysis of Fully-Tied RAF

The fully-tied RAF is similar to the user-tied RAF except it has an additional feature that prevents the adversary from sending a token on behalf of a service. In other words, if the adversary is able to get access to a fully-tied token, s/he is not able to send a request on behalf of an uncorrupted service to another service.

In this section, we show the security guarantees of a fully-tied RAF. We first provide a technical definition of a fully-tied RAF token. Next, we define a game between the adversary and the fully-tied RAF: In this game the adversary is given oracle access to the RAF function and is able to make queries. At some point, the adversary generates a token

for a message that they have not queried. The adversary wins the game if the provided token passes the validation.

We also give the security definition of fully-tied RAF and construct a secure fully-tied RAF using the algorithms in Section 3.1.3. Finally, we wrap up this section with a lemma that shows our construction is secure against a chosen-message attack.

Definition 4. *A fully-tied token is a quadruple* $\Pi = (Gen, UserIssue, ServiceIssue, Ver)$ *of ppt algorithms where:*

- $K = (k_0, \cdots, k_{n-1}) \leftarrow Gen(1^\lambda)$ *with security parameter* λ.
- $r_0 = (m_0, t_0) \leftarrow UserIssue(k_0, p, c_0)$ *generates a token* r_0 *for the root payload* p *and the command* c_0 *with key* k_0.
- $r_j = (m_j, t_j) \leftarrow ServiceIssue(i, K, r_{j-1}, c_j)$ *internally call* $Ver(K, r_{j-1})$ *and if* r_{j-1} *is invalid, then it returns nothing; otherwise, it generates a token* r_j *for the input token* r_{j-1}*, the service ID* i*, and the command* c_j *with the key* k_i*. Note that ServiceIssue consists of two phases: (1) Verification, in which it calls Ver algorithm and verifies the validity of the token* r_{j-1}*. For this part,* K *is needed. (2) Token generation, where token* r_j *is computed using the key of the service with service ID* i *(i.e.,* k_i*).*
- $b \leftarrow Ver(K, r_n = (m_n, t_n))$*, where Ver takes a set of keys* K *and a token* r_n*. It outputs* b*, with* $b = 1$ *meaning the token is valid and* $b = 0$ *meaning the token is not valid.*
- *It is called "correct" if, for every* K *output by* $Gen(1^\lambda)$ *and every token* r *generated by UserIssue or ServiceIssue,* $Ver(K, r) = 1$ *it holds.*

According to the above definition, a fully-tied token gets a payload and extends it with a command. This definition looks similar to Definition 1. However, an important difference exists: here, each module has its own secret key, while in Definition 1, modules do not own/use any secret keys. This enables modules to add a new command to the token and sign the token with their secret keys.

We define the following game for a PPT adversary $\mathcal{A}$, the security parameter λ, and a fully-tied token $\Pi = (Gen, UserIssue, ServiceIssue, Ver)$:

Game 2. ***Forge a Fully-Tied Token or for short ForgeFTT****:*

- $K = (k_0, \cdots, k_{n-1}) \leftarrow Gen(1^\lambda)$ *where* $n =$ *Number of Services in the system.*
- *The adversary* $\mathcal{A}$ *is given input* 1^λ *and access to oracles:* $Ver(K, .)$*,* $UserIssue(k_0, ., .)$*, and* $ServiceIssue(i, K, ., .)$ *and knows keys* k_i *for* $i \in I \subseteq \{1, \ldots, n\}$*. Let* Φ *represent the set of all tokens that* $\mathcal{A}$ *has requested. Eventually, the adversary outputs* $r = (m, t)$*.*
- $\mathcal{A}$ *wins if and only if (1)* $Ver(K, r = (m, t)) = 1$*, (2)* $r \notin \Phi$*, and (3)* r *is not an output of* $ServiceIssue(i, K, ., .)$ *for some* $i \in I$*.*
- *The experiment returns 1 if the adversary wins the game; otherwise, it returns 0.*

This game is exactly the same as Game 1 with one extra condition in Step 3, which is "r is not an output of $ServiceIssue(i, K, ., .)$ for some $i \in I$".

Definition 5. *A fully-tied token* $\Pi = (Gen, UserIssue, ServiceIssue, Ver)$ *is unforgeable under a chosen-message attack, or just secure, if for all PPT adversaries* $\mathcal{A}$ *there exists a negligible function* ***negl*** *such that:*

$$\Pr[ForgeFTT_{\mathcal{A},\Pi}(\lambda) = 1] \leq negl(\lambda) \tag{4}$$

A secure fully-tied token guarantees the authenticity of the user and service commands embedded in a token, i.e., not only can the adversary not forge a token with a fake user command, but they also cannot forge a valid token on behalf of any service unless they corrupt the service (know the secret key of the service).

Lemma 2. *Assume that the HMAC used in Construction 2 is secure and indistinguishable from a random function. Then, Construction 2 is a secure, fully-tied token that is unforgeable under chosen-message attacks.*

Construction 2. ***Fully-Tied Token RAF***

Let HMAC be indistinguishable from a random number. Define a fully-tied token $\widehat{RAFT} = (Gen, UserIssue, ServiceIssue, Ver)$ *as follows:*

- $Gen(1^\lambda)$*: Upon input* 1^λ*, choose* $K = (k_0, k_1, ..., k_{n-1}) \leftarrow \{0,1\}^\lambda$ *where* $n =$ *number of services in the system.*
- $UserIssue(k_0, p, c)$*: Upon input key* $k_0 \in \{0,1\}^\lambda$*, payload* p *and the command* c *compute* r *by running Algorithm 1.*
- $ServiceIssue(i, K, r', c)$*: Upon input key* $K = \{k_0, k_1, ..., k_{n-1}\} \leftarrow \{0,1\}^\lambda$ *, token* r' *and the command* c *compute* r *by running Algorithm 3.*
- $Ver(K, r = (m, t))$*: Upon input key set* $K = \{k_0, k_1, ..., k_{n-1}\} \leftarrow \{0,1\}^\lambda$*, token* r *compute* b *by running Algorithm 5.*

We defer to Appendix A for the full proof. Here, we simply highlight the main ideas involved in the proof.

Proof sketch. The intuition behind the proof of this lemma is the same as Lemma 1. Here, our argument also proceeds via a reduction approach.

We argue that the only way for an adversary to win the game is to either generate a valid token, which is a result of *UserIssue* or a valid token, which is the result of *ServiceIssue*. Next, we show that a token generated by *UserIssue* can be represented as a user-tied RAF token with exactly one command, and tokens generated by *serviceIssue* can be represented by a user-tied RAF token with no command. This argument enables us to use Lemma 1 to prove the security of the fully-tied RAF and show its upper bound. □

Discussion. For a fully-tied RAF, the upper bound on the probability of successfully forging a token is much tighter than the upper bound for a user-tied RAF. This is reasonable because in this type of RAF mechanism, each service has its own secret key for applying the *HMAC*. This means even if the adversary gets access to a valid token, they are not able to generate a valid token on behalf of an uncorrupted service.

4.3. Discussion of Security Guarantees

In this section, we clarify the security guarantees provided by the RAF mechanism by distinguishing between formally proven properties and system-level protections enforced by architectural components.

4.3.1. Formally Proven Guarantees

The core cryptographic property of RAF tokens is *unforgeability under chosen-message attacks*. For both user-tied and fully-tied RAF tokens, we formally prove that no probabilistic polynomial-time (PPT) adversary can generate a valid token for a new message unless it has access to at least one of its ancestor tokens. This guarantee is captured through the security games defined in Section 4 and formalized in Definitions 2 and 5. The proofs rely on the assumption that HMAC behaves as a pseudorandom function.

4.3.2. System-Level Guarantees

In addition to the formal guarantees, RAF provides several important security properties at the system level:

- **Replay Prevention:** The blacklist component ensures that each base-RAF token can be validated at most once per service, effectively preventing replay attacks.
- **Command Integrity and Binding:** Each RAF token carries an explicit command, and the recursive structure ensures that all derived tokens are cryptographically bound to the original command chain.
- **Misuse Limitation:** If a token is leaked prior to its use, its usage remains restricted to the specific command(s) embedded within it. Once the token has been used, any subsequent attempt to reuse it is rejected by the blacklist mechanism. Consequently, the impact of a leaked token is strictly limited, significantly reducing potential damage compared to traditional bearer tokens.
- **Execution Consistency:** The policy enforcer ensures that any command derived from a token follows a valid execution flow, preventing adversaries from injecting unrelated or malicious operations.

4.3.3. Threat Coverage

Under the threat model defined in Section 1, RAF provides protection against the following adversarial capabilities:

- **Token Leakage:** An adversary who obtains a token cannot use it to perform arbitrary actions beyond those encoded in the token.
- **Replay Attacks:** Reuse of tokens is prevented by the blacklist mechanism.
- **Token Forgery:** Generating valid tokens without access to ancestor tokens is computationally infeasible under standard cryptographic assumptions.
- **Malicious Command Injection:** Attempts to alter or extend command sequences are rejected by the policy enforcer.

We emphasize that the formal analysis focuses on token unforgeability, while replay prevention and execution consistency are enforced through system-level components (blacklist and policy enforcer).

5. Proof of Concept

Section 4 presented the formal security analysis of the RAF mechanism, including game-based definitions and proofs for user-tied and fully-tied RAF tokens. In this section, we complement that analysis by validating the practicality of RAF through a proof-of-concept implementation using OpenStack as the testbed. Specifically, we developed a Python library and modified Keystone to support RAF token generation and validation, and we evaluate the associated performance overhead compared to Fernet-based authentication.

The developed library enables the following functionalities:

- Issuing a user-tied RAF token given a Fernet token or another user-tied RAF token.
- Validating a user-tied RAF token using the Fernet key and extracting the sequence of embedded commands.

We also modified Keystone to accept and validate user-tied RAF tokens. The modifications are limited to the token issuance and validation components, where we extend the existing Fernet-based workflow to support RAF token generation and recursive validation. Importantly, the core functionality of Keystone remains unchanged, and the default Fernet token mechanism is preserved, allowing RAF to operate alongside existing authentication processes.

In the rest of this section, we explain the details of our experiments.

5.1. Experiment Platform and Evaluation Setup

Using KVM, we set up a virtual machine (VM) with 4 vCPUs and 8 GB vRAM. The physical host was a DELL XPS 9530 laptop taking advantage of an Intel Core(TM) i7-4712HQ CPU @ 2.30 GHz with 16 GB memory and 250 GB SSD. The operating systems of the host and virtual machine were Ubuntu 18.04.3 LTS. We had installed OpenStack Stein on our VM using the DevStack script.

The comparison between RAF and Fernet tokens is conducted under the same experimental conditions to ensure fairness. Both mechanisms are evaluated within the same OpenStack environment, using identical hardware, software configuration, and workload patterns.

The evaluation focuses on execution latency, measured as the time required to complete representative API operations. Each experiment is repeated multiple times, and the reported results correspond to the average values to reduce the impact of transient system variations.

RAF and Fernet tokens are used in comparable configurations, ensuring that the observed differences reflect only the overhead introduced by the RAF mechanism. This setup allows for a fair comparison between the two approaches.

Finally, the experiments are reproducible, as they rely on a standard OpenStack deployment with well-defined modifications and a deterministic evaluation procedure.

5.2. RAFT Library

Snippet Listing A4 represents the user-tied RAF library. It contains two classes:

- KeystoneRaft: This class encapsulates several methods to validate RAF tokens. Some of its important methods are:
  - isRAFT: Gets a token and returns true if the token was an RAF token.
  - CheckExpirationTime: Gets a token and checks its expiration time. If the token is expired, then it raises an exception error.
  - GetKey: This method is used internally to find the key of the token.
  - ValidateRAFT: Gets an RAF token and checks if it is valid or not. If the token is valid, this function returns all the commands inside the token. Also, it sets the fernet_token property of the class.
- ClientRaft: This class allows clients and other modules to create a new RAF token for their requests. The main methods of this class are:
  - __init__: This is the constructor of the class and allows us to define the parent token of the RAF token, which we want to create.
  - SetParentToken: Allows setting or changing the parent token.
  - SetCommand: Is used to set the command that will be a part of the token.
  - Finalize: Generates and returns an RAF token based on the parent token and the command that had been set.

5.3. The Required Changes

If you install OpenStack with its default settings, the source code of Keystone will be at */opt/stack/keystone* folder. Let's call this folder the *base folder*. Then, our library must be added to the following folder:

```
base folder/keystone/token
```

The source code of the Fernet token must be at:

```
base folder/keystone/token/providers/fernet
```

In the *fernet* folder, there is a file *token_formmater.py*, which contains the entry point for the Fernet library. We added the following code at the library importing section of the file (at the beginning of the file):

```
import RAFT
```

Then, we modified the *validate_token* function of *TokenFormatter* class, as shown in snippet Listing A5. When this function is called, we first check the input token. If it is an RAF token, then we process it using an instance of *KeystoneRAFT* class. Upon successful validation, we extract the original Fernet token and assign it to the token. In this way, the rest of the function checks the privileges of the root Fernet token. At the end of the function, if the input token was an RAF token, we adjust the expiration time of the token according to the shortest expiration time in the RAF token.

After applying the changes to the source code, the Keystone service must be restarted. Do not restart the virtual machine. If you do so, you need to uninstall OpenStack and try to install it again. In order to restart the Keystone service, execute the following command in the Linux terminal.

```
systemctl restart devstack@keystone.service
```

5.4. Single Token Generation and Verification Overhead

Table 3 shows the average time (in milliseconds) needed for creation and verification of a token, deviation, deviation above, and deviation below the average. If we use the Fernet token mechanism, the creation and validation time is always near the average in row 1. In the RAFT mechanism, whenever a user wants to issue a new command, they need to create a new RAF token from a Fernet token. By default, each Fernet token is valid for an hour. Hence, when the user grants a Fernet token, they can use it to create new RAF tokens locally for one hour. In the statistics about the RAF, we also include the time of granting a Fernet token.

Table 3. Execution time in milliseconds for generating and validating of a token based on 100 sample.

#	Token Type	Average	Dev.	Dev. to Above	Dev. to Bellow
1	Fernet	76.54	9.39	8	0
2	RAF, 1 ¶	79.24	13.38	4	2
3	RAF, 2	56.79	3.02	12	11
4	RAF, 10	40.83	1.57	9	6
5	RAF, 50	37.88	0.81	13	15
6	RAF, 100	37.21	0.64	13	13

¶: The number indicates the number of tokens issued per hour.

As mentioned earlier, one idea for mitigating the bearer token problem is to shorten the lifetime of Fernet tokens in a way that there is no time to use them twice. As our experiment shows, if the user issues more than one command in an hour (rows 3 to 6), the RAF mechanism works 30 to 50 percent better than the idea of shortening the lifetime of Fernet tokens.

Table 4 shows that the length of the command that we add to an RAF token does not noticeably affect the average time (in milliseconds) of creation and verification of a token. For this experiment, we create RAF tokens for hypothetical commands with 1, 100, 200, and 1000 character lengths.

Table 4. The effect of the length of the command (in characters) that is added to an RAF token in generation and verification time in milliseconds based on 100 samples.

#	Command Length	Average	Dev.	Dev. to Above	Dev. to Bellow
1	0	40.11	5.89	8	1
2	100	41.01	9.7	6	0
3	200	39.34	3.94	14	7
4	1000	40.78	5.02	12	8

5.5. RAF Processing Overhead Analysis

The RAF solution puts a negligible processing overhead on the Fernet processing time. Since many processes were running on the OpenStack server, this negligible overhead was not measurable using OpenStack. Hence, we calculated the overhead locally. In our platform Section 5.1, issuing 100 RAF tokens (excluding granting a Fernet token from Keystone) took 8.1 milliseconds, and verification of just the RAF tokens (excluding the verification of the base Fernet) took 13.2 milliseconds. These numbers show the extra processing time needed for an RAF token compared to a Fernet token.

Table 5 shows the summary of the first set of our experiments (see the source code in Snippet Listing A6) that focuses on the creation and validation time of user-tied RAF and Fernet token. Note that The process of issuing and validating a fully-tied RAF token with one command and a user-tied RAF token with one command is exactly the same. The only difference between issuing and validating a fully-tied RAF and a user-tied RAF token for more than one command is that they use different keys for the HMAC of the second and more commands. Hence, we do not expect any considerable change in the result if we repeat them for fully-tied RAF. The first set includes the following seven experiments:

- Row #1 shows the execution time for getting 100 Fernet tokens from Keystone.
- Row #2 represents the execution time for getting a Fernet token and generating 100 RAF tokens using the Fernet token.
- Row #3 exposes the execution time for getting and validating 100 Fernet tokens.
- Rows #4 to #7 demonstrate the execution time of getting a Fernet token and issuing 100 RAF tokens with hypothetical commands with 0, 30, 60, and 200 lengths (in characters).

To get more reliable results, we repeat each experiment several times, but because of space limitations, we only represent the results of the experiments.

As mentioned earlier, one idea for mitigating the bearer token problem is to shorten the lifetime of Fernet tokens in a way that there is no time to use them twice. From rows #1 and #2, we can conclude that for token generation, if a user issues more than one command in a specific period, the RAF solution is faster than the short-life Fernet token idea. However, the creation time is only one side of the problem. Each token needs to be validated by Keystone. If we include the validation time (rows #3 to #7), the RAF solution still works at most 50 percent better if each user issues several commands.

The length of the command that we add to an RAF token has unnoticeable overhead, as understood from rows #4 to #7. In fact, we repeated experiments #4 to #7 much more than five times, and each time we got very similar results.

5.6. RAF and Fernet in Action

In the second set of experiments, we examined the effect of using user-tied RAF instead of Fernet tokens in the execution time of four different commands as follows:

Create volume is a moderate workload command that creates a volume, i.e., a permanent memory like a physical disk. Snippet Listing A2 shows the sample command that we used

for creating a volume. We did not load any image into the volume. If we want to load an image into the volume, it can be considered a high-load task.

Table 5. Execution time in seconds for generating and validating 100 RAF tokens and Fernet tokens. For creating RAF tokens, we only interact with Keystone one time and then create 100 RAF tokens locally.

#	Experiment Title	First	Second	Third	Forth	Fifth	Avg
1	Fernet Generation	4.3519	4.3519	4.3593	4.4529	4.4897	4.4011
2	RAFT Generation	0.0489	0.0519	0.0510	0.0505	0.0481	0.0501
3	Fernet Generation and validation	8.9275	8.8489	8.8471	8.5349	8.8222	8.7961
4	RAF token Generation and validation; 0 L	4.6791	4.5629	4.6725	4.6074	4.6696	4.6383
5	RAF token Generation and validation; 30 L	4.6528	4.6620	4.5986	4.6332	4.6296	4.6353
6	RAF token Generation and validation; 60 L	4.6668	4.5719	4.6438	4.7847	4.5213	4.6377
7	RAF token Generation and validation; 200 L	4.6302	4.6704	4.7203	4.5572	4.6420	4.6440

Create VM (server) is a typical command in OpenStack that consumes significant resources. Snippet Listing A1 shows the sample command that we used for creating a VM. In this sample, we used *cirros*, which is an ultra-lightweight operating system. The *cirros* operating system (OS) includes a few core functionalities of a Linux OS and can only be used for tests.
Image list is a light load command that does not need any parameters.
Project list is another light load command that returns the list of projects that are available for the owner of the token. This command also does not need any parameters.

Table 6 represents a comparison between the execution time (in seconds) of the four different commands using RAF and Fernet tokens. We put the result of five executions for each command in the table. Based on the result, the overhead of using RAF instead of Fernet in typical create-server and create-volume commands is less than 1 percent. Here, we used Cirros OS to create a server that is a very small, non-practical operating system. In a real deployment, usually creating a VM using an operational OS takes several seconds, and the RAF overhead is not noticeable in such cases. In general, typical cloud operations like creating servers, migrating servers, and taking backups of servers are very time-consuming compared to token verification. Hence, we expect that this overhead will be much less than 1 percent for a real deployment.

Snippet Listing A7 shows the source code that we developed for the second set of experiments.

Table 6. Comparison between the execution time of four different commands using RAFT and Fernet tokens.

Experiment Title	First	Second	Third	Forth	Fifth	Avg	Ratio
Creating a volume (Fernet)	0.3243	0.3681	0.4546	0.4396	0.4246	0.4022	1.0078
Creating a volume (RAF)	0.3583	0.4021	0.4140	0.4313	0.4211	0.4054	
Creating a VM (Fernet)	0.5208	0.5966	0.8908	2.3995	0.6762	1.0168	1.0030
Creating a VM (RAF)	0.7032	0.7740	0.8938	2.0452	0.6831	1.0198	
Getting image list (Fernet)	0.0584	0.0571	0.0569	0.0605	0.0526	0.0571	1.0280
Getting image list (RAF)	0.0548	0.0598	0.0616	0.0553	0.0620	0.0595	
Getting project list (Fernet)	0.0293	0.0332	0.0274	0.0296	0.0294	0.0298	1.0326
Getting project list (RAF)	0.0303	0.0323	0.0296	0.0311	0.0305	0.0307	

5.7. Summary of Performance Results

Evaluating modifications to a production OpenStack deployment under real workloads is challenging. Therefore, we conduct controlled experiments to measure the overhead introduced by RAF compared to Fernet tokens.

Our results show that RAF introduces only a marginal overhead in terms of execution latency compared to Fernet. This is expected, as both RAF and Fernet rely on symmetric cryptographic primitives, resulting in similar computational costs.

In contrast, token mechanisms based on public-key cryptography, such as JWT, typically incur higher computational overhead due to more expensive cryptographic operations. While we do not provide an empirical comparison with JWT, it is reasonable to expect RAF to achieve better performance than such approaches, given its reliance on lightweight symmetric operations.

Although our implementation of RAF is based on OpenStack and involves modifications to Keystone, the proposed mechanism is not specific to this platform. RAF is designed as a general token-based framework that can be integrated into other systems that rely on bearer tokens for authentication and authorization.

The core components of RAF, including recursive token derivation, blacklist enforcement, and policy-based validation, are independent of OpenStack and can be adapted to other architectures such as microservices-based systems, RESTful APIs, or cloud platforms with centralized or decentralized identity management.

While implementation details may vary depending on the target platform, the underlying design principles of RAF remain applicable, making it a flexible approach for enhancing token security in a wide range of systems.

6. Conclusions and Discussion

Modular system design requires robust authentication and authorization mechanisms. While token-based authentication improves security compared to traditional credential-based approaches, it introduces the well-known "bearer token" problem, where possession of a token is sufficient to gain access. This creates a fundamental trade-off between usability, performance, and security in modular systems.

To address this challenge, this paper introduces RAF, a novel token-based authentication mechanism designed to satisfy the Modularity Security Requirement. RAF limits the impact of token leakage by binding tokens to specific commands and enforcing single-use semantics. The proposed design combines cryptographic guarantees with system-level components, including a policy enforcer and a blacklist, to prevent token misuse, replay attacks, and unauthorized command execution.

We present two variants of RAF: user-tied RAF and fully-tied RAF. User-tied RAF offers improved usability and scalability by avoiding service-specific key management, while fully-tied RAF provides stronger security guarantees by binding tokens to individual services. These two variants highlight an inherent trade-off between ease of deployment and security strength.

The security of the RAF was formally analyzed using game-based definitions, showing that forging valid tokens without access to ancestor tokens is computationally infeasible under standard cryptographic assumptions. Additionally, we implemented a proof of concept on OpenStack and evaluated RAF in comparison with Fernet-based authentication. The results demonstrate that RAF introduces only a marginal performance overhead while significantly improving security guarantees.

6.1. Generalization to Other Platforms

Although our implementation is based on OpenStack and requires modifications to Keystone, the RAF mechanism is not specific to this platform. RAF is a general token-based framework that can be integrated into other systems that rely on bearer tokens, including microservices architectures, RESTful APIs, and cloud platforms with centralized or distributed identity management. The core design principles of RAF, including recursive token derivation and command binding, remain applicable across different environments.

6.2. Limitations and Deployment Considerations

Despite its advantages, the RAF introduces several practical challenges. First, the blacklist component requires synchronization in distributed systems, which may introduce additional overhead under high request volumes. Second, the propagation of tokens across multiple services increases system complexity and may impact performance in deeply nested request chains. Third, fully-tied RAF requires the management and distribution of service-specific secret keys, which may affect scalability in large or dynamic environments.

In addition, integrating RAF into existing systems may require modifications to authentication workflows and identity management components, which could impact backward compatibility and deployment effort.

6.3. Future Work

Future work includes exploring scalable implementations of the blacklist component in distributed environments, investigating stateless alternatives to reduce storage overhead, and optimizing the RAF for large-scale deployments. Another promising direction is the integration of the RAF with existing authorization frameworks to further enhance system security while maintaining efficiency.

Author Contributions: Conceptualization, R.R.; methodology, R.R. and M.v.D.; software, R.R.; validation, R.R.; formal analysis, R.R. and M.v.D.; investigation, R.R.; writing—original draft preparation, R.R.; writing—review and editing, R.R. and M.v.D.; supervision, M.v.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data are available from the corresponding author upon reasonable request.

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

Appendix A. RAF Security Guarantees

Appendix A.1. Proof of Lemma 1

Based on Game 1, we define two other games, which we will need for the proof of Lemma 1:

- Consider $ForgeUTT^{i}_{\mathcal{A},\Pi}$ to be the notion for forging a token with exactly i commands. For this, we can define a game that is exactly as Game 1 with one extra condition in Step 3, which is *"m contains exactly i commands"*
- Consider $ForgeUTT^{\leq i}_{\mathcal{A},\Pi}$ to be the notion for forging a token with at most i commands. For this, we can define a game that is exactly as Game 1 with one extra condition in Step 3, which is *"m contains at most i commands"*

Proof. The intuition behind the proof of this lemma is that for a PPT adversary, forging a user-tied RAF without observing any of its ancestors involves distinguishing the HMAC from a random function. We simplify the descriptions of Algorithms 1, 2 and 4 as follows:

- For the key k and input payload p and the command c
  $UserIssue(k, p, c) = HMAC(HMAC(k, p), p \parallel c)$

- For the key k, input token $r = (m, t)$, and the command c

$$ServiceIssue(k, r = (m, t), c) = \begin{cases} HMAC(t, m \parallel c) & \text{if } Ver(k, r) \text{ is valid} \\ nothing & \text{if } Ver(k, r) \text{ is invalid} \end{cases}$$

- For the key k, and the input token $r = (m, t)$, first, $Ver(k, r)$ decomposes m and gets $\{p, c_1, \cdots, c_n\}$, then calculates

$$\begin{aligned} h_0 &= HMAC(k, p) \\ h_1 &= HMAC(h_0, p \parallel c_1) \\ &\vdots \\ h_n &= HMAC(h_{n-1}, p \parallel c_1 \parallel \cdots \parallel c_n) \end{aligned}$$

  Then, if $h_n = t$, it returns *valid*; otherwise, it returns *invalid*.

  Let $\mathcal{A}$ be a probabilistic polynomial-time adversary, and let $\varepsilon(.)$ be a function such that:

$$\Pr[ForgeUTT_{\mathcal{A},RAF}(\lambda) = 1] = \varepsilon(\lambda) \tag{A1}$$

and let Φ represent the set of all tokens that $\mathcal{A}$ has queried. Now, we define a new scheme $RAF' = (UserIssue', ServiceIssue', Ver')$, which is similar to RAF except that a random function $f(.)$ is replaced with the first HMACs in $UserIssue$ and Ver algorithms. In other words, *HMAC(k,p) is replaced by f(p)*.

We show that this implies the existence of a polynomial-time algorithm that can distinguish the HMAC from a random one with advantage $O(\varepsilon)$. This will then imply that ε must be negligible, as required.

We define the notation $RAF(k, p, c_1, \ldots, c_i)$, respectively $RAF'(k, p, c_1, \ldots, c_i)$ to stand for an RAF/RAF′ token with i commands and define RAF and RAF' as follows:

$$RAF(k, p, c_1, \ldots, c_i) = \begin{cases} HMAC(RAF(k, p, c_1, \ldots, c_{i-1}), p \parallel c_1 \parallel \ldots \parallel c_i) = h_i & i > 0 \\ HMAC(k, p) = h_0 & i = 0 \end{cases}$$

and

$$RAF'(k, p, c_1, \ldots, c_i) = \begin{cases} HMAC(RAF'(k, p, c_1, \ldots, c_{i-1}), p \parallel c_1 \parallel \ldots \parallel c_i) = h_i & i > 0 \\ f(p) = h_0 & i = 0 \end{cases}$$

From $\mathcal{A}$ we will now construct a polynomial-time distinguisher $\mathcal{D}$ that is given an oracle $\mathcal{O}$ that is either of $HMAC(k, .)$ or a random function $f(.)$. The construction $\mathcal{D}$ works as follows:

- Upon input λ, adversary $\mathcal{D}$ passes λ to $\mathcal{A}$.
- When $\mathcal{A}$ queries its oracle $UserIssue(k, ., .)$ with a message $m' = (p', c')$, $\mathcal{D}$ queries $\mathcal{O}$ with p' and receives the result h'. Next, it calculates the value $t' = HMAC(h', m')$ and hands it to $\mathcal{A}$ and continues.
- When $\mathcal{A}$ queries its oracle $Ver(k, .)$ with a message $r'' = (m'', t'')$, $\mathcal{D}$ runs Algorithm A1. Algorithm A1 is exactly the same as Algorithm 4 except that instead of using HMAC to compute h_0, it uses the oracle $\mathcal{O}$. Upon receiving b'' from Algorithm A1, $\mathcal{D}$ hands it to $\mathcal{A}$.

- When $\mathcal{A}$ queries its oracle $ServiceIssue(k,.,.)$ with a message $(r' = (m', t'), c')$, $\mathcal{D}$ runs Algorithm A1 with r' and gets the response b'. If $b' = 0$, $\mathcal{D}$ does nothing. However, if $b' = 1$, $\mathcal{D}$ hands $HMAC(t', m' \parallel c')$ to $\mathcal{A}$ and continues.

At the end, when $\mathcal{A}$ outputs $r = (m, t)$, $\mathcal{D}$ runs Algorithm A1 and receives b. If $b = 1$ and r and any of its ancestors are not in Φ, then $\mathcal{D}$ outputs 1. Otherwise it outputs 0.

Algorithm A1 Pseudocode of verifying

```
Function Verify(r)
    (p, c_0, ··· , c_n, t) ← Unpack(r)
    i = 0
    h_0 = O(p)
    while i ≤ n do
        h_i = HMAC(h_{i−1}, p ‖ c_0 ‖ ··· ‖ c_i)
        i = i + 1
    end
    if h_n = t then
        return 1;
    else
        return 0;
    end
End Function
```

Since $\mathcal{A}$ runs in polynomial time, D also runs in polynomial time. From the construction of $\mathcal{D}$, it is clear that depending on $\mathcal{O}$, $\mathcal{A}$ either plays $ForgeUTT_{\mathcal{A},RAF}$ or $ForgeUTT_{\mathcal{A},RAF'}$, and we have

$$\Pr[ForgeUTT_{\mathcal{A},RAF}(\lambda) = 1] = Pr[D^{HMAC(k,.)}(\lambda) = 1] \tag{A2}$$

$$\Pr[ForgeUTT_{\mathcal{A},RAF'}(\lambda) = 1] = Pr[D^{f(.)}(\lambda) = 1] \tag{A3}$$

From (3), (A2) and (A3)

$$|\Pr[ForgeUTT_{\mathcal{A},RAF}(\lambda) = 1] - \Pr[ForgeUTT_{\mathcal{A},RAF'}(\lambda) = 1]| \leq \alpha(\lambda) \tag{A4}$$

The analysis presented so far can be repeated in exactly the same way for the game $ForgeUTT^i$. This gives

$$|\Pr[ForgeUTT^i_{\mathcal{A},RAF}(\lambda) = 1] - \Pr[ForgeUTT^i_{\mathcal{A},RAF'}(\lambda) = 1]| \leq \alpha(\lambda) \tag{A5}$$

Let $\beta(.)$ be a function so that $\Pr[ForgeUTT^0_{\mathcal{A},RAF}(\lambda) = 1] = \beta(\lambda)$. Since for $i = 0$, $RAF(k, p) = HMAC(k, p)$ and $RAF'(k, p) = f(p)$ this indicates that

$$\Pr[ForgeUTT^0_{\mathcal{A},RAF}(\lambda) = 1] = \Pr[D^{HMAC(.)} = 1] = \beta(\lambda)$$

and

$$\Pr[ForgeUTT^0_{\mathcal{A},RAF'}(\lambda) = 1] = \Pr[D^{f(.)} = 1] = \frac{1}{2^\lambda - C}$$

where $C = poly(\lambda)$ is the maximum number of queries distinguisher D applies. Therefore,

$$\alpha(\lambda) \geq |\Pr[D^{HMAC(.)} = 1] - \Pr[D^{f(.)} = 1]| \geq \beta(\lambda) - \frac{1}{2^\lambda - C}.$$

By the assumption that HMAC is secure (i.e., $\alpha(\lambda)$ is negligible) and because $C = poly(\lambda)$, it follows that $\beta(\lambda)$ is also negligible in λ. To be more precise

$$\Pr[ForgeUTT^{0}_{\mathcal{A},RAF}(\lambda) = 1] = \beta(\lambda) \leq \alpha(\lambda) + \frac{1}{2^{\lambda} - C} \tag{A6}$$

Consider some adversary $\mathcal{A}_{i+1}$ in $ForgeUTT^{i+1}_{\mathcal{A}_{i+1},RAF'}(\lambda)$. This adversary produces a token r with a chain of ancestors h_i, where h_i denotes the intermediate HMAC value after processing the first i commands, and $h_0 = f(p)$.

Let $G(\cdot)$ denote a recursive function defined as follows:

$$\begin{aligned} G([p, c_1], f(p)) &= HMAC(f(p), p \parallel c_1), \text{ and} \\ G([p, c_1, \ldots, c_{i+1}], f(p)) &= HMAC(G([p, c_1, ..., c_i], f(p)), p \parallel c_1 \parallel \cdots \parallel c_{i+1}). \end{aligned}$$

Assume $t = G([p, c_1, \ldots, c_{i+1}], f(p))$. So, we can write $r = ([p, c_1, ..., c_{i+1}], t)$.

In order to help creating a token r that verifies correctly, only information about $f(p)$ can help. Since f is a random function,$f(q)$ for $q \neq p$ does not give any information about $f(p)$, which also means that $G(., f(p))$ and $G(., f(q))$ are statistically independent. Thus, the information that the distinguisher $\mathcal{D}$ provides to the adversary $\mathcal{A}$ gives no useful information unless oracle $\mathcal{O}$ uses $f(p)$ for its computations. We can therefore neglect all the oracle accesses that do not relate to computations with $f(p)$. This gives a new adversary $\mathcal{A}_i$ such that

$$\Pr[ForgeUTT^{i+1}_{\mathcal{A}_{i+1},RAF'}(\lambda) = 1] = \Pr[ForgeUTT^{i+1}_{\mathcal{A}_i,RAF'}(\lambda) = 1] \tag{A7}$$

By repeatedly applying the recursive definition of G, we obtain for $i \geq 0$,

$$\Pr[ForgeUTT^{i+1}_{\mathcal{A}_i,RAF'}(\lambda) = 1] = \Pr[ForgeUTT^{i}_{\mathcal{A}_i,RAF}(\lambda) = 1], \tag{A8}$$

which follows from the following induction argument: Let $k = f(p)$, $p' = p \parallel c_1$, and $c'_j = c_{j+1}$ for $j \in \{1, 2, \ldots, i\}$. As a base case notice that

$$\begin{aligned} & RAF'(k, p, c_1) \\ = \; & HMAC(f(p), p \parallel c_1) \\ = \; & HMAC(k, p') \\ = \; & RAF(k, p'). \end{aligned}$$

Let $i \geq 1$. As an induction hypothesis assume

$$RAF'(k, p, c_1, \ldots, c_i) = RAF(k, p', c'_1, \ldots, c'_{i-1}).$$

In exactly the same way as before we can prove the induction step

$$\begin{aligned} & RAF'(k, p, c_1, \ldots, c_{i+1}) \\ = \; & HMAC(RAF'(k, p, c_1, \ldots, c_i), p \parallel c_1 \parallel c_2 \parallel \ldots \parallel c_{i+1}) \\ = \; & HMAC(RAF(k, p', c'_1, \ldots, c'_{i-1}), p' \parallel c'_1 \parallel \ldots \parallel c_i) \\ = \; & RAF(k, p', c'_1, \ldots, c'_i). \end{aligned}$$

By using induction in i, we conclude that the induction hypothesis holds for all $i \geq 1$, and (A8) follows.

By defining p_i and p'_i as

$$p_i = \Pr[ForgeUTT^i_{\mathcal{A}_i,RAF}(\lambda) = 1] \text{ and } p'_i = \Pr[ForgeUTT^i_{\mathcal{A}_i,RAF'}(\lambda) = 1] \quad \text{(A9)}$$

we conclude from Equations (A5) and (A9)

$$|p_i - p'_i| \leq \alpha(\lambda)$$

which in turn proves

$$p_i \leq \alpha(\lambda) + p'_i.$$

From Equations (A6) and (A9) we have

$$p_0 = \beta(\lambda)$$

and from Equations (A8) and (A9) we have

$$p'_{i+1} = p_i.$$

These properties of p_i and p'_i can be combined to prove

$$p_n \leq \alpha(\lambda) + p'_n = \alpha(\lambda) + p_{n-1} \leq \ldots \leq n \cdot \alpha(\lambda) + p_0 \leq n \cdot \alpha(\lambda) + \beta(\lambda).$$

We can now conclude that

$$\begin{aligned} \Pr[ForgeUTT^{\leq n}_{\mathcal{A}_i,RAF}(\lambda) = 1] &= \max_{i \leq n} \Pr[ForgeUTT^i_{\mathcal{A}_i,RAF}(\lambda) = 1] \\ &\leq \Pr[ForgeUTT^n_{\mathcal{A}_i,RAF}(\lambda) = 1] \\ &\leq n \cdot \alpha(\lambda) + \beta(\lambda). \end{aligned} \quad \text{(A10)}$$

Since the adversary can at most call the oracle $poly(\lambda)$ times, $n = poly(\lambda)$. In fact, because of the policy enforcer, n cannot be more than the number of modules in a system. Since $\alpha(\lambda)$ and $\beta(\lambda)$ are negligible functions in λ and $n = poly(\lambda)$, (A10) proves the lemma. □

Appendix A.2. Proof of Lemma 2

Based on Game 2, we define two other games which we will need for the proof of Lemma 2:

- Consider $ForgeFTT^i_{\mathcal{A},\Pi}$ to be the notion for forging a token with exactly i commands. For this, we can define a game that is exactly as Game 2 with one extra condition in Step 3, which is *"m contains exactly i commands"*
- Consider $ForgeFTT^{\leq i}_{\mathcal{A},\Pi}$ to be the notion for forging a token with at most i commands. For this, we can define a game that is exactly as Game 2 with one extra condition in Step 3, which is *"m contains at most i commands"*

Proof. The intuition behind the proof of this lemma is that for a PPT adversary, forging a fully-tied RAF (i.e., $\widehat{RAF}$) without observing the input message before involves distinguishing the HMAC from a random function.

Lets introduce the notation $\widehat{RAF}(K, p, c_1, \ldots, c_i)$ (where $K = (k_0, k_1, \ldots, k_{n-1})$ and n is the number of services) to stand for a fully-tied RAF token with i commands and define $\widehat{RAF}$ as follows:

$$\widehat{RAF}(K,p,c_1,\ldots,c_i)=\begin{cases} HMAC(HMAC(k_0,p),p \parallel c_1)=h_1 & i=1 \\ HMAC(k_{i-1},p \parallel c_1 \parallel \ldots \parallel c_i \parallel \widehat{RAF}(K,p,c_1,\ldots,c_{i-1}))=h_i & n \geq i>1 \end{cases}$$

Let us also simplify the descriptions of Algorithms 1, 3 and 5 as follows:

- For the key k_0 and input payload p and the command c:

$$UserIssue(k_0,p,c)=HMAC(HMAC(k_0,p),p \parallel c)=h_1=\widehat{RAF}(K,p,c_1) \quad \text{(A11)}$$

- For the key k_{i-1} ($i \in \{2,\ldots,n\}$ where n is the number of services), input token $r=(m,t)=(p \parallel c_1 \parallel \ldots \parallel c_{i-1},h_{i-2})$, and the command c_i:

$$\begin{aligned} ServiceIssue(i-1,K,r=(m,t),c_i) &= HMAC(k_{i-1},m \parallel c_i \parallel t) \\ &= HMAC(k_{i-1},p \parallel c_1 \parallel \cdots \parallel c_{i-1} \parallel h_{i-1}) \\ &= h_i \\ &= \widehat{RAF}(K,p,c_1,\ldots,c_i) \end{aligned} \quad \text{(A12)}$$

- For the key set $K=(k_0,k_1,\ldots,k_{n-1})$ where n is the number of services, and the input token $r=(m,t)$, first, $Ver(K,r)$ decomposes m and gets $\{p,c_1,\cdots,c_i\}$, then calculates

$$\begin{aligned} & h_1=HMAC(HMAC(k_0,p),p \parallel c_1) \\ & \textit{Find corresponding key } k_1 \textit{ for } c_2 \\ & h_2=HMAC(k_1,p \parallel c_1 \parallel c_2 \parallel h_1) \\ & \textit{Find corresponding key } k_2 \textit{ for } c_3 \\ & h_3=HMAC(k_2,p \parallel c_1 \parallel c_2 \parallel c_3 \parallel h_2) \\ & \vdots \\ & \textit{Find corresponding key } k_{i-1} \textit{ for } c_i \\ & \quad \textit{Extract } k_n \textit{ from } c_n \\ & \quad h_i=HMAC(k_{i-1},p \parallel c_1 \parallel \cdots \parallel c_i \parallel h_{i-1}) \end{aligned}$$

  Then, if $h_i=t$, returns *valid*; otherwise, returns *invalid*.

  Let $\mathcal{A}$ be a probabilistic polynomial-time adversary and let $\varepsilon(.)$ be a function such that

$$\Pr[ForgeFTT_{\mathcal{A},\widehat{RAF}}(\lambda)=1]=\varepsilon(\lambda) \quad \text{(A13)}$$

and let Φ represent the set of all tokens that $\mathcal{A}$ has queried. We want to show that $\varepsilon(\lambda)$ is negligible.

According to the Game 2, the adversary $\mathcal{A}$ wins the game $ForgeFTT_{\mathcal{A},\widehat{RAF}}$ if they can forge a token that is either an output of *UserIssue* or *ServiceIssue* for the case which the k_i is not leaked.

In other words,

$$\begin{aligned} \Pr[ForgeFTT_{\mathcal{A},\widehat{RAF}}(\lambda)=1] &= \Pr[ForgeFTT^{1\leq i\leq n}_{\mathcal{A},\widehat{RAF}}(\lambda)=1] \\ &= \max_{1\leq i\leq n} \Pr[ForgingFTT^{i}_{\mathcal{A},\widehat{RAF}}(\lambda)=1] \end{aligned} \quad \text{(A14)}$$

where $i \notin I$.

A closer look at the structure of user-tied RAF token (RAF) and fully-tied RAF token ($\widehat{RAF}$) shows that, for $i = 1$,

$$\widehat{RAF}(K, p, c_1) = HMAC(HMAC(k_0, p), p \parallel c_1) = RAF(k_0, p, c_1) \tag{A15}$$

and for $1 < i \leq n$, if we consider $k' = k_{i-1}$ and $p' = p \parallel c_1 \parallel c_2 \parallel \ldots \parallel c_i \parallel h_{i-1}$ then,

$$\begin{aligned} \widehat{RAF}(K, p, c_1, \ldots, c_i) &= HMAC(k_{i-1}, p \parallel c_1 \parallel \ldots \parallel c_i \parallel h_{i-1}) \\ &= HMAC(k', p') \\ &= RAF(k', p') \end{aligned} \tag{A16}$$

From Lemma 1 and Equation (A15) we have

$$\Pr[ForgingFTT^1_{\mathcal{A},\widehat{RAF}}(\lambda) = 1] = \Pr[ForgingUTT^1_{\mathcal{A},RAF}(\lambda) = 1] \leq \alpha(\lambda) + \beta(\lambda) \tag{A17}$$

where $\alpha(\lambda)$ and $\beta(\lambda)$ are negligible functions in λ (for more details see Equation (A6)).

From Lemma 1 and Equation (A16) we have (for $1 < i \leq n$):

$$\Pr[ForgingFTT^i_{\mathcal{A},\widehat{RAF}}(\lambda) = 1] = \Pr[ForgingUTT^0_{\mathcal{A},RAF}(\lambda) = 1] \leq \beta(\lambda) \tag{A18}$$

From Equations (A13), (A14), (A17) and (A18) we have

$$\varepsilon(\lambda) = \Pr[ForgeFTT_{\mathcal{A},\widehat{RAF}}(\lambda) = 1] \leq \alpha(\lambda) + \beta(\lambda) \tag{A19}$$

Since $\alpha(\lambda)$ and $\beta(\lambda)$ are negligible functions in λ, (A19) proves the lemma. □

Appendix B. Source Code

Listing A1. The sample command for creating a virtual machine that we used in our experiment.

```
def get_create_server_sample():
    cmd = "compute/v2.1/servers"
    data ={
        "server": {
            "name": "vm_name",
            "imageRef": "ce0afaaa-e236-47c6-95e8-47c7694eb74c",
            "flavorRef": "1",
            "max_count": 1,
            "min_count": 1,
            "networks": [{"uuid":
             ↪  "5eeb14b4-47a9-44aa-bade-b225b7713a6b"}]
        }
    }

    cmd += " " + str(data)
    return cmd
```

Listing A2. The sample command for creating a volume that we used in our experiment.

```
def get_create_volume_sample():
    cmd = "volume/v2/08b72d6e4f2b465d96e9e0db2f10d232/volumes"
    data = {
```

```
        "volume": {
            "status": "creating",
            "name": "vol_name",
            "imageRef": "ce0afaaa-e236-47c6-95e8-47c7694eb74c",
            "attach_status": "detached",
            "volume_type": "lvmdriver-1",
            "size": 1
        }
    }
    cmd += " " + str(data)
    return cmd
```

Listing A3. The python code for extracting the payload of a project scoped token given a Fernet key and token.

```
# this code is adapted from the source code of Keystone Stein
from cryptography.fernet import Fernet
import msgpack
import uuid

def restore_padding(token):
    mod_returned = len(token) % 4
    if mod_returned:
        missing_padding = 4 - mod_returned
        token += '=' * missing_padding
    return token

key = "Qh4ZzunoX36Ri0TKVa3bXqzTQKzwqT3G4JfmGw1ZNtU="
f = Fernet(key)
token="gAAAAABdpxhmvMe_byl3qKlJ0KVXizdSyL_38Idxam2ap7O1T9_xzX9eVJ6WCozRKlXjH6oZlDuOyS0nI_57u0G0ceOt7coU
↪  tDPPI1TipydgxMekVNtbhdHuR8A9BMvY1pPAVkGV_23Hd_Ste0eiTXP7m_7W77Vj3X2qGkjkeuinyGZsTclYZOc"
print("Token Length = %d" % len(token))
token = restore_padding(token)
serialized_payload = f.decrypt(token.encode('utf-8'))
print("Payload Length = %d" %len(serialized_payload))

versioned_payload = msgpack.unpackb(serialized_payload)
version, payload = versioned_payload[0], versioned_payload[1:]
print("version = %d" % version) # prints 2 which means Project Scoped Token

(is_stored_as_bytes, user_id) = payload[0]
if is_stored_as_bytes:
    user_id = uuid.UUID(bytes=user_id)
print("User Id = %s" % user_id)

print("Method = %d" % payload[1]) # prints 2 which means "Password"

(is_stored_as_bytes, project_id)=payload[2]
if is_stored_as_bytes:
    project_id = uuid.UUID(bytes=project_id)

print("Project Id = %s" % project_id)

print("Expiration Time %s " % payload[3])
print("Audit Id = %s"% payload[4])
```

Listing A4. The source code of user-tied RAFT Library.

```
#importing the required library
import os
import time
import six

from cryptography.hazmat.backends import default_backend
import base64
from cryptography import fernet
from cryptography.hazmat.primitives.hmac import HMAC
```

```python
from cryptography.hazmat.primitives import hashes
from cryptography import utils
from struct import pack,unpack, unpack_from

#KeystoneRAFT is developed to be added to Keystone. It allows Keystone to verfy RAFT tokens
class KeystoneRAFT(object):
    expirationTime = 10
    @classmethod
    def restore_padding(cls, token):
        """Restore padding based on token~size.

        :param token: token to restore padding on
        :type token: six.text_type
        :returns: token with correct~padding

        """
        # Re-inflate the padding
        mod_returned = len(token) % 4
        if mod_returned:
            missing_padding = 4 - mod_returned
            token += '=' * missing_padding
        return~token

    #the function which is used for initialization of new object
    def __init__(self, backend=None):
        if backend is None:
            backend = default_backend()
        self._backend = backend

    def isRAFT(self, token):
        """
            Checks the first byte of the given token to find out if it is a RAFT or NOT.
            Returns True if the
        """
        if six.indexbytes(base64.urlsafe_b64decode(ClientRAFT.restore_padding(token)),0)== 0x91:
            return True
        else:
            return~False

    def CheckExpirationTime(self,token,pos):
        """
            Check the expiration time of a token adn rise an exception if the token is already expired
        """
        exp_time, = unpack_from(">Q",token,pos)
        if self.expirationTime > exp_time:
            self.expirationTime =exp_time

        if exp_time < time.time():
            raise Exception("the token has expired")
        return~exp_time

    def GetKey(self, token):
        """
            This function recursively extracts the parent messages and checks their expiration time
            ↪  until it gets to the root token.
            Then it starts to recalculate the hmac of every parent's messages.
        """
        v, = unpack_from(">B",token,0)
        if v==0x91 :
            #it must be a RAFT
            lenId, = unpack_from(">H",token,1)
            OriginalId, = unpack_from(">" + str(lenId)+"s", token,3)
            cmd = token[19+lenId:]
            self.CheckExpirationTime(token, 3+lenId)
            keys, cmds = self.GetKey(OriginalId)
            cmds.append(cmd)
            sign_key = keys[:16]
            h = HMAC(sign_key, hashes.SHA256(), self._backend)
            h.update(token)
            hmac=h.finalize()
            return hmac, cmds
        else:
            # when we get the root token which is supposed to be Fernet token
            sign_key = base64.urlsafe_b64decode(self.SourceKey)[-32:-16]
```

```
            h = HMAC(sign_key, hashes.SHA256(), self._backend)
            h.update(token)
            signature = h.finalize()
            self.fernetToken = base64.urlsafe_b64encode(token+signature)
            return signature,[]
        return~""

    def ValidateRAFT(self, token):
        """
            This function get a token and using GetKey function rebulds the HMAC of the message part of
            ↪  the token.
            If calculated HMAC fits with the token signature, then the token is valid; Otherwise the
            ↪  function raises an exception error
        """
        token = self.restore_padding(token)
        token = base64.urlsafe_b64decode(token)
        v, = unpack_from(">B",token,0)
        self.SourceKey = 'Qh4ZzunoX36Ri0TKVa3bXqzTQKzwqT3G4JfmGw1ZNtU='
#        self.SourceKey = fernet_utils.load_keys()
        if v==0x91 :
            lenId, = unpack_from(">H",token,1)
            OriginalId, =  unpack_from(">"+str(lenId)+"s",token,3)
            self.expirationTime = self.CheckExpirationTime(token, 3+lenId)
            cmd = token[19+lenId:-32]
            print(cmd)
            keys,cmds = self.GetKey(OriginalId)
            cmds.append(cmd)
            sign_key = keys[:16]
            h = HMAC(sign_key, hashes.SHA256(), self._backend)
            h.update(token[:-32])
            hmac=h.finalize()
            if hmac== token[-32:]:
                #print("It is valid")
                return True,cmds
            else:
                raise Exception("Error: Not a RAFT token (MAC)")
        else:
            raise Exception("Error: Not a RAFT token (Format)")
        return~""



#ClientRaft is used for creating a new RAFT from either a Fernet or other RAFT.
class ClientRAFT(object):
    _lifeTime = 1  #feault lifetime for a RAFT
#the function which is used for initialization of new object
    def __init__(self, parent_token=None, backend=None):
        if backend is None:
            backend = default_backend()
        self._backend = backend

        if parent_token is not None:
            self.SetParentToken(parent_token)

    @classmethod
    def restore_padding(cls, token):
        """Restore padding based on token~size.

        :param token: token to restore padding on
        :type token: six.text_type
        :returns: token with correct~padding

        """
        # Re-inflate the padding
        mod_returned = len(token) % 4
        if mod_returned:
            missing_padding = 4 - mod_returned
            token += '=' * missing_padding
        return~token

    def SetParentToken(self,parent_token):
        """
            Allows to define/redfine the parent token of this object
        """
```

```
        parent_token= self.restore_padding(parent_token)
        self._parentToken = base64.urlsafe_b64decode(parent_token)
        #the last 32 bytes of the parent token is the HMAC, which the first 16 bytes of it is the
        ↪  signing of current token
        self._signKey = self._parentToken[-32:-16]
        self._encryptionKey = self._parentToken[-16:]
        self._id = self._parentToken[:-32]

    def SetCommand(self, command):
        """
            Specifinying a command that is supposed to be added to the parent token
        """
        self._command = command


    def Finalize(self, life_time = None):
        """
        Packs all the items needed to be in the token,
            specifies the lifetime,
            calculates the HMAC of the packed message,
            and convert the result to a base64 string, which is the new RAFT token
        """
        if life_time is not None:
            self._lifeTime = life_time

        self._NewToken = pack(">BH" +
                              str(len(self._id))+"s",
                              0x91,
                              len(self._id),
                              self._id)
        self._command=pack(">Q8s"+str(len(self._command))+"s",
        ↪  int(time.time())+self._lifeTime,os.urandom(8),self._command)
        ↪  #bytearray(self._command,"utf8")
        self._NewToken+=self._command
        self.h = HMAC(self._signKey, hashes.SHA256(), self._backend)
        self.h.update(self._NewToken)
        self._NewToken += self.h.finalize()
        return base64.urlsafe_b64encode(self._NewToken).rstrip(b'=')
```

Listing A5. The validate_token function of the TokenFormatter class after applying the required changes.

```
    def validate_token(self, token):
# The following code is added to check if the token is RAFT, then process the RAFT using our library
        try:
            raftFlag = False
            kr = RAFT.KeystoneRAFT()
            if kr.isRAFT(token):
                raftFlag = True
                kr.ValidateRAFT(token)
                token = kr.fernetToken
         except Exception as ex:
            template = "An exception of type {0} occurred. Arguments:\n{1!r}"
            message = template.format(type(ex).__name__, ex.args)
#End of processing a~RAFT.

#The old code for processing Fernet Token
        serialized_payload = self.unpack(token)
        versioned_payload = msgpack.unpackb(serialized_payload)
        version, payload = versioned_payload[0], versioned_payload[1:]
        for payload_class in _PAYLOAD_CLASSES:
            if version == payload_class.version:
                (user_id, methods, system, project_id, domain_id,
                 expires_at, audit_ids, trust_id, federated_group_ids,
                 identity_provider_id, protocol_id, access_token_id,
                 app_cred_id) = payload_class.disassemble(payload)
                break
        else:
            raise exception.ValidationError(_(
                'This is not a recognized Fernet payload version: %s') %
                version)
```

```
        if isinstance(system, bytes):
            system = system.decode('utf-8')

        issued_at = TokenFormatter.creation_time(token)
        issued_at = ks_utils.isotime(at=issued_at, subsecond=True)

# The following code, adjusts the expiration time if the token was~RAFT

        if raftFlag:
            expires_at = kr.expirationTime
            expires_at = BasePayload._convert_float_to_time_string(expires_at)

        expires_at = timeutils.parse_isotime(expires_at)
        expires_at = ks_utils.isotime(at=expires_at, subsecond=True)
# end of expiration~adjustment

        return (user_id, methods, audit_ids, system, domain_id, project_id,
                trust_id, federated_group_ids, identity_provider_id,
                protocol_id, access_token_id, app_cred_id, issued_at,
                expires_at)
```

Listing A6. The source code of the first set of our experiments.

```
import requests
import os
import time
import json
from RAFT import ClientRAFT,KeystoneRAFT

base_url = "http://192.168.122.29" #The IP address of our OpenStack~server

def get_scoped_token(username, password, domain,project_name):
    data= {
        "auth": {
            "identity": {
                "methods": ["password"],
                "password": {
                    "user": {
                        "domain": {
                            "name": domain
                        },
                        "name": username,
                        "password": password
                    }
                }
            },
            "scope": {
                "project": {
                    "domain": {
                        "name": "Default"
                    },
                    "name":  project_name
                }
            }
        }
    }
    url = base_url + "/identity/v3/auth/tokens"
    r = requests.post(url,json=data)
    return r.headers["X-Subject-Token"]

def check_token(token):
    url = base_url + "/identity/v3/auth/tokens"
    h = {"X-Auth-Token" : token,"X-Subject-Token" : token}
    r = requests.get(url, headers=h)
    return~r

def firstExp1():
    print("Creating 100 Fernet tokens")
    start_time= time.time()
    for i in range(0,100):
        fernet_token = get_scoped_token("admin","123","Default","admin")
    execution_time = time.time() - start_time
    print(execution_time)
```

```python
def firstExp2():
    print("Creating 100 RAFTs + a Fernet token")
    start_time= time.time()
    fernet_token = get_scoped_token("admin","123","Default","admin")
    for i in range(0,100):
        raft_builder = ClientRAFT(fernet_token)
        raft_builder.SetCommand("")
        raft_builder.Finalize()
    execution_time = time.time() - start_time
    print(execution_time)

def firstExp3():
    print("Creating and validating 100 Fernet tokens")
    start_time= time.time()
    for i in range(0,100):
        fernet_token = get_scoped_token("admin","123","Default","admin")
        check_token(fernet_token)
    execution_time = time.time() - start_time
    print(execution_time)

def firstExp4():
    print("Creating and validating 100 RAFTs with no command")
    start_time= time.time()
    fernet_token = get_scoped_token("admin","123","Default","admin")
    for i in range(0,100):
        raft_builder = ClientRAFT(fernet_token)
        raft_builder.SetCommand("")
        raft_token= raft_builder.Finalize(1)
        check_token(raft_token)
    execution_time = time.time() - start_time
    print(execution_time)

def firstExp5to7(CmdLength):
    print("Creating and validating 100 RAFTs with a hypothetical 200 characters command")
    start_time= time.time()
    fernet_token = get_scoped_token("admin","123","Default","admin")
    cmd = "a" * CmdLength
    for i in range(0,100):
        raft_builder = ClientRAFT(fernet_token)
        raft_builder.SetCommand(cmd)
        raft_token= raft_builder.Finalize(10)
        check_token(raft_token)
    execution_time = time.time() - start_time
    print(execution_time)

def firstExpIssueTime():
    print("Locally Creating 100 RAFTs experiment")
    fernet_token = get_scoped_token("admin","123","Default","admin")
    cmd = "a" * 20
    start_time= time.time()
    for i in range(0,100):
        raft_builder = ClientRAFT(fernet_token)
        raft_builder.SetCommand(cmd)
        raft_token= raft_builder.Finalize(10)
    execution_time = time.time() - start_time
    print(execution_time)

def firstExpVerificationTime():
    print("Locally creating and validation 100 RAFTs experiment")
    fernet_token = get_scoped_token("admin","123","Default","admin")
    cmd = "a" * 20
    total_time = 0
    for i in range(0,100):
        raft_builder = ClientRAFT(fernet_token)
        raft_builder.SetCommand(cmd)
        raft_token= raft_builder.Finalize(10)
        start_time= time.time()
        kv = KeystoneRAFT();
        kv.ValidateRAFT(raft_token)
        execution_time = time.time() - start_time
        total_time+=execution_time
    print(total_time)
```

Listing A7. Examining the overhead of using RAFT with some commands.

```
import requests
import os
import time
import json
import expCommandLength
from RAFT import ClientRAFT,KeystoneRAFT

base_url = "http://192.168.122.29" # the IP address of our OpenStack~server

def get_project_list(token):
    url = base_url + "/identity/v3/projects"
    h = {"X-Auth-Token" : token,"X-RAFT-Token" : token}
    r = requests.get(url,headers=h)
    if r.status_code == 200:
        return r.content
    else:
        print(r.content)
        return~None

def create_VM(token, vm_name):
    url = base_url + "/compute/v2.1/servers"
    h = {"X-Auth-Token" : token,"X-RAFT-Token" : token, "Accept": "application/json", "Content-Type":
    ↪  "application/json", "User-Agent": "python-novaclient"}
    data ={
        "server": {
            "name": vm_name,
            "imageRef": "ce0afaaa-e236-47c6-95e8-47c7694eb74c",
            "flavorRef": "1",
            "max_count": 1,
            "min_count": 1,
            "networks": [{"uuid": "5eeb14b4-47a9-44aa-bade-b225b7713a6b"}]
        }
    }
    r = requests.post(url,json=data,headers=h)
    return~r

# the default values used in the following function are only valid in our server.
def create_Volume(token, volume_name, vol_size=1 , project_id="08b72d6e4f2b465d96e9e0db2f10d232" ,
↪  imageId= "ce0afaaa-e236-47c6-95e8-47c7694eb74c" ):
    url = base_url + "/volume/v2/" + project_id +  "/volumes"
    h = {"X-Auth-Token" : token,"X-RAFT-Token" : token, "Accept": "application/json", "Content-Type":
    ↪  "application/json", "User-Agent": "python-novaclient"}
    data = {
        "volume": {
            "status": "creating",
            "name": volume_name,
            "imageRef": imageId,
            "attach_status": "detached",
            "volume_type": "lvmdriver-1",
            "size": vol_size
        }
    }

    r = requests.post(url,json=data,headers=h)
    return~r

def get_imagelist(token):
    url = base_url + "/image/v2/images"
    h = {"X-Auth-Token" : token,"X-RAFT-Token" : token}
    r = requests.get(url,headers=h)
    if r.status_code == 200:
        j = json.loads(r.content)
        return j
    else:
        print(r.content)
        return~None

def SecondExp1():
    print("Creating a Volume using Fernet")
    for j in range(0,5):
        fernet_token = get_scoped_token("admin","123","Default","admin")
```

```
        start_time= time.time()
        for i in range(0,1):
            create_Volume(fernet_token, "vreza"+str(i))
        execution_time = time.time() - start_time
        print(execution_time)

def SecondExp2():
    print("Creating a Volume using RAFT")
    for j in range(0,5):
        fernet_token = get_scoped_token("admin","123","Default","admin")
        start_time= time.time()
        for i in range(0,1):
            raft_builder = ClientRAFT(fernet_token)
            raft_builder.SetCommand(expCommandLength.get_create_volume_sample())
            raft_token= raft_builder.Finalize(100)
            create_Volume(raft_token, "vreza"+str(i))
        execution_time = time.time() - start_time
        print(execution_time)

def SecondExp3():
    print("Creating a Virtual Machine using Fernet")
    for j in range(0,5):
        fernet_token = get_scoped_token("admin","123","Default","admin")
        start_time= time.time()
        for i in range(0,1):
            create_VM(fernet_token, "vm"+str(j))
        execution_time = time.time() - start_time
        print(execution_time)

def SecondExp4():
    print("Creating a Virtual Machine using RAFT")
    for j in range(0,5):
        fernet_token = get_scoped_token("admin","123","Default","admin")
        start_time= time.time()
        for i in range(0,1):
            raft_builder = ClientRAFT(fernet_token)
            raft_builder.SetCommand(expCommandLength.get_create_server_sample())
            raft_token= raft_builder.Finalize(100)
            create_VM(raft_token, "vm"+str(j))
        execution_time = time.time() - start_time
        print(execution_time)

def SecondExp5():
    print("Getting image list using Fernet")
    for j in range(0,5):
        fernet_token = get_scoped_token("admin","123","Default","admin")
        start_time= time.time()
        for i in range(0,1):
            get_imagelist(fernet_token)
        execution_time = time.time() - start_time
        print(execution_time)

def SecondExp6():
    print("Getting image list using RAFT")
    for j in range(0,5):
        fernet_token = get_scoped_token("admin","123","Default","admin")
        start_time= time.time()
        for i in range(0,1):
            raft_builder = ClientRAFT(fernet_token)
            raft_builder.SetCommand(expCommandLength.get_image_list_command())
            raft_token= raft_builder.Finalize(2)
            get_imagelist(raft_token)
        execution_time = time.time() - start_time
        print(execution_time)

def SecondExp7():
    print("Getting project list using Fernet")
    for j in range(0,5):
        fernet_token = get_scoped_token("admin","123","Default","admin")
        start_time= time.time()
        for i in range(0,1):
            get_project_list(fernet_token)
        execution_time = time.time() - start_time
        print(execution_time)
```

```
def SecondExp8():
    print("Getting project list using RAFT")
    for j in range(0,5):
        fernet_token = get_scoped_token("admin","123","Default","admin")
        start_time= time.time()
        for i in range(0,1):
            raft_builder = ClientRAFT(fernet_token)
            raft_builder.SetCommand(expCommandLength.get_project_command())
            raft_token= raft_builder.Finalize(2)
            get_project_list(raft_token)
        execution_time = time.time() - start_time
        print(execution_time)
```

References

1. Hogan, K.; Maleki, H.; Rahaeimehr, R.; Canetti, R.; van Dijk, M.; Hennessey, J.; Varia, M.; Zhang, H. On the Universally Composable Security of OpenStack. In Proceedings of the 2019 IEEE Secure Development (SecDev) Conference, Tysons Corner, VA, USA, 23–25 September 2019.
2. Srinivasan, V.U.; Angal, R.; Sondhi, A. OAuth Framework. US Patent 8,935,757, 13 January 2015.
3. Recordon, D.; Reed, D. OpenID 2.0: A platform for user-centric identity management. In *Proceedings of the Second ACM Workshop on Digital Identity Management*; ACM: New York, NY, USA, 2006; pp. 11–16.
4. Auth0. Introduction to JSON Web Tokens. Available online: https://jwt.io/introduction (accessed on 6 May 2026).
5. Jones, M.; Hardt, D. The OAuth 2.0 Authorization Framework: Bearer Token Usage. October 2012. Available online: https://www.rfc-editor.org/rfc/rfc6750 (accessed on 6 May 2026).
6. MIT Kerberos. Kerberos: The Network Authentication Protocol. Available online: https://web.mit.edu/kerberos/ (accessed on 6 May 2026).
7. OpenStack. OpenStack: Open Source Cloud Computing Infrastructure. Available online: https://www.openstack.org/ (accessed on 6 May 2026).
8. OpenStack. Identity Tokens. In *OpenStack Security Guide*. Available online: https://docs.openstack.org/security-guide/identity/tokens.html (accessed on 6 May 2026).
9. OpenStack. OpenStack Use Cases. Available online: https://www.openstack.org/use-cases/ (accessed on 6 May 2026).
10. OpenInfra Foundation. 2025 Annual Report. Available online: https://openinfra.org/annual-report/2025/ (accessed on 6 May 2026).
11. cURL. cURL: Command Line Tool and Library for Transferring Data with URLs. Available online: https://curl.se/ (accessed on 6 May 2026).
12. Fett, D.; Campbell, B.; Bradley, J.; Lodderstedt, T.; Jones, M.B.; Waite, D. OAuth 2.0 Demonstrating Proof-of-Possession (DPoP). RFC 9449. 2023. Internet Engineering Task Force (IETF). Available online: https://www.hjp.at/doc/rfc/rfc9449.html (accessed on 6 May 2026).
13. Campbell, B.; Bradley, J.; Sakimura, N.; Jones, M.B. OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens. RFC 8705. 2020. Internet Engineering Task Force (IETF). Available online: https://www.rfc-editor.org/rfc/rfc8705.html (accessed on 6 May 2026).
14. Curity. The Phantom Token Pattern. 2023. Available online: https://curity.io/resources/learn/phantom-token-pattern/ (accessed on 28 April 2026).
15. Kaliski, B. PKCS# 7: Cryptographic Message Syntax Version 1.5. The Internet Society (1998) 1998. Available online: https://www.rfc-editor.org/rfc/rfc2315.html (accessed on 6 May 2026).
16. Kundu, A.; Mahindru, R.; Mohindra, A.; Salapura, V.; Viswanathan, M. Automatic Security Parameter Management and Renewal. US Patent 10/243,936, 26 March 2019.
17. White, C.; Edwards, S. Client Services for Applied Key Management Systems and Processes. US Patent 9,967,289, 8 May 2018.
18. Adefarati, T.; Bansal, R. Integration of renewable distributed generators into the distribution system: A review. *IET Renew. Power Gener.* **2016**, *10*, 873–884. [CrossRef]