Project acronym:	**PrEstoCloud**
Project full name:	**Proactive Cloud Resources Management at the Edge for efficient Real-Time Big Data Processing**
Grant agreement number:	**732339**

D5.7 PrEstoCloud Security Enforcement Mechanism - Iteration 1

Deliverable Editor:	**Panagiotis Gouvas (UBITECH)**
Other contributors:	**Giannis Ledakis (UBITECH), Panagiotis Parthenis (UBITECH), Giannis Tsantilis (UBITECH), Kostas Theodosiou (UBITECH)**
Deliverable Reviewers:	**Yevgeniya Sulema (NAM), Salman Taherizadeh (CVS)**
Deliverable due date:	**30/6/2018**
Submission date:	**23/8/2018**
Distribution level:	**Public**
Version:	**1.0**

This document is part of a research project funded by the Horizon 2020 Framework Programme of the European Union

Change Log

Version	Date	Amended by	Changes
0.1	11/6/2018	Panagiotis Gouvas	Table of Contents
0.2	18/6/2018	Panagiotis Gouvas	Introduction
0.3	15/6/2018	Panagiotis Gouvas	Chapter 2 – Risk Assessment Analysis
0.4	3/7/2018	Giannis Ledakis, Panagiotis Gouvas	Chapter 3- Trusted Computing principles
0.5	12/7/2018	Kostas Theodosiou	Chapter 3- Cjdns
0.6	9/8/2018	Giannis Tsantilis, Panagiotis Gouvas	Chapter 4 – Perimeter Security
0.7	10/8/2018	Panagiotis Gouvas	Conclusion – version submitted to reviewers
0.8	14/8/2018	Salman Taherizadeh	First Reviewer comments
0.9	17/8/2018	Yevgeniya Sulema	Second Reviewer comments
1.0	22/8/2018	Panagiotis Gouvas	Final Edition

Table of Contents

List of Tables

List of Figures

List of Abbreviations

The following table presents the acronyms used in the deliverable.

Abbreviation	*Description*
ARM	Advanced RISC Machine
AWS	Amazon Web Services
BPF	Berkley Packet Filter
CAPEC	Common Attack Pattern Enumeration and Classification
CPE	Common Platform Enumeration
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DoS	Denial of Service
eBPF	enhanced Berkley Packet Filter
JIT	Just-In-Time
JWT	JSON Web Token
KCM	Kernel Connection Multiplexe
TCP/IP	Transmission Control Protocol/Internet Protocol
TPM	Trusted Platform Module
QoS	Quality of Service
SRK	Storage Root Keys
VM	Virtual Machine
XDP	eXpress Data Path

Executive Summary

This deliverable reports on the work performed under the first development period of the PrEstoCloud framework as far as security aspects are concerned. In the first development period, special emphasis was given to the realization of a complete lifecycle of the platform's components. To this end, the security requirements have been defined based on a threat analysis that is adopted to the final architecture and the lifecycle that has been implemented. The **scope** of the security enforcement layer is to **propose and implement all appropriate control elements** that will **minimize the cyber risk** of the PrEstoCloud framework.

The identification of the proper control elements is inferred based on a traditional cyber risk assessment methodology. According to this methodology [1], the entire framework is 'decomposed' in several components that are addressed as cyber assets. Each cyber asset may be associated with relevant cyber threats and may expose multiple vulnerabilities. The combination of a vulnerability and attack through exploitation of the specific vulnerability by an adversary comprises an attack scenario. Since exploring all possible attack scenarios within one deliverable is not possible, because of the vast space of applicable threats, a proper abstraction has been performed. Based on this abstraction a set of **generic adversarial models** have been explored.

The identified adversarial models relate mainly to **a)** the usage of **untrusted computational** resources; **b)** to the usage of **untrusted communication medium** (for signaling) and **c)** to the **application-specific** attack vectors that are exposed during the deployment of an application over the PrEstoCloud framework. Each of these adversarial models are mitigated using diverse mechanisms. More specifically, as far as untrusted resources are conserned, the proposed countermeasure is the adoption of hardware-assisted crypto-primitives. More specifically, the adoption of **Trusted Platform Module (hereinafter TPM)** is essential in order to prevent specific type of attacks. One could argue that such a protocol is very restrictive since it pre-assumes the purchase of additional dedicated hardware elements. Yet recent developments in the specification [2] have proposed the adoption of **virtual TPM** modules that are much more convenient regarding their integration. In the first phase of development, specific PrEstoCloud jobs have been developed that were executed over ARM processor and were using functional calls to a hardware TPM module. The aim of this proof of concept was to prove that containerized applications can use the deployment manager of PrEstoCloud while in parallel they can bind the TPM module for trust assurance.

Regarding untrusted communications, the control elements are separated in two different categories. The first category has to do with the communication with data center resources while the latter relates to the communication with the edge resources. These two different types of resources entail completely different security requirements. Data center resources are already protected by the network manager of the correspondent IaaS provider. Therefore**, VMs that belong to the same administrative zone belong to an existing overlay that share a uniform isolation policy**. Moreover, the layer-3 security policy of these VMs can be configured through the network manager of the IaaS.

However, as far as edge resources are concerned, this assumption is not true since there is absolutely **no sense of isolation**. As a result, raw computational resources that are onboarded in an cluser of edge nodes must be able to trust each other and communicate securely. These requirements have been covered by the incorporation **of Cjdns protocol** in the PrEstoCloud device stack. Through this protocol mutual trust of resources (that join and leave the cluster dynamically) is achieved along with point-to-point encryption on layer-3. Finally, regarding the perimeter security at the component level, sophisticated state of the art techbology for kernel-based packet processing has been utilized. More, specifically, deployed components that are part of PrEstoCloud deployment actions are managed by a PrEstoCloud agent. This agent is able to interpret perimeter-security policies that are enforced using **eBPF** [3] technology.

1. Introduction

The scope of this deliverable is to elaborate on the security aspects of the PrEstoCloud architecture. PrEstoCloud aims to deliver a framework which allows the deployment of complex applications on top of diverse computational resources which span from data center resources to edge resources. As such it offers advanced orchestration and administration capabilities targeting the optimal use of cloud resources while trying to satisfy runtime QoS policies. These runtime QoS policies may result on several reconfigurations in the deploymed components since some components may be reconfigured or even redeployed in different computational resources. Computational resources span from public cloud offerings such as Amazon AWS, Google Cloud Platform to private cloud offerings (e.g. openstack-based) and edge resources (e.g. IoT devices).

As it is easily inferred, the operational environment is rather complex based on the diversification of the resource types and the various APIs that are used by the PrEstoCloud orchestration component. One of the issues that contribute to the increase of the overall complexity is the security assurance of the entire platform. PrEstoCloud offers multiple 'attack vectors' and as such it should be accompanied by proper mechanisms that guarantee a certain level of security on multiple layers of the architecture.

One of the major goals of this deliverable is to shed light upon the **exposed attack vectors** and to analyze them. In order to perform this task in a more methodological way, a cyber-risk-oriented methodology will be adopted. This approach enables the alignment of the identified security requirements with existing standards that derive from the cyber security domain. For the sake of broader comprehension, this 'alignment' is mandatory. According to the principles of cyber-risk management, a calculated risk is the product of a combination that takes under consideration existing identified vulnerabilities and applicable threats that may be used to take advantage of these vulnerabilities. The scope of this deliverable is to demystify the relevant threats and vulnerabilities that are considered as applicable in the case of PrEstoCloud.

Beyond this identification, special emphasis will be given to the proposal of existing cyber defensive mechanisms (also addressed as cyber control elements) that can be used in order to mitigate relative threats or vulnerabilities. The way these cyber controls elements contribute in the mitigation of risk will be also analysed.

The structure of the current deliverable is as follows; **Chapter 2** introduces the reader to the concepts of Cyber Risk Assessment. This is essential since the terms that will be introduced in this chapter will be used consistently in the rest of the deliverable. Furtheremore, in this chapter the identification of relevant attack types will be listed taking under consideration the lifecycle of the PrEstoCloud platform. **Chapter 3** will be devoted to the **security guarantees** that are required taking under consideration the **untrusted communication medium** between the PrEstoCloud orchestration components and the employed resources. In addition, **Chapter 4** elaborates on the **software-defined programmability** of the resources that is performed based on administrator-defined preferences. Using this type of programmability, dynamic security policies can be applied during the operation of PrEstoCloud platform. Finally, **Chapter 5** concludes this deliverable.

2. Cyber Risk Assessment of PrEstoCloud platform

2.1 Definitions related to Cyber Risk Assessment

In the context of cybersecurity, the identification of abuse cases (use cases that the results of the interaction are harmful to the system) is practically identical with the process of exploring the attack surface of each potential target. Exploiting a target contributes to several consequences which may span from financial, environmental etc. However, a successful exploitation can be performed by an adversary only through the usage of an existing/identified vulnerability. During the security analysis of an ecosystem, it is highly crucial to have a clear view of the concepts that are used during the analysis. In order to come up to a common model that will be used during the lifecycle of the entire project, we will adopt concepts from the domain of Risk Assessment. Figure 1 provides a high-level view of concepts that are widely used in the (Cyber) Risk Assessment domain.

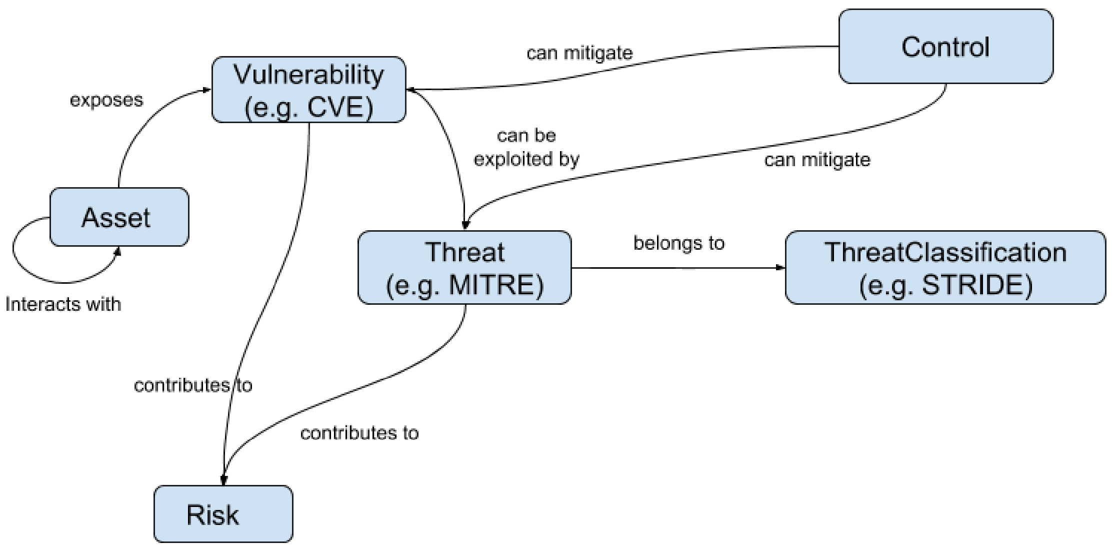


Figure 1 Concepts of Cyber Risk Assessment

The starting point is "Asset". An asset can be a cyber, physical or cyber-physical element within an organization that is used in the frame of business operations. Furthermore, an asset can be something tangible or intangible. Furthermore, an asset may entail a specific business value by itself. Irrelevant of the nature and the type of an asset, each asset may entail several Vulnerabilities.

A vulnerability is the stepping stone of the adversary since s/he must identify one in order to harm an Asset. It should be clarified that vulnerabilities are inherent properties of assets and depend on their nature/type. Creating an abstract model for a vulnerability is rather challenging since many parameters have to be taken into consideration. Although there are many models that have been proposed, the Common Vulnerability and Exposure (CVE) [4] model is considered predominant. According to this model a vulnerability is characterized by two properties; exploitability, i.e. how easily you can make use of the vulnerability and impact i.e. how dangerous is the exploitation of this vulnerability. The model defines sub-scores for Confidentiality, Integrity and Availability (a.k.a. CIA) consequences. An open repository for disclosed vulnerabilities can be found here: https://www.cvedetails.com.

A vulnerability may be exploited during an 'Attack' that is able to make use of this vulnerability. In the cyber security domain, the terms Attack and Threat coincide. In the frame of this document and in the scope of the entire project these terms will be semantically equivalent. Analogous to the Vulnerabilities, Threats maintain their own metamodel. A widely used model/taxonomy is Common Attack Pattern Enumeration and Classification (CAPEC) [5]. The objective of the CAPEC effort is to describe most common attack patterns classified in an intuitive manner. The attacks were defined using the CAPEC taxonomy which is defined and described below. As seen in Table 1 several types of threats are identified and categorized per domain and mechanism.

Table 1 - Indicative CAPEC Attack Tree Classification

Category	Attack
Excavation – 116[1]	JSON Hijacking (aka JavaScript Hijacking) - 111
	Directory Indexing -127
	Common Resource Location Exploration - 150
	Cross-Domain Search Timing - 462
	Generic Cross-Browser Cross-Domain Theft - 468
	Probe Application Screenshots - 498
	Probe Application Error Reporting - 54
	Probe Application Queries - 545
	Probe Application Memory - 546
Interception – 117	Sniffing Network Traffic - 158
	Accessing/Intercepting/Modifying HTTP Cookies - 31
	Harvesting Usernames or User IDs via Application API Event Monitoring - 383
	Intent Intercept - 499
Footprinting – 169	Host Discovery – 292
	Port Scanning – 300
	Network Topology Mapping – 309
	Malware-Directed Internal Reconnaissance - 529
Fingerprinting – 224	OS Fingerprinting - 311
	Application Fingerprinting - 541
Social Information Gathering Attacks – 404	Social Information Gathering via Research - 405
	Social Information Gathering via Dumpster Diving - 406
	Social Information Gathering via Pretexting - 407
	Information Gathering from Traditional Sources - 408
	Information Gathering from Non-Traditional Sources - 409
Information Elicitation via Social Engineering - 410	Social Information Gathering via Pretexting - 407
Injection - 152	Code Injection – 241

The successful exploitation of a vulnerability through an attack leads to Impact. In order to prevent this impact, specific Control elements should be installed. However, as depicted in Figure 1 a control element can be applied at the Vulnerability Level or at the Threat Level (never to both of them). Imagine of a scenario where a software remotely exploitable vulnerability is identified. If the administrator patches this vulnerability s/he applies a control at the vulnerability Level. If an administrator setup a Web Application

[1] The number following each CAPEC category or member represents its unique identification number, CAPEC-ID, which enables a fast discovery and retrieval of its description.

Firewall (a.k.a. WAF) in a virtual machine s/he applies a control at the Threat Level since many attacks can be simultaneously mitigated (e.g. SQL injection, Remote File inclusion etc.)

The knowledge of assets, vulnerabilities and applicable threats are the prerequisites that have to be defined in order to quantify Risk. Sometimes, the concept of Risk is misunderstood with the concept of Threat; yet it is not. In general, risk is calculated through an equation as follows:

Risk =f(TL , VL, IL) [*Equation 1*]

where TL represents the possibility of an attack being materialized, VL represents the exploitability of the Vulnerability and IL the impact of an exploitation. As it can be inferred by equation 1, in order to calculate tangible cyber risks a complete knowledge of relevant threats shoule exist. Since the official cyber threats in the MITRE repository exceed 700 enties and the **PrEstoCloud attack vector cannot be excaustively fingerprinted** a simplified model will be employed. According to this simplified model all threats will be grouped based on some **high-level threat levels** (or even addressed as threat categories). The simplified model that will be used is the STRIDE [6] model.

2.2 The STRIDE Threat Model and its application in PrEstoCloud

As already mentioned the CAPEC Threat Model is very excaustive; yet not usefull in an upper-level analysis. To this end, we will adopt a classification model that was proposed by Microsoft. According to this model, any cyber-threat (i.e. attack) can belong to one of the following categories:

- **Spoofing:** Assuming the identity or the attributes of another user.
- **Tampering:** Refers to scenarios in which a user maliciously changes data. This can be referring to both static, persisted data, as well to data being transmitted.
- **Repudiation:** Users may dispute certain action if there is insufficient data in order to prove otherwise. The dispute can relate to both legal and illegal operations.
- **Information Disclosure:** Refers to users gaining accessing to data to which they are not supposed to have access to. As in the case of tampering, this includes both stored and data in flow.
- **Denial of Service:** Denying the access of users to services which they are entitled to use by disturbing the user, the platform providing the service, or the communication channels.
- **Elevation of Privilege:** Refers to the scenarios in which a malicious user has gained privileged access to the system.

Based on this classification, we will elaborate on realistic attack scenarios that are applicable in the case of PrEstoCloud. The following Table 2 summarizes these attack scenarios.

Table 2 – PrEstoCloud Attack scenarios

STRIDE category	**Attack Scenario**
Spoofing	PrEstoCloud maintains two categories of credentials. The first category is the PrEstoCloud-platform credentials which are used in order to validate (authenticate) a user against the platform. The second category relates to the credentials that are used by PrEstoCloud in order to interact with IaaS resources (e.g. OpenStack). Theoretically, identity-spoofing attacks can be performed in order to imitate both credential-owners. Another attack surface which is extremely prone to spoofing attacks is the communication protocol of the edge resources. Since edge resources are decentralized, there is high possibility for someone to "pretend" a honest resource and join a cluster of edge resources.
Tampering	PrEstoCloud is developed using microservices design patterns. As a result, the solution consists of multiple components. More of them expose a REST api that can be consumed. A tampering attack could use the REST signatures of the components invocation and alter the arguments.
Repudiation	These types of attack are not directly applicable to PrEstoCloud.

Information Disclosure	Information disclosure can be achieved in two different layers. First of all, the signaling that is used between the PrEstoCloud components and the IaaS providers can be overheared. This communication may include sensitive information. Moreover, the apps per se that are deployed through PrEstoCloud can serve traffic that is also susceptible to overhearing.
Denial of Service	A DoS attack in only applicable to the apps that are deployed through PrEstoCloud. Yet an application that is under attack should not affect the overall system performance.
Elevation of Priviledge	This is by far the most complicated; yet the most advanced attack vector. According to this adversarial secario a user can craft a specific 'malicious' application that will be deployed through PrEstoCloud. Upon deployment the malicious app will fingerprint the environment and will try to compromise its working environment

2.3 PrEstoCloud Control Elements

As already mentioned in Subsection 2.1, the aim of the exploration of the applicable attacks is to propose a defensive strategy that minimizes their impact. In our case, the defensive strategy cannot be delegated to a monolithic architectural component since the nature of the threats are quite different.

Table 3 – PrEstoCloud Control elements

STRIDE category	**Proposed Controls**
Spoofing	All credentials that are being used by the business logic of the PrEstoCloud platform are **encrypted** using **symmetric encryption algorithm**. More specifically, **AES-256 [7]** is used in order to encrypt sensitive data of the platform. However, it should be noted that the usage of AES is not eliminating all threats since in all symmetric encryption schemes the major concern is the protection of the unique symmetric key that has been used. For this purpose, PrEstoCloud will rely in the near future on a secret-sharing service such as **Vault** [8]. Regarding the Edge resources, there should be a mechanism that assures the trustworthiness of the onboarded device. This mechanism is provided by the **Cjdns protocol** which is analysed on Chapter 3. Cjdns is resolving these issues using **asymmetric cryptography** principles. More specifically, the protocol enforced each edge resource to generate a pair of keys; one public and one private. PrEstoCLoud is communicating with the edge resources through one Gateway which is appointed with the task of handling all signaling with the platform. Each edge resource tha joins the cluster should provide beforehand its public key to the gateway.
Tampering	In order to protect the API calls of the platform, an authorization scheme has been employed. This authorization scheme was based on **JSON-Web-Token technology** (a.k.a. JWT) [9] since the API is JSON enabled. This solution is also compliant with the standard RFC7519 [10].
Repudiation	These types of attack are not directly applicable to the PrEstoCloud project.

Information Disclosure	In general, PrEstoCloud should **not provide trusted-computing guaranetees** since this is a functionality that is assumed to be given by the IaaS provider. One de-facto standard that has been proposed is **TPM** which defines an API that applications can use in order to perform cryptographic operations. This API is offered by specific hardware that must be attached to the IaaS hardware. Using this technique, a compromised IaaS cannot perform information disclosure of sensitive data. Another attack surface is the communication between the edge resources which should always support **transport-level encryption**. As it will be examined on chapter 3, this is covered by the Cjdns protocol.
Denial of Service	DoS attacks are a class of perimeter security attacks that require packet filtering control mechanisms. In most cases, these mechanisms are provided by **firewalling mechanisms**. However, since in PrEstoCloud there are **multiple IaaS** providers that can be used, there should be a solution which is **vendor agnostic and software defined in parallel.** The implemented mechanism relies on a Linux-kernel capability which is called BPF. The mechanism is detailed on chapter 4.
Elevation of Priviledge	As in the case of information disclosure, the control elements for such types of attacks are offered by trusted computing mechanisms.

2.4 Trusted Computing through TPM

As already mentioned, trusted computing is not a focal point for PrEstoCloud. However, it would be preferable for the project, the apps that are orchestrated by the orchestration component to interoperate with TPM-enabled devices. Trusted Platform Module (TPM) is a cryptographic coprocessor, a devoted microcontroller or chip, intended to protect hardware by **joining cryptographic keys into devices**. In addition, TPM is a standard for **a protected crypto processor** and has been included on most PCs and laptop motherboards produced in the past decade . TPM is the core component of trusted computing and includes capabilities such as **random number generation**, **secure generation of cryptographic keys**, **remote attestation** and **sealed storage**. A variety of vendors such as Infineon, Broadcom, Atmel, STMicroelectronics, and Nuvoton produce TPM chips, while many PC manufacturers shipping TPM-enabled PCs such as Dell, Lenovo, HP, Toshiba, and Fujitsu. TPM's technical specification is publicly available, driven by an industry consortium the Trusted Computing Group (TCG) [1]. The specification was also written to be platform independent. The TCG's architecture for hardware-based security was motivated by increasingly sophisticated malware attacks in the late 1990s. The latest version TPM 2.0, is a major redesign of the specification which adds new functionalities and fixes weaknesses of the former TPM 1.2.

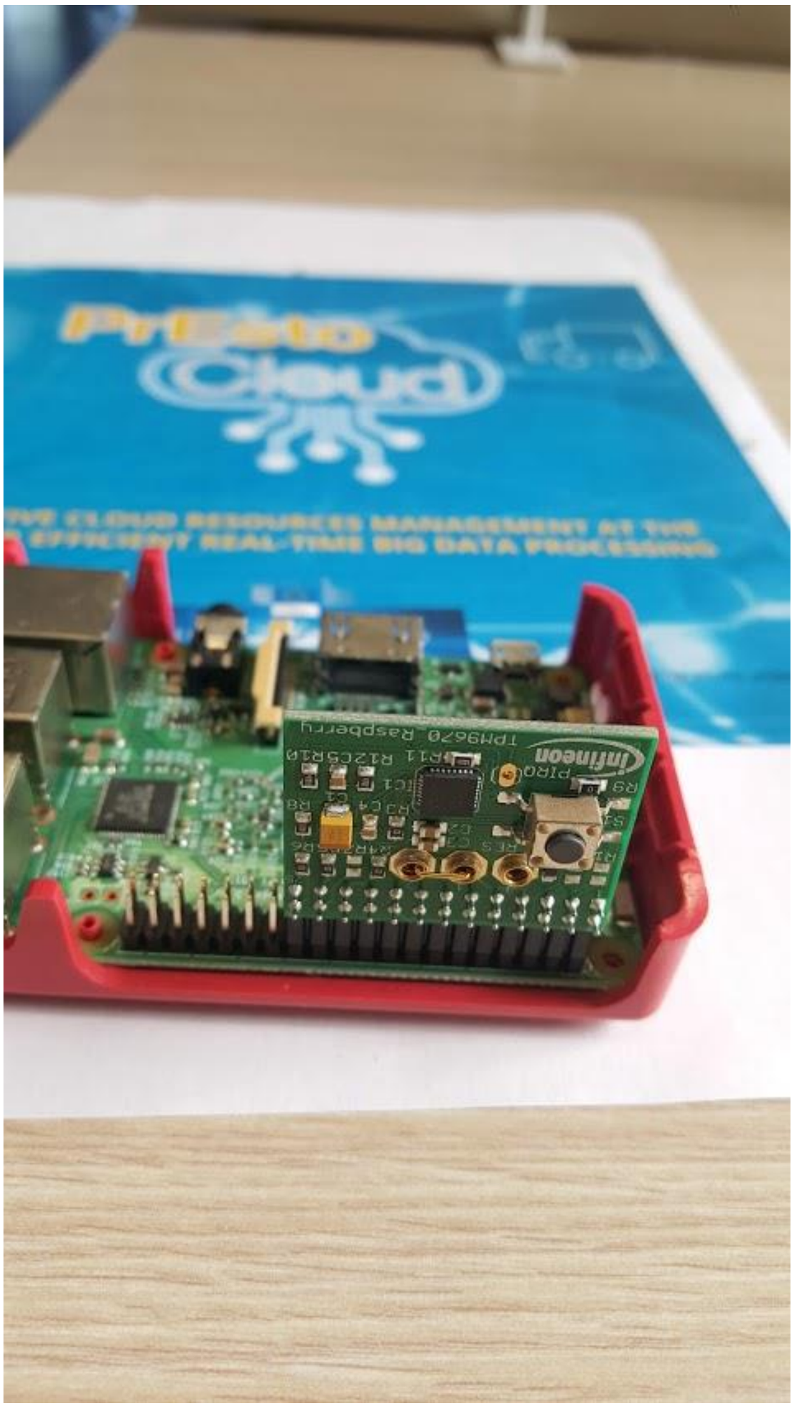


Figure 2 PrEstoCloud TPM testbed based on Infineon module on a Raspberry device

The TPM contains two important functional components **a cryptographic engine** that can perform encryption, digital signatures, and hashing, and a special register set called **Platform Configuration Registers** (PCRs). Because of limited storage space, the TPM does not normally store keys permanently. Rather, it contains multiple Storage Root Keys (SRKs) that are stored persistently, allowing completely separate applications to use the TPM with less coordination. Moreover, the TPM can store platform measurements that help ensure that the platform is trustworthy. This is critical because authentication and attestation are necessary to ensure safer computing in all environments. With the remote attestation, other platforms in the trusted network can decide, to which extent they can trust information from another device, while at the same time maintain the user anonymity. Moreover, TPM Software Stack (TSS) is a TCG software standard that allows applications to intercept the stack at various levels in a portable manner. Applications written to the TSS should work on any system that implements a compliant TSS.

As depicted on Figure 2, a specific testbed has been setup based on a raspberry edge device and TPM2.0 compiant device purchased by Infenion. The goal is to perform a deployment of an application graph which consists of multiple components and one (or more) of its components require TPM functional primitives.

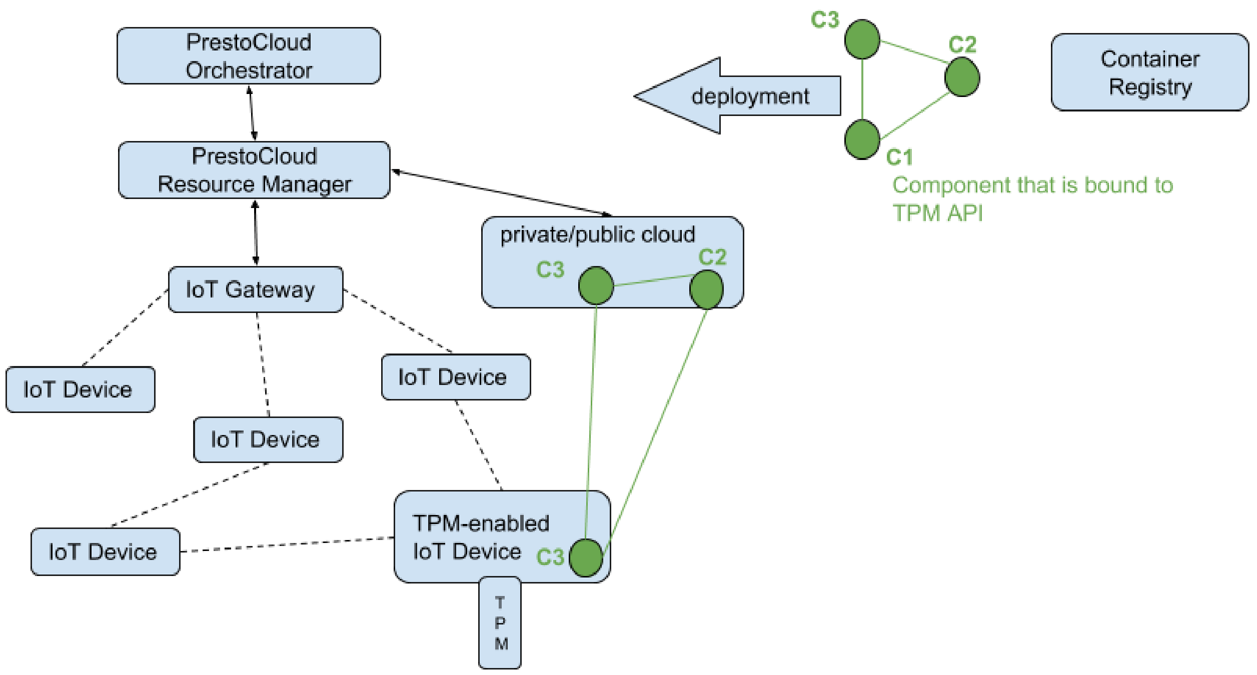


Figure 3 Flow of TPM usage

The flow ot TPM usage is depicted on Figure 3. According to this flow, PrEstoCloud must handle a deployment request of an application. The application is practically a graph consisting of multiple components. One of the components must be placed in a resources where trusted crypto-primitives should exist. The deployment request is transformed to an optimization problem that attempts to satisfy some hard and soft constraints. One of the hard constraints is the existence of a TPM module by one candidate. If a proper resource is found the component will be deployed to the TPM-enabled device and a **proper device mapping** to the booted container **must be performed**. It should be noted that proper device mapping of TPM will be **fully implemented in the second phase of the project**.

3. Trust and Security for loosely coupled edge resources

3.1 Security Requirements regarding edge resources

The PrEstoCloud edge resources are an example of a loosely coupled decentralized mesh networking scheme. A mesh network can be described as a network where everyone on it is the same as everyone else (devices on a meshnet, such as your computer or wireless router, are usually called "nodes"). This is unlike a traditional hierarchical network like the networks of today's internet providers where individual users have to get access from routers above them, and they from routers above them. Nodes on a mesh network cooperate to relay traffic for each other, working together to ensure that everything gets where it needs to go.

The operational characteristics of the edge resources have been thoroughly described in Deliverable 3.9 (Spatiotemporal Processing Capabilities - Iteration 1) [11]. As it was elaborated, an edge cluster consists of multiple nodes that can join or leave spontaneously. In this environment, from the security point of view, there are two major issues that have to clarified. The first relates to when do we consider that an IoT device is trusted and the second relates to how do we guarantee the security of the spontaneous links.

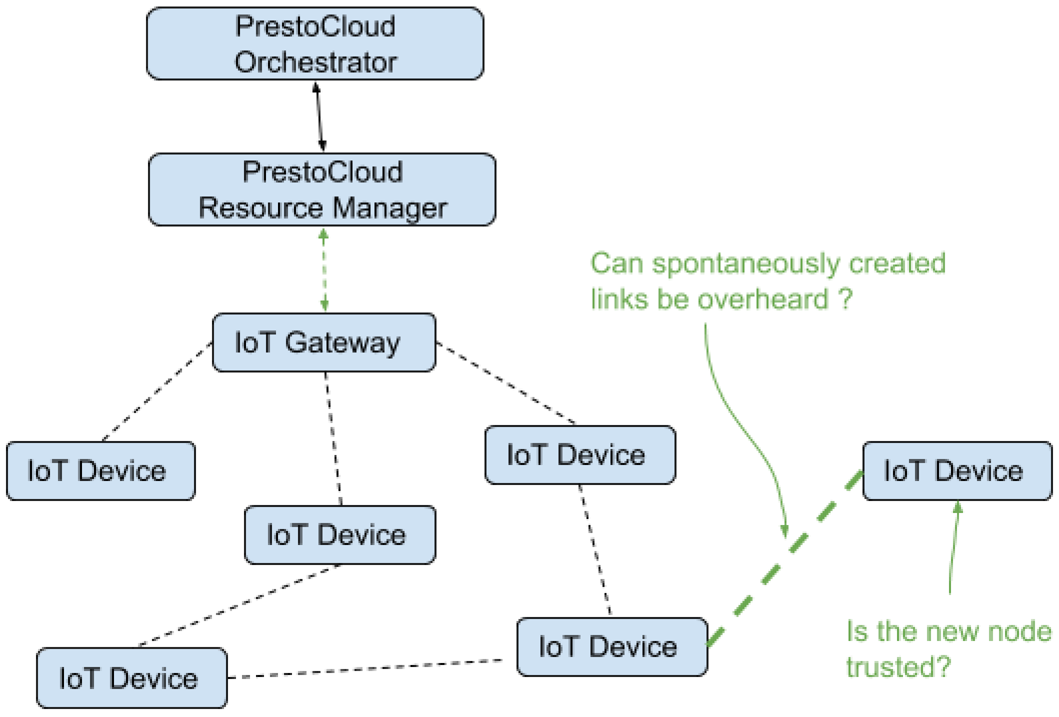


Figure 4 Challenges during the onboarding of new nodes

As thorouly explained in [11], each edge device is running a specific application which is addressed as PreStoCloud Device Stack. This application is responsible for many aspects. One of these aspects, maybe the most crucial, is the formulation of the mesh links and the routability based on a non-static routing protocol. One of the core decisions that have been made during the formulation of the stack was the incorporation of a specific protocol called Cjdns in order to achive end-to-end routability among the edge nodes. However, the decision for the adoption of the specific protocol was also driven based on its security features i.e. **its ability to create encrypted overlays**. More specifically, Cjdns is using asymmetric encryption principles in order to achieve trust among mesh nodes. To this end, each node is generating a public-private key and **peer-acceptance is bound** to the exchange of public key. However, the protocol allows the **"blind" acceptance** of peer requests. Irrelevant of the peer-acceptance mode, **the routed traffic within the overlay is encrypted at the peer level**. For the sake of completeness, the following section provides some details of the adopted protocol while section 3.3 elaborates on the encryption mechanism.

3.2 Fundamental concepts of Cjdns

Cjdns is the only meshnet protocol available that offers fully distributed and yet still global addressing. This means that any meshnet node running Cjdns will interconnect with any other Cjdns node automatically, that no central authority or control of any kind is necessary, and that all Cjdns meshnets are compatible by their very nature. In fact, there is really only ever one single global Cjdns meshnet, even if some parts of it are not currently linked to some others. The moment they are linked, they will function as one, and with the global addressing it means that Cjdns meshnet can scale to the entire planet.

Cjdns also includes secure end-to-end encryption built in to the protocol at the very lowest levels. In fact, the encryption is part of what allows for the global distributed addressing. When a new Cjdns node is set up, a cryptographic key pair [12] is generated and the node's IP address is derived from that key. Any communication to your node is automatically encrypted with your key, and communications with any other IP address can be cryptographically verified as secure and genuine by comparing the keys used to the address itself. What this all means is that nobody on the meshnet can see your private communications except for you and the node you are actually communicating with.

Another interesting feature of Cjdns is it's efficient routing. Because it is designed to have lower resource requirements (primarily memory) than traditional internet routing, Cjdns uses a system of routing that minimizes the amount of information a router needs to do it's job. A side benefit of this is that no individual node knows any more about who you are communicating with then it absolutely needs to, which generally means it only knows what the next hop is along the path, not the final destination. This further enhances your privacy, beyond what is even possible on the internet without additional specialized tools like Tor [13]. (Note, however, that Cjdns does not offer actual anonymity anywhere near the level that Tor does, nor is it intended too. It does, however, offer just enough to make mass surveillance impractical, while not sacrificing performance like Tor does.)

On a technical note, Cjdns is a "layer 3" [14] protocol that runs directly on top of the MAC layer, intended as a replacement for the standard TCP/IP protocol used in today's network and internet connectivity. All you need is a plain direct ethernet or ad-hoc wireless connection, nothing more, for it to work. It actually implements TCP/IP on top of itself and offers a standard IPv6 interface to applications. All your current software and servers will work just fine without modification, provided they support IPv6 .It does not rely on any other meshnet or internetworking protocols to function. It coexists with other such protocols without issue, will work over nearly any kind of connection, over the current internet, and also will route current internet traffic over itself. Most users currently have and use both Cjdns and typical internet connections on their computers simultaneously.

Hyperboria [15] is the name given to the Cjdns meshnet as it exists today. Up to this point, it has primarily existed as a proof of concept and testbed for the developing Cjdns protocol. As the protocol matures, however, and projects meant to bring meshnet connectivity to the general public move forward, it will become the seed from which the new global meshnet will grow.

How Cjdns Works

Cjdns is made of three major components which are woven together. There is a **switch**, a **router**, and a **CryptoAuth** module. With total disregard for the OSI layers, each module is inherently dependent on both of the others. The router cannot function without routing in a small world which is made possible by the switch, the switch is blind and dumb without the router to command it, and without the router and switch, the CryptoAuth has nothing to protect.

Switch

The switch design is unlike an IP or Ethernet router, it does not need to have knowledge of the globally unique endpoints in the network. Like ATM switching, the packet header is altered at every hop and the reverse path can be derived at the end point or at any point along the path but unlike ATM, the switch does not need to store active connections and there is no connection setup.

Definitions

- Interface: A point-to-point link to another Cjdns switch. This may be emulated by Ethernet frames, UDP packets or other means.
- Self Interface: A special Interface in each switch, packets sent for this interface are intended for the node which this switch is a part of. Upon reaching the ultimate hop in its path, a packet is sent through the Self Interface so it can be handled by the next layer in the node.
- Director: A binary codeword of arbitrary size which when received by the switch will direct it to send the packet down a given Interface.
- Route Label: An ordered set of Directors that describe a path through the network.
- Encoding Scheme: The method by which a switch converts one of its internal Interface ID (EG: array index) to a Director and converts a Director back to its internal representation. Encoding schemes may be either fixed width or variable width but in the case of variable width, the width must be self-describing as the Directors are concatenated in the Route Label without any kind of boundary markers.
- Encoding Form: A single representation form for encoding of a Director. For a given Encoding Form, there is only one possible way to represent a Director for a given Interface. A variable width Encoding Scheme will have multiple Encoding Forms while a fixed width Encoding Scheme will have only one.
- Director Prefix: For switches which implement variable width encoding, the least significant bits of the Director is called the Director Prefix and is used to determine the width of the Director. **NOTE** Because the Route Label is read from least significant bit to most significant bit, the Director Prefix is actually the bits furthest to the *right* of the Director.

Flow

When a packet comes in to the switch, the switch uses its Encoding Scheme to read the least significant bits of the Route Label in order to determine the Director and thus the Interface to send the packet down. The Route Label is shifted to the right by the number of bits in the Director, effectively *removing* the Director and exposing the Director belonging to the next switch in the path. Before sending the packet, the switch uses its Encoding Scheme to craft a Director representing the Interface which the packet came *from*, does a bitwise reversal of this Director and places it in the empty space at the left of the Route Label which was exposed by the previous bit shift. In this way, the switches build a mirror image of the return Label allowing the endpoint, or any hop along the path, to derive the return path by simple bitwise reversal without any knowledge of the Encoding Schemes used by other nodes.

Router

A router has 3 functions; it periodically searches for things, responds to searches, and forwards packets. When a router responds to a search, it responds with nodes which it thinks will get closer to the destination. The responses MUST NOT have addresses which are, in address space distance, further from the responding node than the search target, and they MUST NOT have routes which begin with the same interface as the route to the querying node. These two simple rules provide that no search will ever go in circles and no route will ever go down an interface, only to be bounced back. While the second rule can only be enforced by the honor system, querying nodes MUST double check the first rule. The node doing the searching adds the newly discovered nodes to their routing table and to the search, then continues the search by asking them.

Upon receiving a search response containing one's own address, a node SHOULD purge all entries from its table whose routes begin with that route. This will control the proliferation of redundant routes.

The "address space distance" between any two given addresses is defined as the result of the two addresses XOR'd against one another, rotated 64 bits, then interpreted as a big endian integer. The so called "XOR metric" was pioneered in the work on Kademlia DHT system [16] and is used to forward a packet to someone who probably knows the whole route to the destination. The 64 bit rotation of the result is used to improve performance where the first bits of the address is fixed to avoid collisions in the IPv6 space.

Adding nodes to the routing table from search responses is done by splicing the route to the node which was asked with the route to the node in the response, yielding a route to the final destination through them.

Routers choose the node to forward a packet to in a similar way to how they answer search queries. They select nodes from their routing table except in this case the selection contains only one node. The packet is sent through the CryptoAuth session corresponding to this node and the label for getting to it is applied to the packet before sending to the switch. The "search target" for forwarding a packet is the IPv6 destination address of the packet.

3.3 The CryptoAuth

The CryptoAuth is a mechanism for wrapping interfaces, you supply it with an interface and optionally a key, and it gives you a new interface which allows you to send packets to someone who has that key. Like the rest of Cjdns, it is designed to function with best effort data transit. The CryptoAuth handshake is based on piggybacking headers on top of regular data packets and while the traffic in handshake packets is encrypted and authenticated, it is not secure against replay attacks and has no forward secrecy if the private key is compromised. The CryptoAuth header adds takes 120 bytes of overhead to the packet, causing a fluctuating MTU.All cryptoAuth headers are 120 bytes long except for the data header which is 4 bytes and the authenticatedData header which is 20 bytes. The first 4 bytes of a CryptoAuth header is used to determine it's type, if they are zero, it is a "connect to me" header, if they are equal to the obfuscated value of zero or one, it is a handshake1 packet and if they are the obfuscated value of two or three, it is a handshake2 packet. if it is the obfuscated value of a number exceeding three, it is a data or authenticated data packet.

There are 5 types of CryptoAuth header:

1) Connect To Me Packet (Used to start a session without knowing the other node's key.)

When a node receives a connect to me packet from a node which it does not know, it should establish a session and send back a handshake1 packet, if it already has a session, it should drop the packet silently. The connect to me packet has no useful information except for it's system state and "Permanent Public Key" field, the rest of the packet should be filled with random.

2) Handshake1 (Hello Packet - The first message in beginning a session)

A handshake1 packet contains an authentication field, a random nonce, the node's perminent public key, a poly1305 authenticator field, and the temporary public key followed by the content, all encrypted and authenticated using the perminent public keys of the two nodes and the random nonce contained in the packet. The content and temporary key is encrypted using crypto_box_curve25519poly1305xsalsa20() function.

3) Handshake2 (Key Packet- The second message in a session)

A handshake2 packet likewise contains an authentication field, a random nonce, a perminent public key field (which is not used but still must be present) a poly1305 authenticator, and an encrypted temporary key and content. this time the temporary key and content is encrypted using the perminent key of the sending node and the temporary public key of the other party (which was sent in the handshake1 packet).

4) Data Packet

The Data Packet is the default data packet. The first 4 bytes are used as the nonce , in this case it is a 24 byte nonce and crypto_box_curve25519poly1305xsalsa20_afternm() is used to encrypt and decrypt the data, using the shared secret computed using crypto_box_curve25519poly1305xsalsa20_beforenm() with one peer's temporary public key and the other peer's temporary secret key (both roles are symmetrical and produce the same shared secret).As crypto_box_curve25519poly1305xsalsa20_afternm() requires a 24-byte nonce, the 4-byte nonce is copied in a 24-byte array that is passed to the primitive. The peer which sent the

Key packet writes it in the first four bytes of the array; and the other peer to the next four bytes. This distinction is made so that packets from one peer can not be sent back to this peer to make it believe it is from the other peer. A peer should always send its nonce in increasing big-endian order, otherwise they will be dropped by the Replay Protector of the receiver if they are too much out of order.

5) Authentication Field (Describe a way to hash a password)

This field allows a node to connect using a password or other shared secret, the authtype subfield specifies how the password should be hashed, the auth- derivations field specifies how many times the shared secret hash function must be run and the authentication hash code is the 5 bytes of the sha256 of the result from the hash function. Auth type of 0 indicates that the node is not offering authentication credentials in the handshake, auth type 1 is a trivial sha256 of the password to create the hash. When a client presents authentication credencials, the result of doing the number of derivations given on the password is then appended to the key generated by point multiplication of the public and private keys and sha256'd to generate the shared secret. This authentication scheme is designed to be resistant to MiTM attacks as well as attacks on the underlying asymmetric cryptography which protects the connection. Basicly it is using a password as what it is, a shared secret. If the auth type field is set to 0, the key generated by scalar multiplication will not be fed to sha256, it will instead be hashed using hsalsa20 as normally used in crypto_box_curve25519xsalsa20. If the A bit (at 14 byte offset) is set, the connection will use poly1305 to authenticate all packets, regardless of whether auth-type is 0.

```
                     1               2               3
   0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
0 |   AuthType    |                                               |
  +-+-+-+-+-+-+-+-+           AuthType Specific                   +
4 |                                                               |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
8 |                            Unused                             |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 5 The "AuthType Specific" field is specific to the authentication type(source: https://github.com/cjdelisle/cjdns/blob/master/doc/Whitepaper.md).

Auth Types:

1) **AuthType Zero**

AuthType Zero is no authentication at all. If the AuthType is set to zero, all AuthType Specific fields are disregarded and SHOULD be set to random numbers.

This AuthType is the one used in Key packets and for inner (end-to-end) cryptoauth sessions.

2) **AuthType One**

```
                   1               2               3
   0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
0 |   Auth Type   |                                               |
  +-+-+-+-+-+-+-+-+           Hash Code                           +
4 |                                                               |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
8 |                             Unused                            |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 6 AuthType One is a SHA-256 based authentication method(source : https://github.com/cjdelisle/cjdns/blob/master/doc/Whitepaper.md)

With AuthType One, the shared secret (password) is hashed once and the result is appended to the 32 byte output from scalar multiplication of the curve25519 keys these 64 bytes are hashed again with SHA-256 to make the symmetric key to be used for the session. It is also hashed a second time and the result copied over the first 8 bytes of the authentication header before the AuthType field is set. The effect being that the "Hash Code" field contains bytes 2 through 8 the hash of the hash of the password (counting indexes from 1). This is used as a sort of username so that the other end knows which password to try using in the handshake.

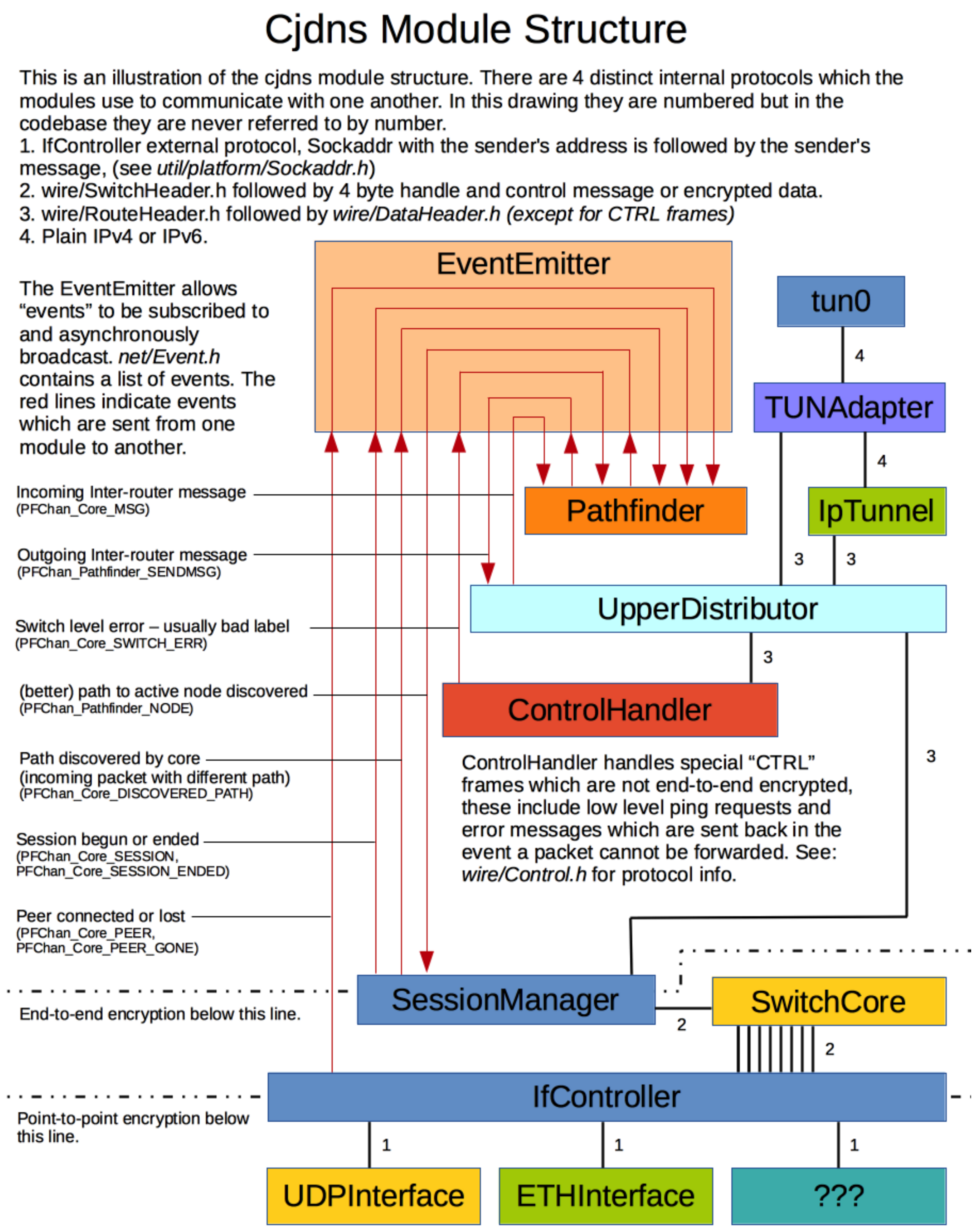
3) AuthType Two

AuthType Two is similar to AuthType One, except that bytes 2 to 8 of the Hash Code are bytes 2 to 8 of the SHA-256 hash of a login, which is known by the received of the packet to be associated with the password used for making the symmetric secret.

Packet Lifecycle

The journey of a packet begins at the user interface device (TUN or similar). The user sends an IPv6 packet which comes in to the TUN device and enters the engine, it is checked to make sure its source and destination addresses are valid and then a router lookup is made on the destination address. Cjdns addresses are the first 16 bytes of the SHA-512 of the SHA-512 of the public key. All addresses must begin with the byte 0xFC otherwise they are invalid, generating a key is done by brute force key generation until the result of the double SHA-512 begins with 0xFC. After the router lookup, the node compares the destination address to the address of the next router, if they are the same, the inner layer of encryption is skipped. Assuming they are different, the IPv6 header is copied to a safe place and a CryptoAuth session is selected for the destination address, or created if there is none, and the packet content is passed through it. The IPv6 header is re-applied on top of the CryptoAuth header for the content, the packet length field in the IPv6 header is notably *not* altered to reflect the headers which are now under it.

The packet is now ready to send to the selected router. Switch gets a CryptoAuth session for the router it's sending to from it's pool, if there isn't any, it creates one. The packet is cryptoauthed, protecting the ipv6 header and adding another crypto header (again, 120 bytes unless the handshake is complete). Upon receiving the packet, the next node sends the packet through its CryptoAuth session thus revealing the switch header and it sends the packet to its switch. The switch most likely will send the packet out to another endpoint as per the dictate of the packet label but may send it to its router, eventually the node for which the packet is destined will receive it. The router, upon receiving the packet will examine it to see if it appears to be a CryptoAuth Connect To Me packet, Handshake1, or Handshake2. If it is one of these, it will insert the IPv6 address, as derived from the public key in the header, into a hashtable so it can be looked up by the switch label, otherwise it will do a lookup. If the Address cannot be found in its hashtable, it will try asking the router if it knows of a node by that label and if all fails, the packet will be dropped. Base on the IPv6 address, it will lookup the CryptoAuth session or create one if necessary, then pass the opaque data through the CryptoAuth session to get the decrypted IPv6 header. If the source address for the packet is the same as the double SHA-512 of the public key for the router from which it came, it's assumed to have no inner layer of encryption and it is written to the TUN device as it is. If its source address is different, it is passed back through a CryptoAuth session as selected based on the source IPv6 address. The IPv6 header is then moved up to meet the content (into the place where the CryptoAuth header had been) and the final packet is written out to the TUN device.


Cjdns Module Structure

This is an illustration of the cjdns module structure. There are 4 distinct internal protocols which the modules use to communicate with one another. In this drawing they are numbered but in the codebase they are never referred to by number.
1. IfController external protocol, Sockaddr with the sender's address is followed by the sender's message, (see *util/platform/Sockaddr.h*)
2. wire/SwitchHeader.h followed by 4 byte handle and control message or encrypted data.
3. wire/RouteHeader.h followed by *wire/DataHeader.h (except for CTRL frames)*
4. Plain IPv4 or IPv6.

The EventEmitter allows "events" to be subscribed to and asynchronously broadcast. *net/Event.h* contains a list of events. The red lines indicate events which are sent from one module to another.

ControlHandler handles special "CTRL" frames which are not end-to-end encrypted, these include low level ping requests and error messages which are sent back in the event a packet cannot be forwarded. See: *wire/Control.h* for protocol info.

EventEmitter
tun0
4
TUNAdapter
4
IpTunnel
Pathfinder
Incoming Inter-router message (PFChan_Core_MSG)
Outgoing Inter-router message (PFChan_Pathfinder_SENDMSG)
3
3
UpperDistributor
Switch level error – usually bad label (PFChan_Core_SWITCH_ERR)
3
(better) path to active node discovered (PFChan_Pathfinder_NODE)
ControlHandler
Path discovered by core (incoming packet with different path) (PFChan_Core_DISCOVERED_PATH)
3
Session begun or ended (PFChan_Core_SESSION, PFChan_Core_SESSION_ENDED)
Peer connected or lost (PFChan_Core_PEER, PFChan_Core_PEER_GONE)
SessionManager
SwitchCore
End-to-end encryption below this line.
2
2
IfController
Point-to-point encryption below this line.
1
1
1
UDPInterface
ETHInterface
???

Figure 7 Cjdns Module Structure (source: https://www.skycoin.net/blog/development-updates/development-update-72/)

4. Advanced Perimeter Security Policy Enforcement

4.1 Architecture of Perimeter security

One of the traditional issues in security is the policy enforcement of perimeter security rules. This practically relates to the problem of translating a specific allowance/droping policy to tangible packet filtering business logic. The question at this point is the following "who is provide this business logic"? Most of the IaaS providers offer perimeter firewalls that can be configured using high-level APIs. This SDN-alike principle is absolutely fine in case all deployments are being performed in one IaaS provider. In PrEstoCloud this is not the case. The entire project focuses on multi-IaaS capabilities that have to be supported. In order to avoid vendor lock-in issues, in the frame of PrEstoCloud a generic kernel-oriented method was used which is totally IaaS-agnostic. The method is presented in Figure 8.

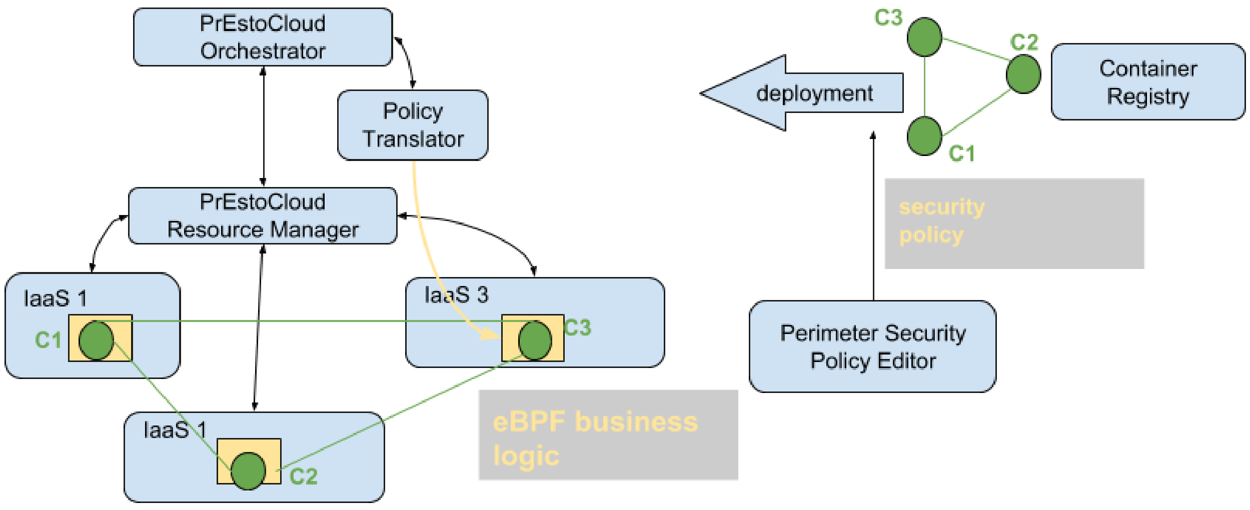


Figure 8 Perimeter security business logic

As it is depicted, PrEstoCloud offers a Perimeter Security Policy Editor which is used to provide allowance/dropping packet policies based on an **abstract format**. These rules are later on translated in specific scripts that are directly **executable by the kernel of the operating system that is hosting the docker container of the component**. This executable is formatted in **eBPF script** and is **extremely efficient**. The following section will introduce the reader to the insights of the BPF technology

4.2 Introduction to eBPF technology

The need for fast network packet inspection and monitoring was obvious in early versions of UNIX with networking support. In order to gain speed and avoid unnecessary copying of packet contents between kernel and userspace, the notion of a kernel packet filter agent was introduced [17],[18]. Different UNIX-based OSes implemented their own versions of these agents. The solution later adopted by Linux was the BSD Packet Filter introduced in 1993 [19], which is referred to as Berkeley Packet Filter (BPF). This agent allows a userspace program to attach a filter program onto a socket and limit certain dataflows coming through the socket in a fast and effective way. Linux BPF originally provided as et of instructions that could be used to program a filter: this is nowadays referred to as classic BPF (cBPF).

Later a new, more flexible, and richer set was introduced, which is referred to as extended BPF (eBPF)[20]. Linux BPF can be viewed as a minimalistic virtual machine construct [21] that has a few registers, a stack and an implicit program counter. Different operations are allowed in side a valid BPF program, such as fetching data from the packet, arithmetic operations using constants and input data, and comparison of results against constants or packet data. The Linux BPF subsystem has a special component, called verifier, that is used to check the correctness of a BPF program; all BPF programs must approved by this component before they can be executed. Verifier is a static code analyzer that walks and analyzes all branches of a BPF program; it tries to detect unreachable instructions, out of bound jumps, loops etc. Verifier also enforces the maximum length of aBPF program to 4096 BPF instructions [22].

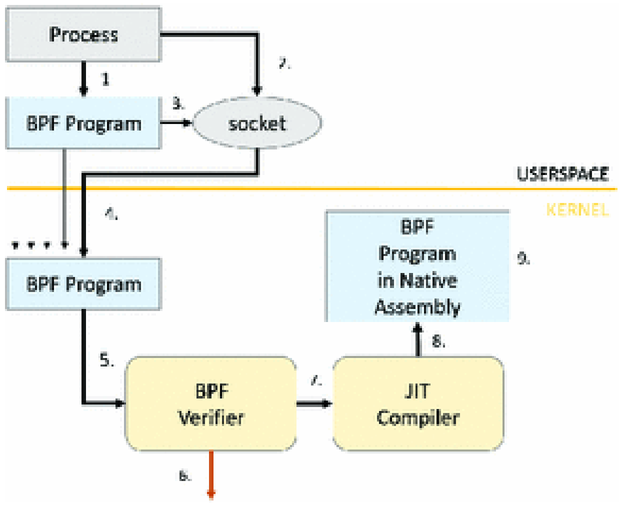


Figure 9 Typical flow of a BPF program

While originally designed for network packet filtering, nowadays Linux BPF is used in many other areas, including system call filtering in SecComp [23], tracing [24]and Kernel Connection Multiplexer(KCM) [25]. In order to improve packet filtering performance even further, Linux utilizes a Just-In-Time (JIT) compiler [26] to translate BPF instructions into native machine assembly. JIT support is provided for all major architectures, including x86 and ARM. This JIT compiler is not enabled by default on standard Linux distributions, such as Ubuntu or Fedora, but it is typically enabled on network equipment such as routers**.** Figure 9 shows a simplified view of how a BPF program is loaded and processed in the Linux kernel. First, a user space process creates a BPF program, a socket, and attaches the program to the socket (steps 1-3). Next, the program is transferred to the kernel, where it is fed to the verifier to be checked (steps 4-5). If the checks fail (step 6), the program is discarded and the user space process is notified of the error; otherwise, if JIT is enabled, the program gets processed by the JIT compiler (step 7). The result is the BPF program in native assembly, ready for execution when the associated Socket receives data (steps 8-9). The program is placed in the kernel module mapping memory space, using the vmalloc() kernel memory allocation primitive.

Fundamentally eBPF is still BPF: it is a small virtual machine which runs programs injected from user space and attached to specific hooks in the kernel. It can classify and do actions upon network packets. For years it has been used on Linux to filter packets and avoid expensive copies to user space, for example with tcpdump. However, the scope of the virtual machine has changed beyond recognition over the last few years.

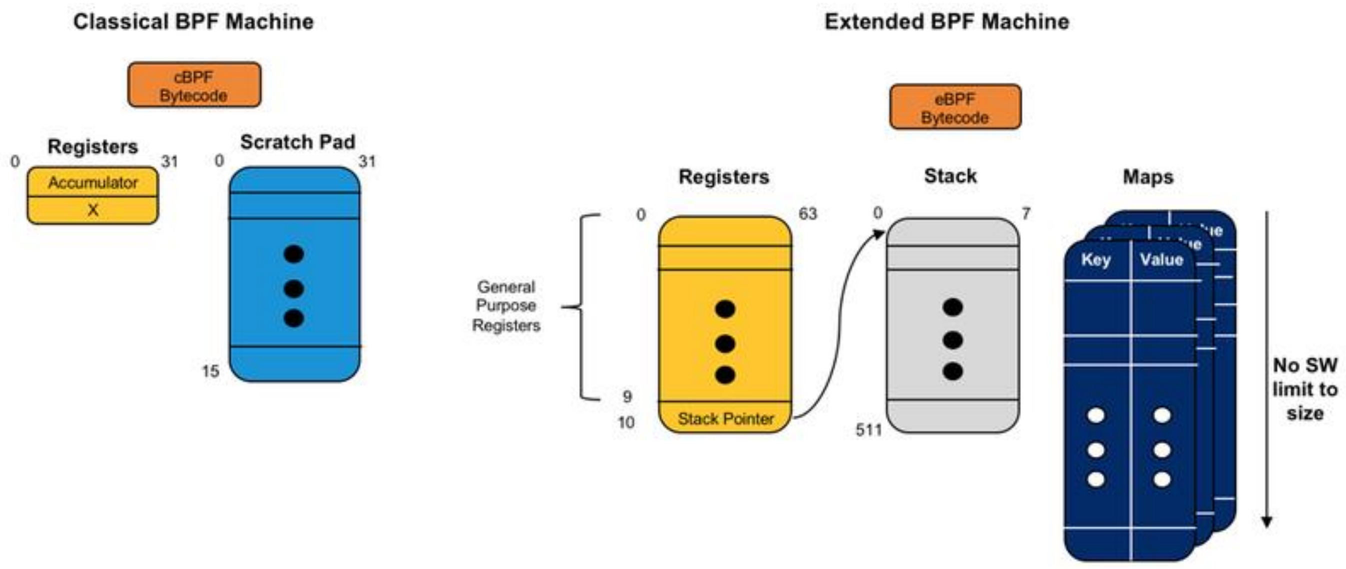


Figure 10 A comparison of the cBPF vs. eBPF machines

Classic BPF (cBPF), the legacy version, consisted of a 32-bit wide accumulator, a 32-bit wide 'X' register which could also be used within instructions, and 16 32-bit registers which are used as a scratch memory store. This obviously led to some key restrictions. As the name suggests, the classic Berkeley Packet Filter was mostly limited to (stateless) packet filtering. Anything more complex would be completed within other subsystems.

eBPF significantly widened the set of use cases for BPF, through the use of an expanded set of registers and of instructions, the addition of maps (key/value stores without any restrictions in size), a 512 byte-stack, more complex lookups, helper functions callable from inside the programs, and the possibility to chain several programs. Stateful processing is now possible, as are dynamic interactions with user space programs. As a result of this improved flexibility, the level of classification and the range of possible interactions for packets processed with eBPF has been drastically expanded.

But new features must not come at the expense of safety. To ensure proper exercise of the increased responsibility of the VM, the verifier implemented in the kernel has been revised and consolidated. This verifier checks for any loops within the code (which could lead to possible infinite loops, thus hanging the kernel) and any unsafe memory accesses. It rejects any program that does not meet the safety criterions. This step, performed on a live system each time a use tries to inject a program, is followed by the BPF bytecode being JITed into native assembly instructions for the chosen platform.

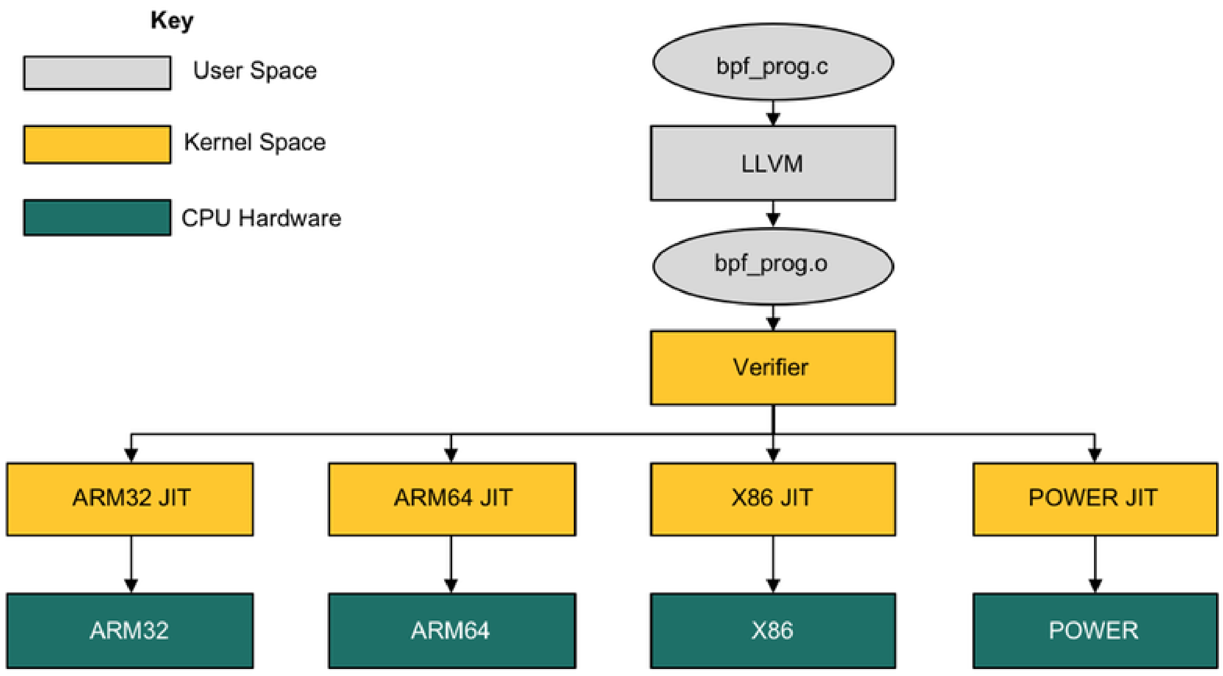


Figure 11 Compilation flow of an eBPF program on the host. Some supported CPU architectures are not displayed

To allow any key functionality which would be difficult to do or optimize within the restrictions of eBPF, there are many helpers which are designed to assist with the execution of processes such as map lookups or the generation of random numbers. The number of hooks for eBPF is proliferating due to its flexibility and usefulness. However, we will focus on those at the lower end of the datapath. The key difference here being that eBPF adds an additional hook in driver space. This hook is called eXpress DataPath, or XDP. This allows users to drop, reflect or redirect packets before they have a skb (socket buffer) metadata structure added to the packet. This leads to a performance improvement of about 4-5X.

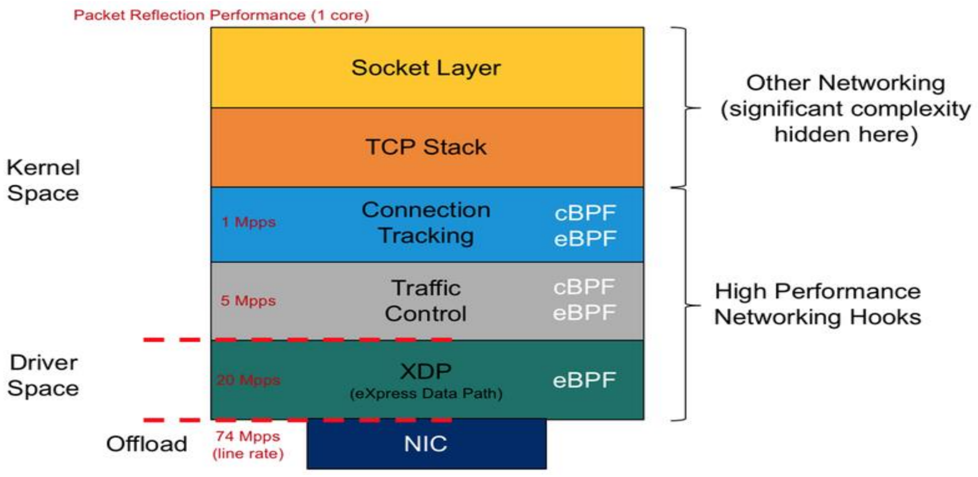


Figure 12 - High-performance networking relevant eBPF hooks with comparative performance for simple use case

4.3 XDP as the evolution of eBPF

XDP is a new system in the kernel, that lets you write custom eBPF programs to filter network packets. We'll call those "XDP programs". The XDP programs run as soon as the packet gets to the network driver (so very very quickly). When an XDP program, it needs to exit with either XDP_TX, XDP_DROP, or XDP_PASS. The XDP packet process includes an in kernel component that processes RX packet-pages directly out of driver via a functional interface without early allocation of **SKB's** [27] or software queues. Normally, one CPU is assigned to each RX queue but in this model, there is **no locking** RX queue, and CPU can be dedicated to busy poll or interrupt model. BPF programs performs processing such as packet parsing, table lookups, creating/managing stateful filters, encap/decap packets, etc. Much of the huge speed gain comes from processing RX packet-pages directly out of drivers RX ring queue, before any allocations of meta-data structures like SKBs occurs.

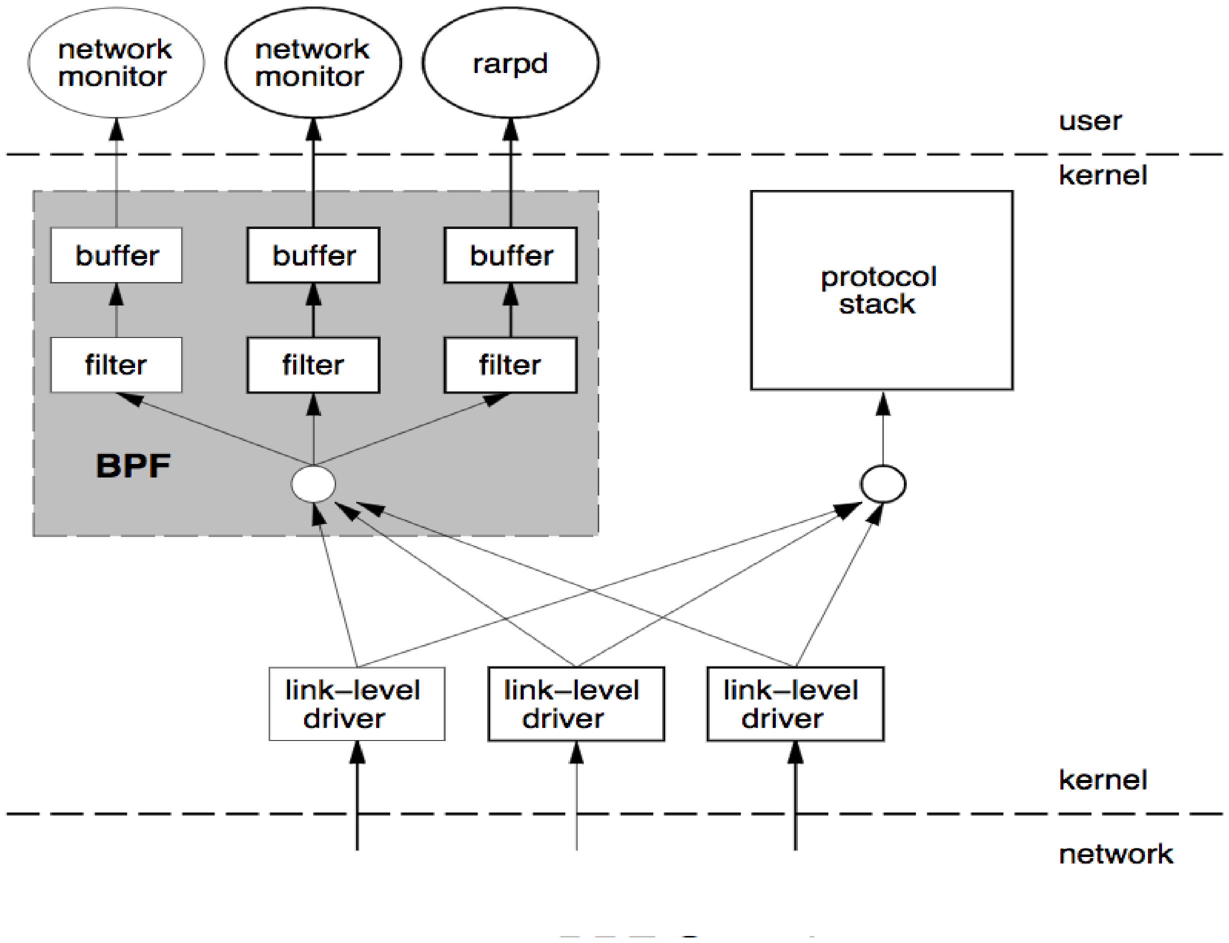


Figure 13 Filtrering network packets with BPF inside the kernel (source: https://zhaozhanxu.com/2018/04/01/Linux/2018-04-01-Linux-eBPF/)

As stated above, the process that eBPF programs go through can be broken down into three distinct steps.

1. **Step 1: Creation of the eBPF program as byte code.** Currently the standard way of creating eBPF programs is to write them as C code and then have LLVM compile them into eBPF byte code which resides in an ELF file. However, the eBPF virtual machine's design is well documented and the code for it and much of the tooling around it is open source. Thus, for *the initiated*, it is entirely possible to either craft the register code for an eBPF program by hand or write a custom compiler for eBPF. Because of eBPF's extremely simple design all functions for an eBPF program, other than the entry point function, must be inlined for LLVM to compile it.
2. **Step 2: Loading the program into the kernel and creating necessary eBPF-maps.** This is done using the bpf syscall in Linux. This syscall allows for the byte code to be loaded along with a declaration of the the type of eBPF program that's being loaded. As of this writing, eBPF has program types for usage as a socket filter, kprobe handler, traffic control scheduler, traffic control action, tracepoint handler, eXpress Data Path (XDP), performance monitor, cgroup restriction, and light weight tunnel. The syscall is also used for initializing eBPF-maps.
3. **Step 3: Attaching the loaded program to a system.** Because the different uses of eBPF are for different systems in the Linux kernel, each of the eBPF program types has a different procedure for attaching to its corresponding system. When the program is attached it becomes active and starts filtering, analyzing, or capturing information, depending on what it was created to do. From here user-space programs can administer running eBPF program including reading state from their eBPF-

maps and, if the program is constructed in such a way, manipulating the eBPF map to alter the behavior of the program.

Due to many different applications of eBPF, the details of step 1, creating the eBPF program, and step 3, attaching it to a system in the kernel, vary by use case. However the core of eBPF, and thus what all applications of it have in common, is step 2. No matter the use case, an eBPF program must be loaded into the kernel and eBPF-maps, if used, must be configured for it. This is all done by the Linux bpf syscall. The Linux bpf syscall has the following signature:

```
int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```
[2]

Note the use of the bpf_attr union. This is a C union which allows for different C structs to be passed to the bpf syscall depending on which command is being used. The code for it can be found in the include/uapi/linux/bpf.h [3] file of the Linux kernel.

There are ten commands for the bpf Linux syscall. Of those ten, six are listed in the man page: BPF_PROG_LOAD, BPF_MAP_CREATE, BPF_MAP_LOOKUP_ELEM, BPF_MAP_UPDATE_ELEM, BPF_MAP_DELETE_ELEM, and BPF_MAP_GET_NEXT_KEY. Of these documented commands, there are really only two types: loading an eBPF program and various manipulations of eBPF-maps. The eBPF-map operations are fairly self descriptive and are used to create eBPF-maps, lookup an element from them, update an element, delete an element, and iterate through an eBPF-map (by using BPF_MAP_GET_NEXT_KEY).

A look at the bpf_enum in include/uapi/linux/bpf.h [4] shows the four other commands: BPF_OBJ_PIN, BPF_OBJ_GET, BPF_PROG_ATTACH, BPF_PROG_DETACH. All together this gives us the following 10 commands.

```
enum bpf_cmd {
   BPF_MAP_CREATE,
   BPF_MAP_LOOKUP_ELEM,
   BPF_MAP_UPDATE_ELEM,
   BPF_MAP_DELETE_ELEM,
   BPF_MAP_GET_NEXT_KEY,
   BPF_PROG_LOAD,
   BPF_OBJ_PIN,
   BPF_OBJ_GET,
   BPF_PROG_ATTACH,
   BPF_PROG_DETACH,
};
```

Furthermore, the Linux bpf syscall man page mentions three eBPF-map types: BPF_MAP_TYPE_HASH, BPF_MAP_TYPE_ARRAY, and BPF_MAP_TYPE_PROG_ARRAY. Digging into the include/uapi/linux/bpf.h [5] file, however, reveals 11 different types as of Linux kernel 4.11. Along with this, the Linux kernel reserves the

[2] *https://github.com/torvalds/linux/blob/v4.11/kernel/bpf/syscall.c#L1031*

[3] https://github.com/torvalds/linux/blob/v4.11/include/uapi/linux/bpf.h#L148

[4] https://github.com/torvalds/linux/blob/v4.11/include/uapi/linux/bpf.h#L73

[5] https://github.com/torvalds/linux/blob/v4.11/include/uapi/linux/bpf.h#L86

first C enum option as BPF_MAP_TYPE_UNSPEC to ensure that zero isn't a valid map type. Presumably, this is in case zero, with it's many forms in C, does not accidentally get passed as the map types argument.

```
enum bpf_map_type {
  BPF_MAP_TYPE_UNSPEC,
  BPF_MAP_TYPE_HASH,
  BPF_MAP_TYPE_ARRAY,
  BPF_MAP_TYPE_PROG_ARRAY,
  BPF_MAP_TYPE_PERF_EVENT_ARRAY,
  BPF_MAP_TYPE_PERCPU_HASH,
  BPF_MAP_TYPE_PERCPU_ARRAY,
  BPF_MAP_TYPE_STACK_TRACE,
  BPF_MAP_TYPE_CGROUP_ARRAY,
  BPF_MAP_TYPE_LRU_HASH,
  BPF_MAP_TYPE_LRU_PERCPU_HASH,
  BPF_MAP_TYPE_LPM_TRIE,
};
```

In only a few years, eBPF has been integrated into a number of Linux kernel components. This makes the outlook of eBPF's future both interesting but also unclear in direction. This is because there are really two futures in question: the future of uses for eBPF and the future of eBPF as a technology. The immediate future of eBPF's uses probably has the most certainty. It's highly likely that the trend of using eBPF for safe, efficient, event handling inside the Linux kernel will continue. Because eBPF is defined by the simple virtual machine it runs on, there is also the potential for eBPF to be used in places other than the Linux kernel. The most interesting example of this, as of this writing, is with Smart NICs.

Because of the large amount of processing power in them, recent Smart NICs have become a target for eBPF. With this, they pose a prime method for a variety of uses, including mitigating DoS attacks and providing dynamic network routing, switching, load balancing, etc. Finally, there is the future of eBPF as a technology. Due to eBPF's restrictive and simple implementation it offers a highly portable and performant way to process events. More than that, however, eBPF forces a change in how problems are solved. It removed objects and stateful code, instead opting for just functions and efficient data structures to store state. This paradigm vastly shrinks the possibilities of a program's design, but in doing so it also makes it compatible with nearly any method of program design. Thus, eBPF can be used synchronously, asynchronously, in parallel, distributed (depending on the coordination needs with the data store), and all other manner of program designs.

5. Conclusions

The current deliverable presented the cyber-security control mechanisms that have been developed/integrated in the frame of PrEstoCloud. The scope of the deliverable was to elaborate on the attack vectors that are exposed by PrEstoCloud framework. PrEstoCloud aims to deliver a platform where complex applications that handle streams can be deployed in multi-cloud resources, including edge resources. The diversification of the resources along with the overall dynamicity of the application lifecycle contribute in the exposure of a relatively big attack surface. In order to minimize the cyber risk of adversaries that may wish to take advantage of this attack surface, a security enforcement layer has been implemented. The ultimate purpose of the security enforcement layer is to propose and implement all appropriate control elements that will minimize the cyber risk of the entire framework.

The identification of the control elements was defined based on a cyber risk assessment methodology, according to which, the framework is broken down in several components. These components are addressed as cyber assets. Each cyber asset may be associated with relevant cyber threats and may expose multiple vulnerabilities. These threats and vulnerabilities can be used by adversaries in order to deliberately damage the confidentiality integrity and availability of the running applications. Such exploitation attempts are also addressed as attack scenarios. As already elaborated, the attack scenarios that are applicable, in the context of PrEstoCloud relate mainly to the usage of untrusted computational resources, to the usage of untrusted communication medium and to the inherent vulnerabilities of the deployed application.

Each of these attack scenarios are mitigated using different control mechanisms. More specifically, as far as untrusted resources are conserned, the proposed countermeasure is the adoption of a TPM module. Such a module can be used in a hardware version or even in a virtualized version. In the first phase of development, a fully functional proof-of-concept was implemented, according to which, specific applications that were orchestrated through the platform used TPM function calls. The proof of concept was dedicated to ARM-based processors since the need of trust assurance is bigger in the case of edge resources.

Regarding untrusted communications, the control elements were separated in two different categories. The first category relates to the communication with data center resources while the latter relates to the communication with the edge resources. As already analyzed, data center resources are already protected by the network manager of the correspondent IaaS provider; hence, VMs that belong to the same administrative zone belong to an existing overlay that share a uniform isolation policy. On the other hand, as far as edge resources are concerned, this assumption is not true since there is lack of isolation. Conclusively, raw computational resources that are onboarded in an edge cluster must be able to trust each other and communicate securely. These requirements have been covered by the incorporation of Cjdns protocol in the PrEstoCloud device stack. Through this protocol mutual trust of resources (that join and leave the cluster dynamically) is achieved along with point-to-point encryption on layer-3.

Finally, regarding the perimeter security at the component level, sophisticated state of the art techbology for kernel-based packet processing has been utilized. More, specifically, since deployed components that are part of PrEstoCloud deployment, are managed by a PrEstoCloud agent, the agent was enhanced so as to interpret perimeter-security policies that are enforced using advanced packet filtering technology such as eBPF.

References

[1] MITRE- Risk Mitigation Planning, Implementation, and Progress Monitoring. Available online: https://www.mitre.org/publications/systems-engineering-guide/acquisition-systems-engineering/risk-management/risk-mitigation-planning-implementation-and-progress-monitoring

[2] TPM Specifications. Available online: https://trustedcomputinggroup.org/resource/tpm-main-specification/

[3] BPF Specifications. Available online: https://www.kernel.org/doc/Documentation/networking/filter.txt

[4] Common Vulnerabilities and Exposures List. Available online: http://cve.mitre.org/

[5] Common Attack Pattern Enumeration and Classification. Available online: https://capec.mitre.org

[6] The STRIDE threat model. Available online: https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)

[7] "Announcing the ADVANCED ENCRYPTION STANDARD (AES)" (PDF). Federal Information Processing Standards Publication 197. United States National Institute of Standards and Technology (NIST). November 26, 2001. Archived (PDF) from the original on March 12, 2017. Retrieved October 2, 2012.

[8] Vault Project. Available online: https://www.vaultproject.io

[9] JWT- JSON Web Tokens. Available online: https://jwt.io

[10] IETF standard RFC7519. Available online: https://tools.ietf.org/html/rfc7519

[11] PrEstoCloud Deliverable D3.9 – Spatiotemporal Processing capabilities http://prestocloud-project.eu/

[12] Wikipedia - Public-key_cryptography. Available online: https://en.wikipedia.org/wiki/Public-key_cryptography

[13] Wikipedia - Tor Network. Available online: https://en.wikipedia.org/wiki/Tor_%28anonymity_network%29

[14] Wikipedia- OSI model. Available online: https://en.wikipedia.org/wiki/OSI_model

[15] Hyperboria framework. Available online: https://hyperboria.net

[16] Maymounkov P., Mazières D. (2002) Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In: Druschel P., Kaashoek F., Rowstron A. (eds) Peer-to-Peer Systems. IPTPS 2002. Lecture Notes in Computer Science, vol 2429. Springer, Berlin, Heidelberg Available online: https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf

[17] J Mogul. Efficient use of workstations for passive monitoring of local area networks , volume 20. ACM,1990

[18] Jeffrey Mogul, Richard Rashid, and Michael Accetta. The packer filter: an efficient mechanism for user-level network code, volume 21 . ACM,1987.

[19] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In USENIX winter, volume 46 ,1993

[20] Prototype Kernel; eBPF - extended Berkeley Packet Filter. Available online: https://prototype-kernel.readthedocs.io/en/latest/bpf/

[21] Daniel Borkmann. On getting tc classifier fully programmable with cls_bpf. www.netdevconf.org/1.1/proceedings/papers/On-getting-tc-classifier-fully-Programmable-with-cls-bpf.pdf ,2016.

[22] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. Linux Socket Filtering aka Berkeley Packet Filter (BPF). www.kernel.org/doc/Documentation/networking/filter.txt ,2016.

[23] SECure COMPuting with filters. www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt ,2016

[24] Alexei Starovoitov. Tracing: attach eBPF programs to kprobes. lwn.net/Articles/ 636976/ , 2015.

[25] Jonathan Corbet. The kernel connection multiplexer. lwn.net/Articles/657999/ ,2015.

[26] Jonathan Corbet. A JIT for packet filters. lwn.net/Articles/437981/ ,2012

[27] VGER.KERNEL.ORG- The SKB structure. Available online: http://vger.kernel.org/~davem/skb.html