POLITECNICO DI TORINO

Master's degree course in Computer Engineering

Master's Degree Thesis

Network Path Attestation

Supervisors
Prof. Antonio Lioy
Dott. Flavio Ciravegna

Candidate
Alessandro Rosario TORRISI

Academic Year 2024-2025

To my father, my mother and my sister

Summary

In modern networked infrastructures, the lack of visibility and control over packet forwarding paths presents significant security risks, particularly for sensitive traffic that may traverse untrusted or compromised network devices. Current routing protocols provide no native mechanism to enforce path constraints or verify the integrity of intermediate routers, creating vulnerabilities that extend beyond traditional encryption-based protections. This thesis addresses these challenges by implementing a network path attestation framework that enables cryptographic verification of router trustworthiness through hardware-based remote attestation.

The research focuses on Network Path Attestation (NPA) as a foundational component of Trusted Path Routing (TPR), a framework designed to ensure that sensitive traffic traverses only verified, trustworthy network nodes. The implementation leverages Software for Open Netowrking in the Cloud (SONiC) as the network operating system, Trusted Platform Module (TPM) hardware as the cryptographic root of trust, and Keylime as the remote attestation orchestration platform. This combination enables continuous monitoring of network device configurations and real-time integrity verification through standardized attestation protocols.

The technical implementation centres on a custom measurement module integrated into SONiC nodes that systematically captures network configuration state through deterministic normalization processes. The module computes cryptographic hashes that are extended into TPM Platform Configurated Register (PCR) 12, creating an immutable chain of measurements reflecting configuration history. The attestation architecture implements the IETF Remote ATtestation procedureS (RATS) background-check model, where SONiC nodes function as attesters responding to verification requests. The Keylime agent coordinates the attestation workflow by requesting TPM quotes and transmitting evidence packages to the verifier for policy based appraisal.

A significant enhancement to the standard attestation framework is the implementation of semantic network state analysis. Beyond detecting PCR mismatches, the system performs differential analysis to identify specific configuration changes that triggered attestation failures, providing network administrators with actionable intelligence to distinguish between legitimate configuration updates and unauthorized modifications that may indicate security incidents.

The research contributes a practical implementation of hardware-based network attestation that extends trusted computing principles from boot-time verification to continuous runtime monitoring of network infrastructure. By combining TPM cryptographic guarantees with comprehensive network state measurement and semantic change analysis, the system provides verifiable assurance that network devices maintain approved configurations. This work establishes a foundation for trusted path routing architectures where sensitive traffic can be confined to cryptographically verified forwarding infrastructures, advancing network security beyond traditional perimeter-based approaches toward device-level integrity verification.

Contents

Chapter 1

Introduction

1.1 Introduction

In today's global and highly interconnected Internet infrastructure, users and service providers have very limited control over the paths that data packets traverse. Although routing protocols ensure delivery, they provide no guarantees or transparency about the specific intermediate systems involved, nor the jurisdictions they belong to. This becomes a critical concern in scenarios where geopolitical boundaries, regulatory requirements, or privacy policies impose constraints on where data is allowed or forbidden to flow.

For example, a user may wish to prevent their packets from passing through certain countries, or to ensure that only routers within a trusted administrative domain are involved in packet forwarding. However, current network architectures offer no native mechanism to express or enforce such path constraints. The end hosts are largely blind to the actual sequence of Autonomous Systems (ASes) and routers their traffic encounters.

Moreover, there is no assurance regarding the security posture of each router on the path. It remains unknown whether an intermediate node might be compromised, misconfigured, or maliciously re-routing or duplicating traffic toward unauthorised parties. In other words, there is an implicit trust in both the path and the intermediate entities, a trust that is neither verifiable nor enforceable in the traditional Internet model. This lack of assurance becomes even more concerning when considering long-term threats such as Harvest Now, Decrypt Later (HNDL) attacks.

In such scenarios, an adversary potentially operating within a router or network segment may intercept and store encrypted traffic today, with the intent to decrypt it in the future, once advances in cryptanalysis or quantum computing make this feasible. Even if strong encryption is used, the exposure of packets to untrusted intermediaries significantly increases the attack surface, especially in cases where sensitive or time-resistant data is transmitted. The ability to restrict and attest to the path that such data takes, therefore, becomes not only a matter of present-day integrity, but also of future confidentiality. This lack of visibility and control poses serious security, privacy, and compliance risks, especially in sensitive or regulated environments. This is precisely the motivation behind the TPR framework [1].

TPR introduces a technical architecture aimed at giving clients greater transparency and control over the security properties of the network paths their data takes. Specifically, it proposes three foundational capabilities:

- Allow clients to choose the desired security attributes of their received network service. Clients can define a set of security and trust requirements (e.g., geographic locality, certification status, operational behaviour) that must be met by the network infrastructure.
- Achieve dependable forwarding by routing only through devices that satisfy the client's trust requirements. Rather than relying blindly on the default behaviour of routing protocols,

TPR enables policy-compliant routing based on the verified properties of each node in the path.

- Provide cryptographic proof to clients that specific packets or flows have traversed a network path with the claimed trust or security properties. This ensures accountability and enables verification, transforming the implicit trust model into an explicit and verifiable assurance framework.

In this work, we will focus on one specific aspect of the TPR framework: NPA. Our objective is to investigate how to attest the trustworthiness of individual nodes along a network path, with particular emphasis on verifying their configuration status. Rather than addressing the entire end-to-end security model, we concentrate on whether a given router or forwarding device is properly configured and compliant with the desired security policies, thus making it eligible to participate in a trusted path.

To this end, we leverage SONiC (Software for Open Networking in the Cloud) [2] as the underlying network operating system, due to its openness, modularity, and increasing adoption in cloud-scale environments. Furthermore, we employ the TPM as a hardware-based root of trust, enabling cryptographic attestation of critical configuration data such as MAC addresses, IP configurations, and routing tables.

Chapter 2

Trusted Computing

2.1 Introduction

Technology surrounds our lives in ways we barely notice. We book flights in seconds from a smartphone, stream movies through platforms that anticipate our tastes, and adjust the temperature of our homes before even walking through the door. We unlock cars without keys, send money with a fingerprint, and track our health with devices worn on our wrists.

All of this feels natural, and more importantly, it is rarely questioned. Every day, individuals place implicit **trust** in the correct functioning of their devices, assuming that a phone will power on, that a computer will behave as expected, and that the software running these systems will operate in accordance with the manufacturer's design [3].

2.2 Trust and Trustworthiness

The concept of trust, previously introduced in a more informal manner, can be analysed from multiple disciplinary perspectives, including computing, sociology, and psychology. In its official specifications, the Trusted Computing Group (TCG) defines trust as an expectation of behaviour, specifically, the belief that a platform will operate in a way that is consistent with the identity of its hardware and software components [4] However, predictable behaviour alone is not sufficient to establish trust. For a system to be considered trustworthy, its actions must not only be consistent, but also aligned with what is deemed expected and acceptable.

According to the seminal report Trust in Cyberspace by the National Research Council [5], the trustworthiness of a networked information system lies in its ability to function correctly even when subject to environmental disruptions, human errors, or deliberate attacks, while simultaneously preventing unauthorised or unintended actions. These definitions provide the conceptual basis for trust in digital systems.

Yet, in order to translate these principles into concrete security guarantees, a number of architectural mechanisms must be in place. At the heart of this infrastructure is the Trusted Computing Base (TCB), which encompasses all essential components, including hardware, firmware, and software that contribute to enforcing the system's security policy. Even the TCB, however, depends on a foundational element that serves as the initial source of assurance. This is known as the Root of Trust (RoT), a minimal and immutable component that is inherently trusted to perform core security functions. In most modern systems, the role of the RoT is implemented through the TPM, a dedicated hardware module designed to measure and report the integrity of the platform's components.

By enabling secure identification, measurement, and reporting of system state, these mechanisms turn the abstract notion of trust into a set of verifiable and enforceable properties.

2.3 Root of Trust

According to the TCG, the foundational security mechanisms of a computing platform rely on a set of critical system components collectively referred to as RoT. These components must be inherently trusted, as their potential misbehaviour cannot be externally detected or proven [4]. The TCG refers to the RoT as a component because there are different ways it can be implemented; firmware, hardware and software. The TCG specifies a minimal and indispensable set of RoT elements required to support operations that influence a platform's overall trustworthiness. In particular, a trusted platform is expected to implement three fundamental RoT entities:

- RoT for Measurement (RTM)
- RoT for Storage (RTS)
- RoT for Reporting (RTR)

While it is not feasible to ascertain at runtime whether a RoT is operating correctly, it is possible to validate how such components have been implemented. In this context, digital certificates play a pivotal role by providing assurances that a given root has been developed and configured according to the requirements necessary to establish trust.

2.3.1 RoT for Measurement

The RTM is responsible for transmitting integrity-relevant measurements to the RTS. In typical implementations, the RTM corresponds to the central processing unit (CPU), operating under the control of the Core Root of Trust for Measurement (CRTM). The CRTM represents the initial set of instructions executed during the platform's boot sequence. Upon system reset, the CPU begins execution with the CRTM, which then communicates identity-related values to the RTS, thus initiating the measurement chain and establishing the basis for a trusted computing environment.

2.3.2 RoT for Storage

The RTS is often instantiated via the TPM, whose internal memory is isolated from access by any unauthorised entity, including the host system. This architectural isolation ensures that the TPM can reliably act as a secure storage entity.

Not all data stored within the TPM is considered sensitive. For example, the current contents of a PCR containing cryptographic digests may be accessible. Conversely, sensitive information, such as private key material, is stored in Shielded Locations that cannot be accessed without proper authorisation. In certain cases, access to one Shielded Location is conditionally dependent on the contents of another, enhancing the protection hierarchy.

2.3.3 RoT for Reporting

The RTR is responsible for producing attestations that reflect the state of the RTS. Typically, this process involves generating digitally signed digests of selected values within the TPM.

Note: Not all Shielded Locations are reportable. For instance, the TPM does not disclose the private portions of cryptographic keys or certain authorisation values stored within protected memory regions.

The values reported by the RTR commonly include:

- Evidence of platform configuration via PCRs (e.g., `TPM2_Quote()`),
- Audit logs (e.g., `TPM2_GetCommandAuditDigest()`),

- Key properties (e.g., `TPM2_Certify()`).

The integrity of the interaction between the RTR and the RTS is essential. Any implementation must be resistant to tampering that could result in inaccurate or misleading reports. Accordingly, a secure instantiation of both components must:

- Be resilient against all forms of software-based attacks as well as the physical threats outlined in the TPM's Protection Profile,
- Provide an accurate cryptographic digest of all integrity related metrics that have been presented.

2.4 Trusted Computing Base (TCB)

A TCB is the collection of system resources (hardware and software) that is responsible for maintaining the security policy of the system. An important attribute of a TCB is that it be able to prevent itself from being compromised by any hardware or software that is not part of the TCB [4].

2.5 Trusted Platform Module (2.0)

A TPM is a hardware, sometimes software components that provide methods to collect and report schemes that are able to identify hardware and software components, and measure the internal state independent of the host system on which it operates. Communication between the TPM and the host is exclusively mediated through a standardised interface defined by the TCG specification [4]. TPMs are implemented on physical resources, which may be either permanently dedicated to the TPM or temporarily allocated to it. These resources can reside within a single physical boundary or be distributed across multiple boundaries. A typical implementation consists of a single chip component, often integrated into a personal computer, which includes a processor, volatile and non volatile memory (RAM, ROM, and Flash). This form of TPM communicates with the host via a low bandwidth interface, such as the Low Pin Count (LPC) bus. In such configurations, the host system cannot access or alter the TPM's internal memory directly, except through the defined input/output buffer of the interface. An alternative architecture involves implementing the TPM functionality in software, executed directly on the host processor operating in a specialised secure mode. In these cases, specific regions of system memory are isolated by hardware so that they remain inaccessible to the host unless it is operating within the designated mode. Upon mode transition, the processor begins execution at predefined entry points, ensuring control flow integrity. Such software based TPMs retain the core properties of hardware based ones, including isolation and restricted interaction through well defined commands. Secure execution modes that enable this include System Management Mode (SMM), ARM TrustZone™, and various virtualisation-based schemes.

The specification's central goal is to define the interaction model between the host and the TPM. It prescribes a command set through which the TPM performs actions on its internal data structures. One of the principal objectives of these commands is to assess the trust state of the platform. However, the effectiveness and reliability of this process are strictly contingent on the secure and correct implementation of the underlying RoT components.

2.5.1 Architecture

The TCG defines the specifications of the TPM by decomposing its architecture into distinct components, each described in detail to clarify its specific role and functionality within the overall design [4]. The official documentation on TPM implementation serves as a fundamental reference and constitutes the primary resource for designers and manufacturers.

A representation of the complete architecture is provided in Figure 2.1, which illustrates a high level overview of the individual components and their interrelationships. The main elements of the TPM are enumerated and briefly described in the following section.

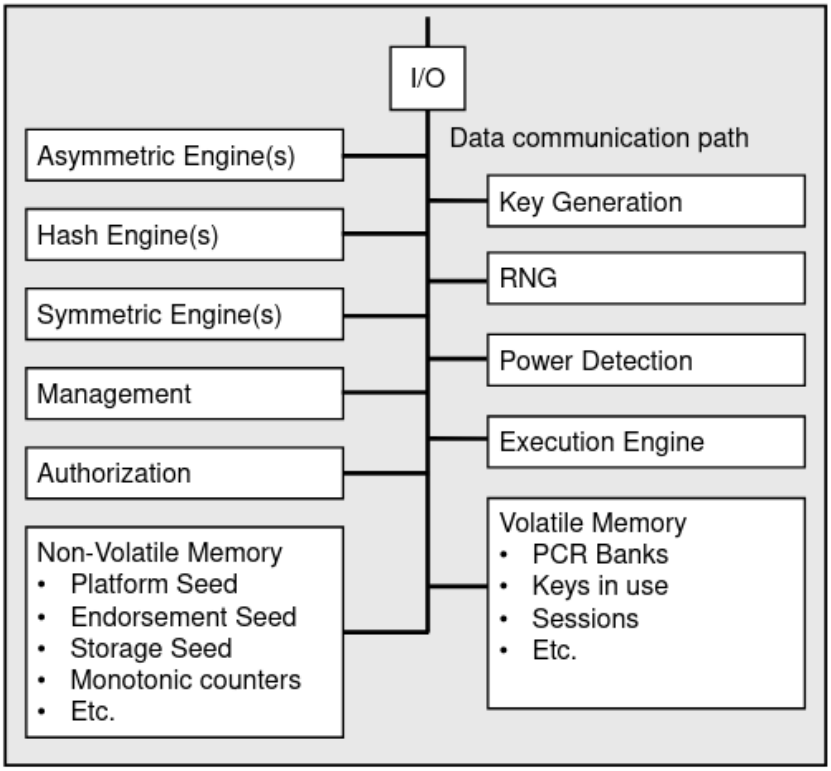

Figure 2.1. TPM architecture (source [4])

- **I/O Buffer**: The communication area between the TPM and the host system. Command data is placed in the buffer by the host and response data is retrieved from it. Although the buffer does not need to be physically isolated, the TPM must ensure that validated command data is protected against modification. Prior to validation, the data must be safeguarded, and before any modification it must reside in a Shielded Location.
- **Cryptography Subsystem**: Implements the cryptographic functions of the TPM. It can be invoked by the Command Parsing module, the Authorisation Subsystem, or the Command Execution module. The subsystem supports a range of conventional operations, including:
  - hash functions,
  - asymmetric encryption and decryption,
  - asymmetric signing and signature verification,
  - symmetric encryption and decryption,
  - symmetric signing (*e.g.*, HMAC and SMAC) and signature verification,
  - key generation.

  Although the TPM has some crypto functionality it is not his only purpose, and so his purpose is not to be as fast as possible, but rather harder to tamper and to be accessed by unauthorised sources, that's why his performance cannot be compared to other components like the Hardware Security Module (HSM)
- **Authorisation Subsystem**: Invoked by the Command Dispatch module at both the beginning and end of command execution. Its role is to verify that valid authorisation has been

provided for accessing each Shielded Location. Depending on the command, some Shielded Locations may require no authorisation, others may require single factor authentication, while more sensitive ones may demand complex authorisation policies. The subsystem relies solely on hash and HMAC functions.

- **Random Number Generator (RNG)**: The RNG, implemented as a Protected Capability without access control, typically consists of an entropy source and collector, a state register, and a mixing function (usually an approved hash function). Entropy is gathered and de biased before updating the state register, which in turn feeds the mixing function to generate random values. A pseudo-random number generator (*PRNG*) may be employed, and when combined with high entropy inputs, it enhances the security of the RNG. The RNG must meet certification requirements relevant to its deployment context and supply sufficient randomness for both internal TPM functions and external requests. In practice, the TPM provides 32 octets per call, with larger requests possibly failing if insufficient entropy is available. Each access returns a fresh random value, with no distinction between internal and external usage.

- **Random Access Memory (RAM)**:. The TPM's RAM is used to store transient data, which may be lost once the device is powered off. For this reason, the TCG specification refers to such data as volatile, although the actual loss behaviour can vary depending on the implementation. When a value has both volatile and non volatile copies, these may reside in a single location, provided that the storage medium supports random access and offers unlimited endurance.

  It is important to note that not all values stored in TPM'RAM are located within Shielded Locations. A specific portion of this memory is reserved for the I/O buffer, which facilitates communication with external components. Beyond this, the TPM's RAM supports three main storage functions, each associated with a dedicated data structure:

  - **Object store:** This contains keys and data loaded from external memory. Typically, an object cannot be used or modified unless it is first placed into TPM RAM, making this component essential for the execution of most operations.
  - **Session store:** Sessions are employed to coordinate and control sequences of TPM operations. They establish authorisation mechanisms and maintain state across multiple commands. Furthermore, sessions can define per command attributes, such as enabling encryption and decryption of parameters or supporting auditing to strengthen security assurance [6].
  - **Platform Configuration Registers (PCRs):** PCRs are specialised registers designed to hold cryptographic hashes representing system states, configurations, and measurements. They are fundamental for attesting to the integrity of the platform, and due to their critical role, they will be analysed in greater detail in a dedicated section.

- **Non Volatile (NV) Memory**: Stores persistent state associated with the TPM. Some NV memory is reserved for use by the platform or entities authorised by the TPM Owner. NV memory contains Shielded Locations, accessible only through Protected Capabilities. Parameters not explicitly defined as persistent may reside either in RAM or NV memory depending on vendor design. To mitigate wear, the TPM should avoid redundant writes by checking if data is unchanged. NV memory may also be used to define persistent data structures (NV Index) or store persistent objects, which can be transparently moved into object slots for efficient access, provided sufficient RAM is available.

- **Power Detection Module**: Manages TPM power states in coordination with platform power states. Platform specific specifications must ensure the TPM is notified of every power state transition. The TPM only recognises ON and OFF states. Any transition requiring a reset of the RTM also resets the TPM, and vice versa. In most cases, the RTM corresponds to the host CPU.

2.5.2 Platform Configuration Register

Platform Configuration Registers (PCRs) are hardware registers within the TPM that represent Shielded Locations, designed to hold measurement log data [4]. A trusted platform is required to keep a record of all events that influence its security posture, particularly during the boot sequence when the Trusted Computing Base (TCB) is being established. Whenever a new entry is added to this log, the TPM obtains either the raw data or its digest, which is then incorporated into a cumulative hash stored inside a PCR. Through this mechanism, the TPM can attest to the PCR value, enabling verification of the integrity of the logged measurements.

The TPM offers two fundamental operations on PCRs:

- **Reset:** The reset operation re initialises a PCR to a predefined value, usually zero. This step is typically performed at the beginning of a boot process to ensure that previous measurements do not interfere with the current measurement cycle.

- **Extend:** The extend operation updates a PCR by concatenating the hash of new data with the current PCR value and hashing the combined result. This produces a cumulative hash that reflects the entire sequence of measurements, ensuring that any modification to the sequence yields a distinct final hash value:

$$\text{PCR}_{\text{new}} = \text{hash}(\text{PCR}_{\text{old}} \,\|\, \text{hash}(\text{new data}))$$

A TPM may support multiple PCR banks, each linked to a specific hash algorithm. These banks facilitate compatibility across different applications that rely on different algorithms. Although banks may differ in the number of registers, PCRs with the same index across banks share identical characteristics, aside from the employed hash function.

PCR values can also serve as conditions for access control within the TPM, denying access to objects when the current PCR values differ from required ones. The value of a PCR can be obtained in several ways: direct reading, attestation inclusion, or integration within a policy.

PCRs may reside in either volatile RAM or non volatile (NV) memory. When stored in NV memory, potential performance impacts must be considered, especially during boot, when numerous measurements are extended. For each supported algorithm, the TPM must provide a corresponding PCR bank, although a bank may be defined with no registers.

The TPM stores varying measurements inside PCRs according to their intended use. The TCG specifications [7] describe how these measurements are employed. Typically, 24 PCRs are defined, though some (indices 17-22) remain unspecified and may be allocated for alternative purposes.

According to the standard shown in Table 2.1 , each PCR is generally tied to a specific set of platform components to be measured. While it is technically possible to use a single PCR for all log entries, this is impractical for monitoring intermediate system states. A clear example of the usefulness of multiple PCRs is during the boot process, where separating measurements ensures that distinct aspects of the platform's configuration and state can be independently validated. Every PCR extension corresponds to an event in the TCG log, which enables external verifiers to reconstruct how the final PCR values were derived.

2.5.3 TPM Keys and Credentials

The Trusted Platform Module (TPM) makes use of a wide range of cryptographic keys in order to perform its internal operations and to support external ones. These keys are primarily **asymmetric** (e.g., RSA, ECC), and may be accompanied by certificates that attest to their validity and bind them to the originating TPM or the platform in which the TPM is deployed.

PCR Index	PCR Usage
0	SRTM, BIOS, Host Platform Extensions, Embedded Option ROMs and PI Drivers
1	Host Platform Configuration
2	UEFI driver and application Code
3	UEFI driver and application Configuration and Data
4	UEFI Boot Manager Code (usually the MBR) and Boot Attempts
5	Boot Manager Code Configuration and Data (for use by the Boot Manager Code) and GPT/Partition Table
6	Host Platform Manufacturer Specific
7	Secure Boot Policy
8–15	Defined for use by the Static OS
16	Debug
23	Application Support

Table 2.1. PCR Index and Usage

Base Terminology

According to TCG defined terminology [8], a *credential* is the combination of:

- a private key generated for a specific purpose, and
- the corresponding `X.509v3` certificate that binds the associated public key to an exact identity.

Here, an **identity** is defined as the pair consisting of the `Subject` and `subjectAltName` fields within a certificate, issued for a particular device by a Certificate Authority (CA). Importantly, identities must be unique within the scope of a given CA.

This terminology is aligned with the IEEE 802.1AR standard [9], which specifies the structure and properties of a device identifier (**DevID**). A DevID is a secure authentication credential, cryptographically bound to a device, and consists of:

1. a private signing key, representing the device's secret and binding the key to the device's identity
2. a corresponding `X.509` certificate containing the public key and a subject name that uniquely identifies the device.

IEEE 802.1AR further distinguishes between two main types of DevIDs:

- **Initial Device Identity (IDevID)**: installed by the Original Equipment Manufacturer (OEM) at the time of device production. Its lifetime is expected to match that of the device, which makes it subject to stricter security restrictions regarding its usage within TPM operations.
- **Local Device Identity (LDevID)**: created and signed by the device owner. These credentials may either replace the IDevID or be used in parallel for different purposes. Typically, LDevIDs have a shorter validity period compared to IDevIDs.

TCG Key Terminology

The management and classification of TPM keys, as well as the enforcement of restriction policies, are closely aligned with the guidelines of IEEE 802.1AR. The initially provisioned keys are inherently secure, since they are protected by the TPM's **Root of Trust for Storage (RTS)**, which prevents unauthorised access and guarantees the binding of the key to the device identity. Additionally, TPM authorisation and policy enforcement mechanisms can be applied to further limit access. Nevertheless, many TPM specific use cases require keys with stricter properties than ordinary signing keys. Such cases involve the use of additional keys and certificates, particularly to support **attestation** and to verify that keys are securely protected within the TPM. The TCG therefore defines several categories of keys [8]:

- **Attestation Key (AK)**: a restricted, non duplicable signing key.
  - *Restricted:* the key can only sign or decrypt TPM generated data.
  - *Non duplicable:* the key is bound to the specific TPM that created it.

  Each AK may be accompanied by an *Attestation Key Certificate*.
- **Signing Key:** a general purpose signing key, often serving as a DevID. Unlike restricted keys, it can be used for broader authentication tasks. Such keys must be paired with a certificate, referred to as a *Signing Certificate*.
- **Storage Key:** a restricted key used exclusively for encryption and decryption. It is typically employed to protect other keys rather than for device identity operations.
- **Endorsement Key (EK):** considered the most critical TPM key, independent of the host platform. The Endorsment Key (EK) is an asymmetric key pair:
  - The **private part** is securely stored inside the TPM and must remain confidential.
  - The **public part** can be accessed and used externally.

  The EK is associated with an **Endorsement Key Credential**, a certificate generally issued by the TPM or platform manufacturer during production. This certificate may contain additional statements about the TPM's origin and security features. While TCG does not mandate specific EK attributes, it is strongly recommended that the EK be configured as a non duplicable, restricted decryption key. This ensures that the EK certificate or public EK does not inadvertently expose privacy sensitive identifiers unique to the platform.

Keys Enrolment and Relationships

Depending on the specific user needs and application requirements, both the platform manufacturer and the platform owner may define additional keys to address those requirements. This process enhances the flexibility of TPM based platforms across diverse use cases while maintaining the security of the overall key certification and management system.

According to the TCG, the generation and provisioning of newly created keys are standardised through well defined enrolment use cases, each characterised by specific requirements and preconditions [8]. At the core of these scenarios lies the establishment of a chain of trust. This chain begins with a consolidated trust anchor always rooted in the TPM as the RoT and is extended through successive binding operations, where each newly created element may itself act as a trust anchor for subsequent components.

To demonstrate that a new key is bound to a particular device, the process starts by binding the TPM to the OEM device. Subsequently, an Attestation Key (AK) is linked to the TPM through the Endorsement Key (EK). The AK certificate then becomes the foundational trust anchor for all subsequent certificates, since it attests that any key generated after the AK resides in the same TPM containing both the EK and the AK. By signing an Initial Attestation Key (IAK) certificate, the OEM Certification Authority (CA) provides the fundamental assurance required for later issuance of Initial Device Identity (IDevID) or Local Device Identity (LDevID)

certificates. An OEM-issued IAK certificate thus explicitly indicates that the IAK is tied to a specific device.

The most commonly adopted enrolment model is illustrated in Figure 2.3. In this model, the Proof of Residency procedure for certifying the Local Attestation Key (LAK) directly leverages the EK and its embedded EK certificate, preinstalled in the TPM by the manufacturer to guarantee a unique association with that hardware. To further strengthen this guarantee, the standard also permits the inclusion of a Platform Certificate issued by the OEM CA, thereby establishing a formal binding between the TPM and the specific platform. Consequently, the Owner CA can reliably issue the LAK certificate by building upon the trust chain anchored in the EK certificate and by validating the LAK's residency within the same TPM.

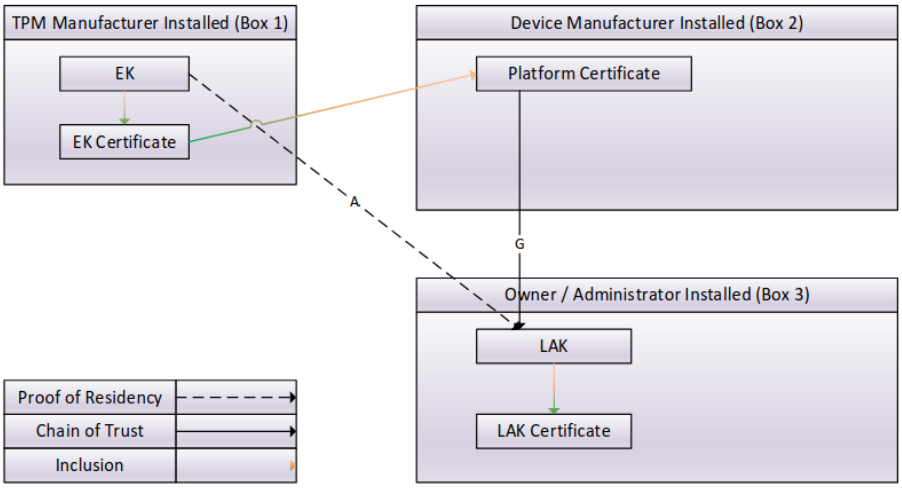

Figure 2.2. Creating LAK Certificate using the Platform Certificate (source [8])

Figure 2.2 depicts the nominal relationships between the different key types. The initial binding associates the IAK with the TPM hosting the reference EK. Once validated by the OEM CA, the IAK functions as the trust anchor for all subsequent keys created and certified by the OEM CA, such as IDevIDs. As the chain evolves, the LAK generated at the user level, must undergo a proof of residency procedure within the same TPM where the IAK resides. Upon successful verification, the LAK assumes analogous functions, enabling the generation of LDevIDs certified by the user CA.

2.5.4 TPM Key Hierarchies

The Trusted Platform Module (TPM) organises its keys into **three hierarchies**, each following a parent-child relationship, where new keys can be derived from parent keys [10].

Hierarchies Overview

1. **Endorsement Hierarchy:** This hierarchy is privacy sensitive, as it is primarily used to generate keys that support device identification and attestation.

2. **Platform Hierarchy:** Managed by the platform manufacturer, this hierarchy is controlled during the early boot process through platform firmware and initialisation code.

3. **Storage ("User") Hierarchy:** Intended for use by the platform owner, such as an enterprise IT department or an end user. This hierarchy is used for operations that do not require strong privacy guarantees, and therefore it is the natural choice for general purpose application keys.

Hierarchy Properties

The selection of a hierarchy directly affects the **authorisation mechanisms** and **policy constraints** that govern the lifecycle of the keys it contains [8].

Each hierarchy is rooted in a **seed**, which is a large random number generated by the TPM and securely kept within the module's protected boundary. The seed provides the cryptographic foundation for the hierarchy and is used to generate **primary objects**, such as the **Storage Root Key (SRK)**.

The Storage Root Key (SRK), acting as the root of the hierarchy, is responsible for encrypting its descendant keys and ensuring the integrity of the key structure [7].

2.5.5 TPM feature summary

The TPM constitutes a highly sophisticated standard, defined through a wide set of specifications that regulate both its structure and its implementation aspects. The extensive documentation produced by the TCG reflects the inherent complexity of the module and the breadth of operations it supports.

The following section provides a concise, high level overview of the principal functionalities of the TPM, which may be of interest to both end users and developers of TPM based solutions. Several of these capabilities are not discussed in detail within the scope of this thesis:

- **Secure key generation and protection**: as extensively discussed in 2.5, the TPM can securely generate and store cryptographic keys, safeguarding them against unauthorised access.
- **Data Binding**: binding enables data to be encrypted with a key internal to the TPM, derived from the Storage Root Key (SRK). This mechanism ensures that the data can only be decrypted by the same TPM that holds the corresponding binding key.
- **Data Sealing**: similar to binding, sealing encrypts data using an internal key of the TPM, but additionally incorporates the platform's state at the time of encryption. Consequently, sealed data can only be decrypted if the current values of the PCRs match those recorded during the sealing process.
- **Platform Integrity Measurement and Reporting**: PCRs support the recording of cryptographic hashes of system states, configurations, and measurements through the extend

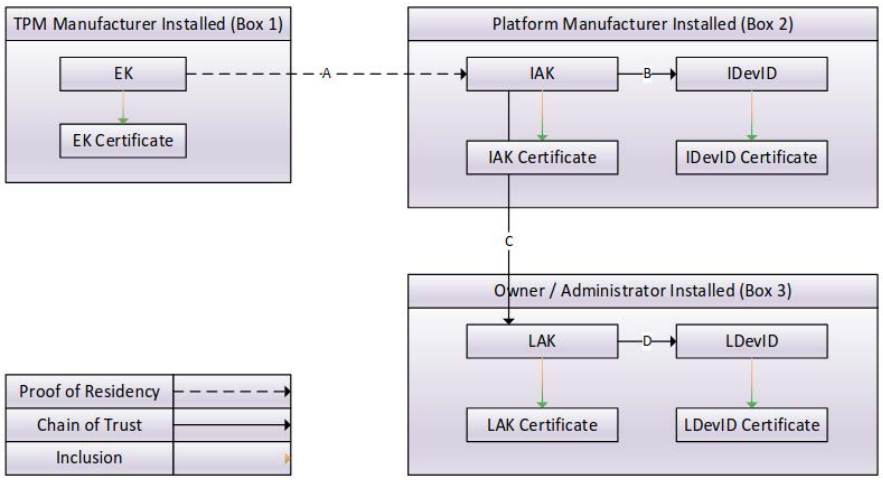

Figure 2.3. Summary of Relationships of IDevID/IAK Keys and Certificates (source [8]

operation. This provides a robust mechanism for monitoring the evolution of the system and for verifying its integrity over time.

- **Device Identification**: each EK and corresponding EK Certificate are unique to the TPM, thereby establishing a reliable trust anchor for identifying both the module and the hosting platform.
- **Privacy and anonymity**: through secure key enrolment procedures, the TPM supports the derivation of Local Device Identifiers (LDevIDs). These identifiers allow device authentication without directly disclosing sensitive information about the platform itself.
- **Remote Attestation (RA)**: the TPM enables the remote verification of a platform's trustworthiness by generating signed attestations of its configuration. Since Remote Attestation (RA) represents a foundational theoretical element of this research, it will be analysed in detail in both in 2.6.

2.6 Remote Attestation

RA is the process of verifying specific properties of a target system, referred to as the *Attester*, by presenting *Evidence* to an external appraiser, known as the *Verifier*, over a network [11]. In TCG terminology, attestation generally denotes the act of having the TPM sign internal data [4]. The assurance of trusted platform properties relies on a hierarchy of attestations, among which RA plays a pivotal role. In essence, a trusted platform can attest to a measurement, thereby confirming that a particular software or firmware state exists. This is achieved by generating a signature over a measurement stored in one or more PCRs, using a certified Attestation Key (AK) securely protected by the TPM. This operation is commonly referred to as a *Quote* [4].

The output of this procedure consists of attested information delivered to the verifier, which validates the received PCR values by comparing them against a set of trusted reference values. Importantly, RA separates the creation of proof from its authentication and validation. This separation allows the verifier to adapt to evolving circumstances; for instance, by rejecting configurations that were previously accepted once new vulnerabilities are discovered.

An attestation architecture, regardless of the specific implementation, must comply with a set of universally recognised principles [11]:

1. **Fresh Information:** The attestation response must capture the dynamic nature of the attester by being generated at the time of the request;
2. **Comprehensive Information:** The attester must provide the verifier with a complete description of its internal state;
3. **Constrained Disclosure:** The attester must authenticate the verifier and enforce policies regulating the disclosure of internal measurements;
4. **Semantic Explicitness:** The identity and properties of the attester should be derivable from the semantic content of the attestation;
5. **Trustworthy Mechanism:** The verifier must receive guarantees about the reliability of the attestation mechanisms themselves. A direct implication is that the attester must be able to deliver correct results even in the presence of corruption.

Logical Execution Flow

The logical execution flow of an attestation process, independent of specific implementation details, typically unfolds as follows:

1. The verifier initiates the process by issuing a challenge request to the attester. This request specifies the PCRs to be validated and includes a nonce to prevent replay attacks by ensuring uniqueness.

2. The attester queries the RTR to read the specified PCRs stored within the RTS, supplying the nonce. In the case of TPM based attestation, access to the RTS is secured via a protected capability. The PCR values are then signed with a certified AK, together with the nonce, and returned to the attester.

3. The attester retrieves the Measurement Log (ML) from the RTM, which contains a chronological record of all operations performed on the target system up to the time of the request.

4. The attester transmits the attestation response to the verifier. This response includes both the signed PCR values from the RTS and the ML.

5. The verifier performs the evaluation:

   (a) it verifies the signature over the measurements, ensuring they originate from the system's authentic RoT;

   (b) it validates the integrity of the ML and replays all PCR extend operations recorded, checking that the replayed results match the received PCR values;

   (c) it analyses the ML entries against its appraisal policy, confirming that only authorised operations were executed.

A figurative representation is present in figure 2.5

IETF Remote ATtestation procedureS (RATS)

RATS [12] are formalised in RFC 9334, which defines the architecture of RA by specifying the *Roles* involved, the interactions among them, and the conceptual messages exchanged.

The RATS framework is intentionally agnostic with respect to processor architectures, the types of *Claims* considered, and the protocols employed.

Within this framework, the *Attester* provides credible information about its state, referred to as *Evidence*, which allows a *Relying Party* to assess its trustworthiness. The Relying Party, in turn, uses this Evidence to make reliable, application specific decisions. A third key participant, the *Verifier*, simplifies the attestation process by evaluating the Evidence against established criteria and generating a verification result that supports the Relying Party's decisions.

One of the main strengths of RATS lies in its flexibility. Depending on implementation choices and infrastructure design, each *Entity* may assume one or more of the defined Roles. This role allocation enables RATS to adapt to heterogeneous systems and device architectures while maintaining a coherent attestation model.

The *Verifier* represents to some extent the central point where most of the messages flow, since exchanging information directly supports attestation evaluation in an RA architecture. Among the entities that interact with the Verifier, two are particularly relevant: the *Endorser* and the *Reference Value Provider*. The former helps the Verifier validate the received *Evidence*, by defining further properties of the *Attester*, typically provided by a manufacturer. The latter supplies the Verifier with a set of reference values, which are compared with those contained in the Claims received as part of the Evidence.

In RATS terminology, a *Claim* is a name value pair that indicates a type of information and its corresponding value. A set of Claims defines the Evidence. RATS further specifies that an *Attester* consists of at least one *Attesting Environment* and one *Target Environment*, both residing in the same domain. The Attesting Environment is an execution environment capable of collecting Claims from the Target Environment and formatting them appropriately. Claims are derived from measurements of relevant assets of the Target Environment, such as system registers, variables, memory, and executed code.

It is important to note that the *TPM* alone does not qualify as an Attesting Environment, since it lacks the active component required to autonomously collect and store measurements within its shielded locations. However, when combined with a *Root of Trust for Measurement (RTM)*, the TPM forms a complete Attesting Environment.

RATS attestation relies heavily on evaluation against *Appraisal Policies*. An Appraisal Policy consists of a series of predefined operations executed sequentially to assess whether an incoming message aligns with specified standards and correctness criteria. Different entities define and enforce Appraisal Policies through various methods, each tailored to handle specific types of messages.

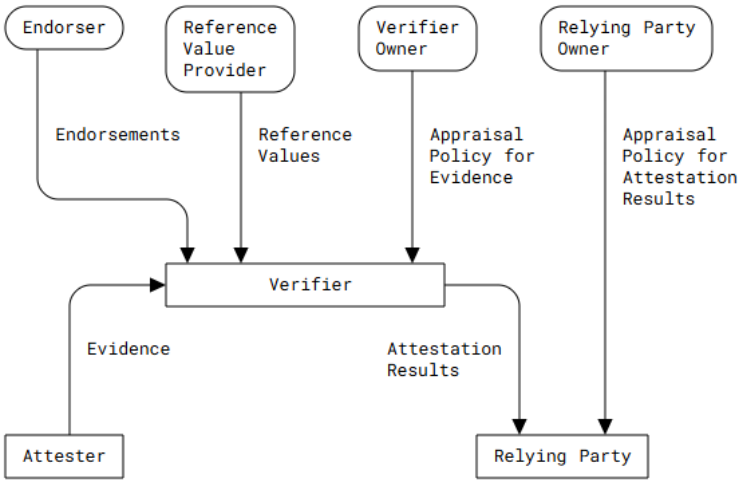

Figure 2.4. RATS architecture

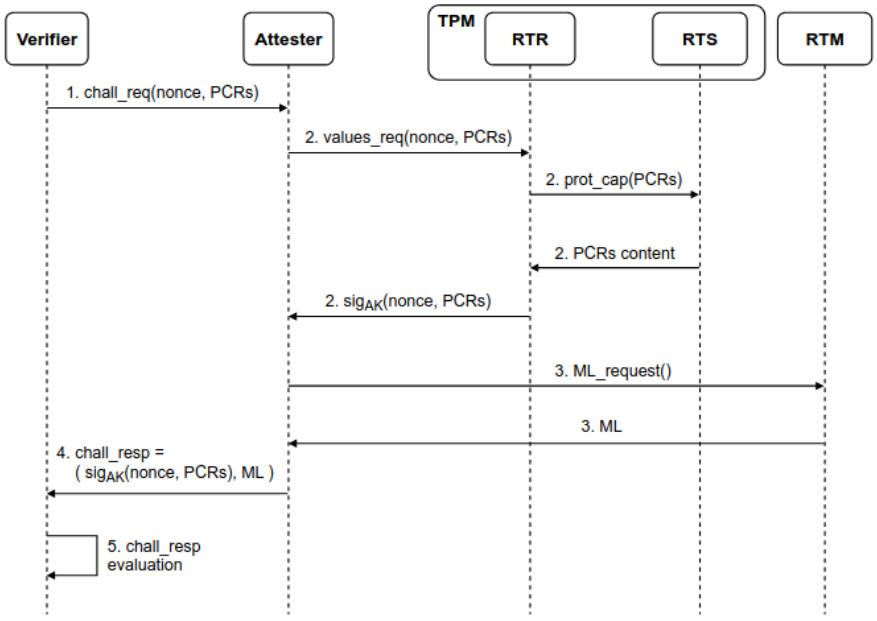

Figure 2.5. RA workflow

RATS defines two main interaction patterns between the Attester, the Verifier, and the Relying Party, which can also be combined to generate more complex variants:

- **Passport Model:** the Attester sends Evidence to a Verifier, which evaluates it against its Appraisal Policy. The Verifier then returns an *Attestation Result*, treated as opaque data by the Attester. The Attester may cache this result and later present it (possibly with additional Claims) to a Relying Party, which compares it against its own Appraisal Policy. The same Attestation Result may be reused with multiple Relying Parties.

- **Background-check Model:** the Attester transmits Evidence directly to a Relying Party, which treats it as opaque and forwards it to a Verifier. The Verifier checks the Evidence against its Appraisal Policy and returns an Attestation Result to the Relying Party, which then evaluates it against its own Appraisal Policy.

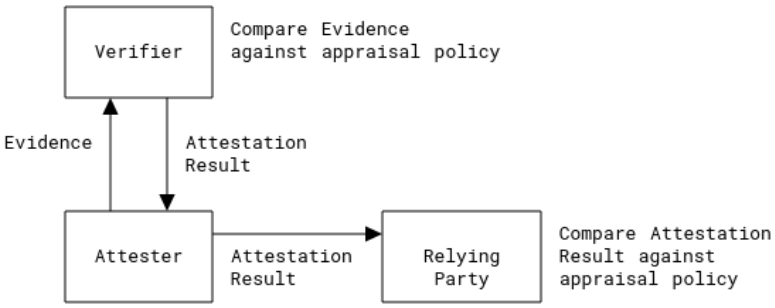

Figure 2.6. Passport model (source [12])

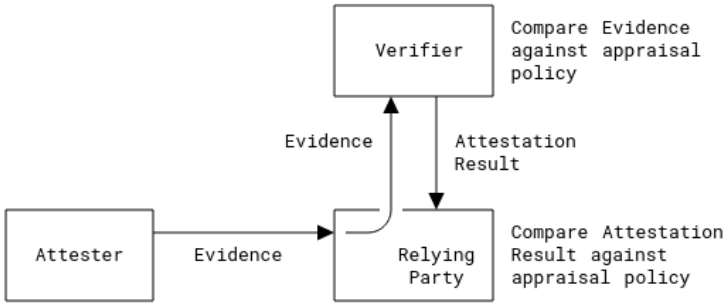

Figure 2.7. Background-check model (source [12])

2.6.1 Keylime: A Practical Implementation

Keylime is an open-source remote attestation framework that provides a concrete implementation of the IETF RATS architecture [13]. Initially developed at MIT Lincoln Laboratory and subsequently adopted as a Cloud Native Computing Foundation (CNCF) project, Keylime enables scalable and automated integrity verification of remote systems through TPM-based attestation.

The framework implements the background-check model defined in RATS, distributing attestation responsibilities across specialised components that correspond to the roles specified in RFC 9334. By leveraging TPM 2.0 capabilities and implementing continuous monitoring mechanisms, Keylime extends traditional attestation beyond boot-time verification to provide runtime integrity assurance.

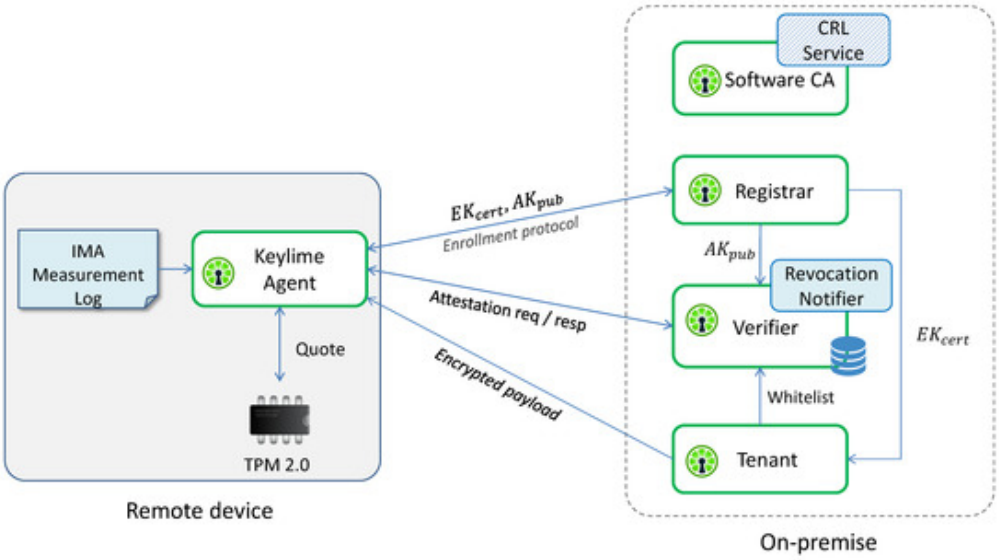

Figure 2.8. Keylime architecture and component interaction [13]

Architecture and Components

Keylime's architecture consists of four primary components, each implementing specific RATS roles and responsibilities (Figure 2.8):

Agent The Keylime Agent operates on the system being attested and implements the RATS *Attester* role. It interfaces directly with the TPM to collect measurements stored in PCRs and retrieves the measurement log. The Agent exposes a REST API through which it receives attestation requests and responds with Evidence consisting of TPM quotes, PCR values, and the measurement log.

The Agent maintains minimal trust assumptions and operates with restricted privileges. It does not perform policy decisions or evaluation; its sole responsibility is to accurately collect and transmit integrity measurements from the TPM. The Agent also manages the measured boot process and can be configured to monitor specific files or directories for runtime integrity verification.

Verifier The Verifier implements the RATS *Verifier* role and constitutes the core attestation engine. It continuously polls registered Agents at configurable intervals, requesting fresh TPM quotes and measurement logs. Upon receiving Evidence from an Agent, the Verifier performs multiple validation operations:

1. It verifies the cryptographic signature on the TPM quote using the AIK certificate, confirming that the measurements originate from an authentic TPM.
2. It validates the measurement log integrity by replaying all extend operations and comparing the computed PCR values against those included in the signed quote.
3. It evaluates the measurement log entries against its Appraisal Policy, which specifies the set of authorised boot components and their expected hash values.
4. It assesses any runtime integrity measurements against configured allowlists.

The Verifier maintains a database of Agent states and generates Attestation Results indicating whether each Agent satisfies the configured policy. When an Agent fails attestation, the Verifier can trigger automated responses, including notification systems or revocation of cryptographic keys.

Registrar The Registrar implements identity management and initial trust establishment for Agents. During Agent enrolment, the Registrar validates the TPM's Endorsement Key (EK) certificate against manufacturer certificates and establishes the authenticity of the AIK through either direct verification or challenge-response protocols.

The Registrar maintains a registry of authorised Agents and their associated TPM credentials. This component implements the trust anchoring mechanism by verifying that Agents possess legitimate TPMs before they are permitted to participate in attestation. The Registrar also stores encrypted AIK credentials that Agents retrieve after successful enrolment.

Tenant The Tenant provides the administrative interface through which operators register Agents with the Verifier and define attestation policies. It implements a command-line tool and REST API for policy configuration, Agent management, and attestation status monitoring.

Through the Tenant, administrators specify the Appraisal Policy that defines acceptable system states. This includes reference measurements for bootloader, kernel, and other critical boot components, as well as runtime monitoring policies. The Tenant also configures the cryptographic material and revocation actions associated with attestation failures.

Attestation Workflow

The attestation process in Keylime follows the IETF RATS background-check model and proceeds through several phases:

Enrolment Phase The Agent first registers with the Registrar by presenting its EK certificate and a generated AIK. The Registrar verifies the EK certificate chain against known manufacturer root certificates. Upon successful validation, the Registrar stores the Agent's TPM credentials and provides an encrypted response that only the authentic TPM can decrypt, establishing initial trust.

Subsequently, the Tenant registers the Agent with the Verifier, providing the attestation policy and any cryptographic material to be protected. The Verifier stores this configuration and begins periodic attestation of the Agent.

Continuous Attestation The Verifier initiates attestation by sending a challenge containing a fresh nonce to the Agent via its REST API. The Agent responds by requesting a TPM quote over specified PCRs using the nonce. The resulting Evidence package includes:

- The TPM-signed quote containing PCR values and the nonce
- The complete measurement log (typically the IMA log)
- Runtime integrity measurements if configured
- The AIK certificate used for signing

The Verifier receives this Evidence and executes its Appraisal Policy against the measurements. It reconstructs the PCR values by replaying the measurement log and compares them with the signed values. Each boot component hash in the log is verified against the reference values in the allowlist. Any deviation results in attestation failure.

Response to Attestation Failure When an Agent fails attestation, the Verifier updates its internal state and can trigger configured responses. These may include revoking cryptographic keys stored on the Agent, executing notification scripts, or updating external policy enforcement points. The continuous nature of attestation ensures that compromises are detected within the configured polling interval.

Role in Attestation

Keylime provides several functions that advance practical remote attestation deployment:

Scalability: Unlike manual attestation procedures, Keylime automates the entire attestation lifecycle. The Verifier can manage thousands of Agents simultaneously, performing continuous integrity verification without operator intervention.

Standards Compliance: By implementing the IETF RATS architecture, Keylime provides interoperability with other RATS-compliant systems and supports standardised Evidence formats and evaluation procedures.

Measured Boot Integration: Keylime leverages the Linux IMA subsystem to extend attestation beyond boot-time measurements to runtime file integrity. This enables detection of compromises that occur after system initialisation.

Key Revocation: The framework implements cryptographic key escrow, where sensitive keys are released to Agents only after successful attestation. Upon detecting integrity violations, the Verifier automatically revokes these keys, preventing compromised systems from accessing protected resources.

Policy Flexibility: Administrators define attestation policies that specify acceptable system states without modifying framework code. Policies can accommodate software updates by including multiple valid hash values for components that change between versions.

Chapter 3

Trusted Path Routing

Modern networks increasingly transport highly sensitive traffic flows that demand assurances extending beyond encryption mechanisms alone. Some end users regard technologies such as IPSec as insufficient to guarantee the confidentiality of their most critical data [14]. These users require that their traffic traverse exclusively devices that have been recently appraised and verified as trustworthy. TPR emerges as a forwarding paradigm explicitly designed to address this requirement, aligning with the IETF RATS architecture [12]. Its goal is to ensure that traffic associated with designated *Sensitive Subnets* is routed only through nodes that possess current and valid *Trustworthiness Vectors* issued by a Verifier.

The foundation of TPR is the dynamic construction of a *Trusted Topology*: a subnetwork composed of *Attested Devices* (routers or switches that have been successfully appraised) and, where applicable, *Transparently-Transited Devices* that do not process packets beyond L2/L3. Within this topology, neighboring routers exchange cryptographically protected artefacts, called *Stamped Passports*. These passports encapsulate fresh evidence derived from TPM-based RA, including details on hardware identity, boot state, firmware versions, and the outcomes of prior appraisals. Each router evaluates the passport of its peer against a defined security policy and either accepts or rejects the adjacency for inclusion in the Trusted Topology.

Once trusted adjacencies are established, standard IGPs such as IS-IS or OSPF propagate them using existing mechanisms (e.g., flex-algorithm, affinity attributes). As a result, packets destined to or originating from Sensitive Subnets are forwarded strictly along paths formed by compliant nodes, while links failing to meet the defined trustworthiness criteria are automatically pruned from the routing fabric. This mechanism enables confidentiality-sensitive flows to benefit not only from encryption but also from verifiable guarantees on the integrity of the devices that forward them.

3.1 Background and Terminology

A comprehensive understanding of TPR requires familiarity with remote attestation principles and related trust mechanisms. The IETF RATS framework [12] specifies how a device (the **Attester**) can prove its operational state to a peer (the **Relying Party**), assisted by an independent **Verifier**. Within the TPR context, each router qualifies as an **Attested Device** once it has successfully undergone a recent verification cycle. Formally, an Attested Device is defined as a router whose most recent attestation by a Verifier yields a **Trustworthiness Vector**. This vector encodes the assessment of individual trustworthiness claims, which may be affirmed, flagged with warnings, or marked as contraindicated. Examples of such claims include:

- bootloader integrity is intact
- firmware version is current
- running configuration matches the approved baseline

3.1.1 Core Roles in Trusted Path Routing

Attested Device

An Attested Device is any router or switch that has undergone remote attestation and has received a positive appraisal from a Verifier. Its current state—including hardware identity, boot sequence, firmware version, running configuration, and relevant executables—is summarized in a Trustworthiness Vector. Only devices with valid and timely attestations are eligible to participate in a Trusted Topology. In the implementation proposed in this work, the **SONiC** node represents the Attested Device, extending its measurements into the TPM and sending them to the Verifier upon request.

Sensitive Subnets

Sensitive Subnets are IP prefixes associated with traffic flows that require enhanced confidentiality guarantees. Policies defined at the edge of the network specify which subnets are considered sensitive, thus triggering the enforcement of TPR. Traffic to or from non-sensitive subnets continues to follow ordinary forwarding policies.

Example of a Sensitive Subnet

A clear example is provided by the financial sector. Consider a banking institution where the subnet `10.42.0.0/24` hosts the servers responsible for core operations such as transaction processing and settlement records. Although all communications are protected by cryptographic protocols like TLS or IPsec, the bank requires additional guarantees. Specifically, it mandates that all traffic to and from this subnet must traverse only devices that have been recently appraised as trustworthy. The rationale is that, beyond payload confidentiality, metadata leakage (for example, communication patterns, volume, or timing) could still expose critical information.

By classifying `10.42.0.0/24` as a Sensitive Subnet, the operator ensures that routing protocols enforce forwarding exclusively through the Trusted Topology. As a result, packets destined for or originating from this subnet are confined to paths formed solely by Attested Devices and, where applicable, Transparently-Transited Devices, in full alignment with the principles of TPR. an examp

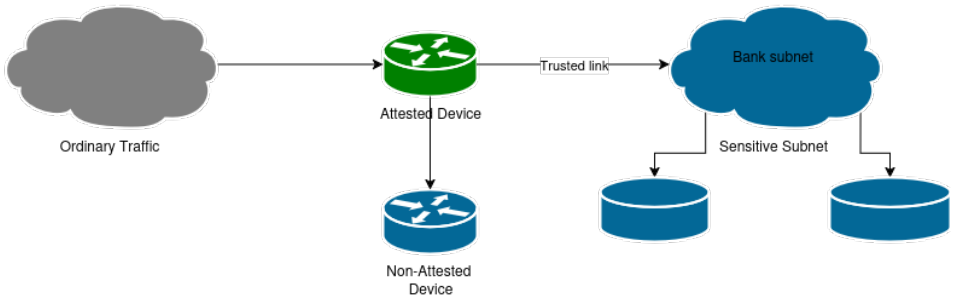

Figure 3.1. Sensitive subent example

Trustworthiness Claims and Trustworthiness Vector

A **Trustworthiness Claim** is a specific assertion about a property or characteristic of an Attested Device. These claims are derived from evidence produced by the device, typically via a TPM, and evaluated by a Verifier. Each claim represents a distinct dimension of trust, allowing operators to express security requirements in a fine-grained manner.

Typical claims may include:

- **Hardware Identity**: confirmation that the device possesses a valid endorsement credential issued by the manufacturer.
- **Boot Process Integrity**: evidence that the bootloader and kernel have not been tampered with during startup.
- **Firmware and Software Versions**: verification that the versions running on the device are within an approved allow-list.
- **Configuration Consistency**: assurance that the current running configuration matches the baseline defined by the operator.
- **Operational State**: confirmation that no prohibited services are enabled and that expected security functions (e.g., ACLs, encryption) are active.

The Verifier processes all available claims and produces a **Trustworthiness Vector**. This vector is essentially a structured report that aggregates the evaluation of multiple claims and associates each with an appraisal result. For consistency, each claim can be assigned one of three appraisal outcomes:

- **Affirming**: the claim is fully satisfied and compliant with the defined policy.
- **Warning**: the claim is not ideal but still tolerable (for instance, a firmware version is slightly outdated but not known to be vulnerable).
- **Contra-indicated**: the claim directly violates policy or indicates potential compromise (e.g., a bootloader hash mismatch).

Concrete Example Consider a router that has just completed an attestation cycle. The evidence is appraised by the Verifier, which generates the following Trustworthiness Vector:

- Hardware identity: **affirming** — manufacturer-issued endorsement key verified.
- Bootloader integrity: **affirming** - PCR values match the expected baseline.
- Firmware version: **warning** — version is one release behind the current allow-list but no critical vulnerabilities are registered.
- Configuration consistency: **affirming** — running configuration hash matches the baseline.
- Operational state: **contra-indicated** — an unauthorized routing daemon is active.

In this case, even though most claims are affirming, the contra indicated appraisal for the operational state prevents the device from joining the Trusted Topology. This example illustrates how the Trustworthiness Vector provides a nuanced, multi dimensional view of a device's state, enabling policy engines to make binary decisions (admit/exclude) based on a structured and auditable set of criteria.

Stamped Passports

A **Stamped Passport** is a cryptographically verifiable artefact exchanged between neighboring devices. It encapsulates the results of a recent appraisal together with guarantees of freshness, such as a nonce or epoch counter, and is digitally signed by the device's TPM. By design, Stamped Passports are exchanged at the link layer and enable peers to verify each other's trustworthiness without the need to query the Verifier for every packet or flow. In practice, they operate as short-lived, signed proofs that a device has been recently appraised and approved to participate in the Trusted Topology.

A Stamped Passport may contain:

- **Device Identity**: derived from the endorsement key or attestation key associated with the TPM.
- **Trustworthiness Vector**: the set of evaluated claims and their appraisal results as issued by the Verifier.
- **Freshness Information**: a nonce, counter, or timestamp to prevent replay of stale attestations.
- **Digital Signature**: computed with the TPM-protected attestation key, binding the above fields into an unforgeable artefact.

Operational Role When two routers attempt to establish an adjacency, each one presents its Stamped Passport to the other. The peer verifies the signature and checks the embedded Trustworthiness Vector against its local policy. If all required claims are affirming and no contraindicated values are present, the adjacency is accepted into the Trusted Topology; otherwise, it is rejected. This mechanism ensures that only devices with a valid and recent security appraisal can become part of the forwarding path for sensitive traffic.

Concrete Example Consider two routers, R_1 and R_2, on the same link. Before admitting R_2 into its Trusted Topology, R_1 requests a Stamped Passport. R_2 responds with a signed artefact containing:

- Hardware identity: **affirming**
- Firmware version: **affirming**
- Configuration baseline: **warning** (slight deviation, tolerated by policy)
- Freshness counter: value incremented within the last appraisal window

After validating the signature and verifying that the warning is acceptable under its local policy, R_1 includes the link to R_2 in its Trusted Topology. If instead the Passport had contained a contraindicated claim (e.g., unverified bootloader), the link would have been excluded automatically.

Trusted Topology

The **Trusted Topology** represents the logical sub-graph of the network that is formed exclusively by Attested Devices (and, where applicable, by Transparently-Transited Devices that have been mutually accepted through the exchange of **Stamped Passports**. Only those links where both endpoints can prove compliance with the required trustworthiness claims are retained. Routing protocols such as IS-IS or OSPF then propagate information exclusively about these trusted adjacencies, allowing the control plane to construct forwarding paths that respect the required trust policy.

Key properties of a Trusted Topology include:

- **Dynamic Assembly**: links are admitted or pruned depending on the most recent attestation results.
- **Policy Enforcement**: only devices whose Trustworthiness Vectors satisfy the defined requirements can participate.
- **Isolation of Sensitive Flows**: traffic to and from Sensitive Subnets is constrained to remain within this sub-graph, never traversing devices of unknown or unverified trust status.

Concrete Example in the Financial Sector Consider a multinational bank operating several data centers across different jurisdictions. The subnet `10.42.0.0/24`, dedicated to transaction processing, is designated as a Sensitive Subnet. The bank requires that all traffic entering or leaving this subnet remain within a Trusted Topology composed exclusively of routers that have been recently appraised.

In practice, this means that only the routers in the bank's European facilities, which have successfully provided valid Stamped Passports, are included in the Trusted Topology for this subnet. Routers in other regions, although reachable at the IP level, are excluded because their firmware is outdated or their appraisal has expired. When a client in Milan initiates a banking transaction, the routing protocol ensures that packets are forwarded only along trusted routers in Milan, Zurich, and Frankfurt, bypassing untrusted devices in other parts of the backbone.

As a result, the bank can guarantee that sensitive financial data never traverses equipment outside the appraised domain, reducing exposure to compromised routers, regulatory conflicts, or metadata leakage. This example highlights how the Trusted Topology transforms an abstract security policy into an enforceable network construct that protects both present-day integrity and long-term confidentiality of sensitive flows.

Table 3.1 summarises the key terminology adopted in TPR.

Table 3.1. Key terminology in Trusted Path Routing

Attested Device	A router that has passed TPM-based attestation and possesses a valid Trustworthiness Vector issued by a Verifier.
Stamped Passport	Composite attestation evidence exchanged at link establishment, comprising Verifier-signed results and a fresh TPM quote.
Trustworthiness Claim/Vector	A statement regarding a device property (e.g., "hardware identity verified", "firmware patched") and the vector of such claims produced by a Verifier.
Sensitive Subnet	An IP prefix explicitly marked by an operator as requiring enhanced protection, to be routed only via trusted paths.
Trusted Topology	The network subgraph formed exclusively by Attested Devices (and optional transparent transit devices) that satisfy the defined trust policy.

Having established these definitions, the next section delves into the interaction of these components in order to Build a Trusted Topology.

3.2 Stamped Passport Attestation Workflow

The establishment of a *Trusted Topology* is not the result of a single check, but rather of a multi-stage workflow that combines device attestation, link-layer authentication, and continuous re-validation. Central to this process is the notion of the *Stamped Passport*, a composite object that merges appraisals from a remote Verifier with fresh evidence generated by the Attester itself. The sequence can be summarized as follows:

- **Step 1 – Evidence Generation.** Each Attester (e.g., a router or switch with a TPM) generates cryptographically signed Evidence. This includes a TPM quote over selected PCRs, where a nonce provided by Verifier-A is bound into the quote to guarantee freshness. This ensures that the measurements reflect the actual device state at that specific moment and cannot be replayed.
- **Step 2 – Attestation Results.** Verifier-A receives the Evidence and performs an appraisal. The outcome is a signed set of *Attestation Results*, composed of:
  1. a *Trustworthiness Vector* with claims such as hardware state, device identity, and executables loaded;

2. the public attestation key that verified the TPM quote;
3. the PCR values and timing counters from the TPM;
4. a signature by Verifier-A over the whole package.

These results are pushed back to the Attester and stored locally for later use.

- **Step 3 – Link Freshness.** When a neighboring router (the *Relying Party*) initiates link-layer authentication, for instance via IEEE-802.1X or MACsec, it provides fresh randomness (a nonce or epoch handle). This creates a new point of freshness specifically tied to the link establishment.
- **Step 4 – Stamped Passport Assembly.** In response, the Attester generates a new TPM quote that incorporates the freshness token from Step-3. It then assembles the *Stamped Passport*, which contains both the latest Attestation Results from Verifier-A and the fresh TPM quote signed by the TPM's Attestation Key. This combination allows the neighbor to validate not only the device's last appraisal by Verifier-A but also the current and immediate state of the device.
- **Step 5 – Relying Party Appraisal.** The Relying Party receives the Stamped Passport and verifies:

  1. that the freshness token from Step-3 is correctly bound;
  2. that Verifier-A's signature is valid and trusted;
  3. that the PCR values and state information match between the Attestation Results and the fresh quote;
  4. that the fresh TPM quote is signed by the same attestation key previously validated by Verifier-A.

  If any of these checks fail, the link is assigned a null Trustworthiness Vector and excluded from the Trusted Topology. If successful, the Relying Party derives its own Trustworthiness Vector (Verifier-B view) and accepts the adjacency.
- **Step 6 – Topology Update.** Finally, the decision is reflected in the routing protocol (e.g., ISIS with Flex-Algo [15]). Links validated with a sufficient Trustworthiness Vector are included in the trusted IGP graph, while links with insufficient guarantees are avoided for sensitive traffic.

The resulting workflow provides two critical assurances. First, the device is appraised independently by a trusted Verifier-A, which anchors the Trustworthiness Vector in a remote assessment. Second, the link-level authentication enforces a fresh attestation each time the connection is established or re-authenticated. Together, these mechanisms ensure that only links whose state has been recently validated are eligible to carry Sensitive Subnet traffic, thereby maintaining the integrity of the Trusted Topology across the network.

3.3 Limitations of TPR

One of the main constraints of TPR lies in its strong dependence on intra-domain routing protocols. The IETF proposal defines a procedure where neighbouring routers perform mutual remote attestation to establish whether a given link can be considered trusted. Once these pairwise relationships are established, the routing protocol—for example *Intermediate System to Intermediate System* (IS-IS) or *Open Shortest Path First* (OSPF)—is used to construct a forwarding plane composed exclusively of trusted adjacencies. In this way, a "trusted sub-topology" emerges, where packets are only forwarded through devices that have successfully provided valid attestation evidence.

Although this mechanism provides a clear security benefit, it also exposes fundamental limitations. Since the trust establishment is designed around the semantics of intra-domain protocols,

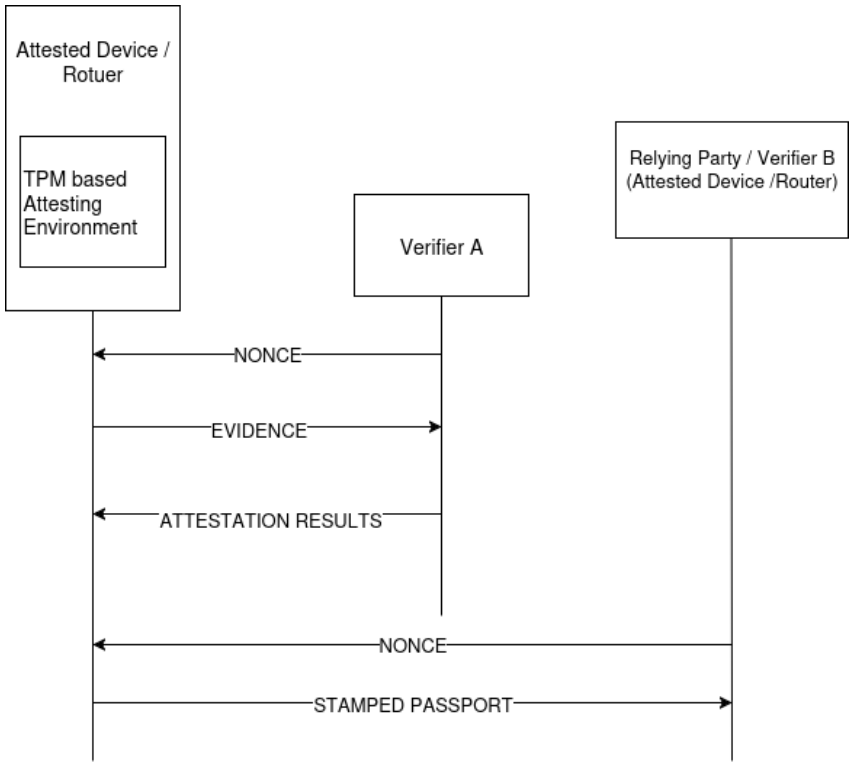

Figure 3.2. Message exchange and timing in the Trusted Path workflow.

the resulting trusted path remains confined to the boundaries of a single administrative domain. In practice, this means that the assurance guarantees offered by TPR are restricted to the scope of one network operator or autonomous system. End-to-end paths that traverse multiple domains, which are the norm on the public Internet, cannot be directly protected by the current TPR approach.

This restriction creates a gap between the original motivation of trusted path routing, to guarantee that sensitive traffic only crosses devices with verifiable integrity and the actual deployment reality of the Internet. While an enterprise or service provider may achieve strong intra-domain assurance, once traffic exits that administrative perimeter the trust model collapses back to traditional inter-domain mechanisms such as BGP, which lack attestation semantics. As a consequence, the security guarantees of TPR today are necessarily partial, tied to local enforcement, and cannot provide users with flexible, policy-compliant control over their paths at a global Internet scale.

3.4 Security Issues

In designing and deploying TPR, it is crucial to acknowledge both the limitations of the appraisal process and the residual risks that persist even in the presence of robust attestation frameworks. The following points summarise the most relevant challenges [1].

- **Constraints on Evidence Collection.** Verifiers are inherently limited by the scope of Evidence that an Attester, such as a router, can provide. Although the state of the art in network attestation is advancing, certain classes of exploits remain undetectable. Attacks leveraging transient states, ephemeral runtime behaviour, or memory-resident malicious payloads may not be captured by static configuration checks or cryptographic measurements. This asymmetry implies that, while Verifiers can provide strong assurance regarding a device's measured configuration and integrity, they cannot claim full visibility over all possible malicious states. Complementary mechanisms, such as runtime anomaly detection or cross-layer correlation, remain necessary.

- **Role of TPM-Anchored Measurements.** Only measurements extended into PCRs can be exposed through a TPM Quote at appraisal time. This creates a hard boundary: any configuration element or operational state not measured and extended is invisible to the Verifier. The decision of what to measure is therefore strategic. Limiting measurements to firmware, kernel images, or configuration files protects against low-level tampering but risks overlooking higher-layer manipulations, such as routing policy injection. Expanding measurements to include dynamic artefacts, like routing tables or `FRR/BGP` states increases assurance but raises challenges in performance, freshness, and reference value management.
- **Implications of Compromised Verifiers.** The trust model of TPR assumes Verifiers themselves remain uncompromised. A successful attack on a Verifier poses systemic risk: false validation of compromised routers would admit malicious nodes into the Trusted Topology, affecting confidentiality, integrity, and availability of flows. To mitigate this risk, architectural safeguards are required, including:
  - Redundancy across multiple independent Verifiers to reduce single points of failure.
  - Distributed consensus before establishing trusted adjacencies.
  - Hardware-protected enclaves to shield decision logic from tampering.
- **Visibility in Link-State Routing Domains.** In link-state routing protocols supporting FlexAlgo, links in the trusted overlay are visible across the entire IGP domain. A compromised router, even if excluded, can infer when it is being bypassed. Such awareness may drive attackers to adapt their strategy, for example by launching denial-of-service attacks to force rerouting, or by compromising additional nodes to regain a position in critical forwarding paths. This shows that TPR significantly reduces the attack surface but cannot entirely eliminate adversarial influence at the network level.

And addition to the work of TPR not introduced in this work is *Proof of Transit* (PoT), which aims to build a verifiable proof of whether a packet has traversed a specific set of nodes [16]. The idea relies on shared-secret cryptographic schemes, such as Shamir's Secret Sharing, or other more advanced algorithms that allow a secret to be divided into n shares. Only by collecting all of these shares can the original secret be reconstructed using the scheme's reconstruction algorithm. Importantly, each individual share reveals no information about the so-called master secret.

In simplified form, the steps of the protocol are:

- A controller node generates a master secret and divides it into shares.
- The shares are distributed to the set of nodes along the intended path.
- As the packet traverses each node, that node contributes its share to the cumulative value carried by the packet.
- Once the packet reaches its destination, the accumulated value is compared with the original master secret. If they match, the verifier gains assurance that the packet indeed passed through all the designated nodes.

A practical implementation of Ordered Proof of Transit (OPoT) has been demonstrated in the context of Service Function Chaining (SFC). The goal is to ensure that packets traverse a prescribed sequence of Virtual Network Functions (VNFs), such as *firewall* $\rightarrow$ *IDS* $\rightarrow$ *NAT*, without deviations. To achieve this, researchers have shown that OPoT can be embedded directly into the Linux kernel using extended Berkeley Packet Filter (eBPF) programs [17]. Running the logic in kernel space allows verification to be performed with minimal overhead and without requiring modifications to each Virtual Network Function (VNF), which is particularly important for *SFC-unaware* functions. In this design, packets carry a cumulative value updated at every hop, which is checked at the egress node to confirm correct traversal. The evaluation highlights that path violations can be detected effectively, while the additional cost of eBPF execution remains limited.

3.5 Related Works

3.5.1 SCION

The Scalability, Control, and Isolation On Next-generation networks (SCION) architecture [18] represents a foundational influence on secure forwarding and shares conceptual similarities with TPR. SCION was designed to overcome the structural vulnerabilities of the current Internet by offering path transparency, isolation between administrative domains, and resilience against routing attacks such as BGP hijacking or path manipulation; with this architecture Autonomous systems (AS) are now able to decide a path on which to send their packet. Its design revolves around three principles. First, end hosts are empowered to select explicit communication paths from a set of cryptographically verifiable options, rather than relying on opaque routing choices made by intermediate autonomous systems. These paths are built from **pre computed** segments distributed by the control plane, which enables endpoints to base their routing on trust preferences and performance criteria. Second, the network is organized into hierarchical *Isolation Domains* that group autonomous systems under shared trust and policy. This allows traffic to be confined to domains explicitly trusted by both endpoints, reducing the risk of unauthorized transit. Third, every hop on a SCION path is secured with cryptographic authenticators known as hop fields, which protect forwarding information against tampering and ensure that only the intended next hop can process it.

Although SCION operates at the inter-domain level, its design goals converge strongly with those of TPR. Both aim to provide verifiable path confinement, guaranteeing that sensitive flows remain within explicitly trusted infrastructure. SCION achieves this with inter-domain path authentication and isolation domains, while TPR employs attestation mechanisms to verify the trustworthiness of devices inside an intra-domain topology. Likewise, SCION's transparency through explicit path selection parallels TPR's disclosure of attested device states, giving endpoints auditability and visibility over where their traffic travels. Both also serve as countermeasures to routing-based threats: SCION resists hijacks and inter-domain manipulation, while TPR protects against compromised routers or insider threats within an enterprise fabric.

The technical parallels between the two systems are also striking. SCION's path construction relies on beacons that accumulate cryptographic information as they propagate, a concept mirrored in TPR's potential use of attestation beacons that collect trust measurements hop by hop. Similarly, SCION validates paths through per-packet cryptographic checks of hop fields, while TPR extends the idea to include dynamic re-attestation of node integrity. Even scalability considerations overlap: SCION introduced hierarchical caching and lightweight cryptographic operations to keep overhead manageable, lessons equally relevant for TPR as it balances attestation frequency with performance demands.

Taken together, SCION and TPR can be seen as complementary: the former secures communication across domains, ensuring that inter-organisational traffic never crosses adversarial networks, while the latter provides fine-grained assurance within a domain by verifying the trust status of each forwarding device. In combination, they point toward an end-to-end security model where communications are constrained to trusted paths at both global and local scales. Moreover, SCION's real-world deployments highlight practical lessons for TPR, such as the importance of incremental adoption, optimisation of cryptographic operations, and flexible policy integration. These insights reinforce the feasibility of trusted forwarding and suggest that TPR can follow a similar trajectory, adapting attestation-based routing to coexist with legacy infrastructure while progressively raising the security baseline of enterprise networks.

3.5.2 Relation to FABRID

While SCION is an advanced Internet architecture that gives endpoints significant control over their traffic's forwarding path at the Autonomous System (AS) level, it lacks the fine-grained transparency and attestation capabilities required to implement trusted path routing based on specific router properties. Endpoints can select inter-domain paths using SCION's packet-carried forwarding state, and these paths are cryptographically authenticated by SCION's public-key

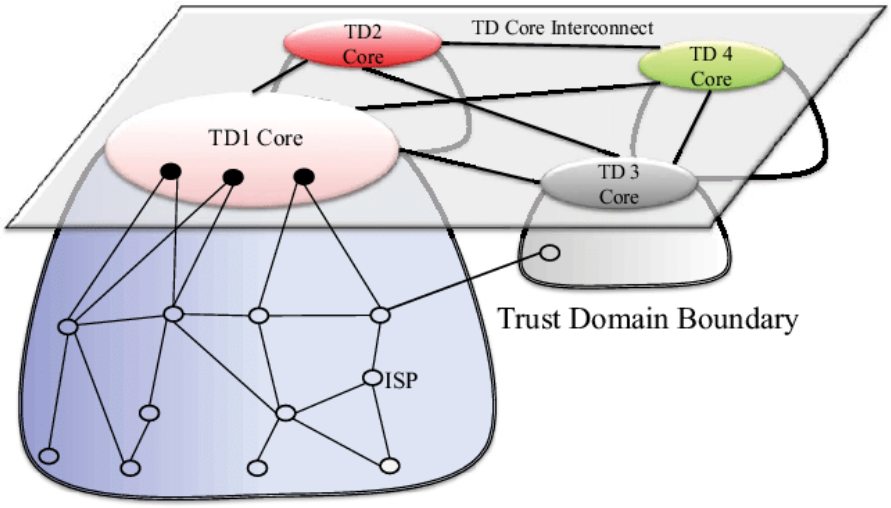

Figure 3.3. Scion architecture overview [source [19]]

infrastructure (CP-PKI), which provides robustness against hijacking, rerouting, and link failures. However, in its base form, SCION operates as a black box: it allows path selection across ASes, but it does not disclose information about the internal routers that compose these domains, nor does it provide attested guaranties about their properties. As a result, endpoints cannot enforce requirements such as vendor selection, software versioning, or exclusion of specific equipment that might pose compliance or security risks.

FABRID (Flexible Attestation-Based Routing for Inter-Domain Networks) was proposed to address these limitations by extending SCION with fine-grained, attestation-based routing [20]. The system enables ASes to expose and transparently attest information about their forwarding devices, allowing endpoints to select paths not only according to AS-level policies but also based on attested router attributes. Examples of such attributes include the software version, vendor, geographic location, or the presence of specific hardware features. FABRID introduces an extensible policy language to express user-defined trust requirements while giving ASes control over the granularity of information they publish to protect internal details. Policy constraints are embedded into packet headers and validated through Remote Attestation, ensuring that only paths matching declared trust requirements are used. For security, FABRID relies on SCION's DRKey infrastructure and a modified version of EPIC for per-packet authentication and path validation, thereby preserving the secrecy and authenticity of policy enforcement decisions.

In essence, FABRID bridges the gap between SCION's inter-domain path awareness and the device-level assurances required for trusted path routing. By combining SCION's scalable inter-domain forwarding with the attestation of internal router properties, FABRID enables endpoints to confine traffic to routes that are both topologically valid and cryptographically verifiable at the device level. This aligns closely with the goals of TPR, as both approaches aim to ensure that sensitive flows remain confined to a trustworthy and verifiable subset of the network.

Chapter 4

Software for Open Netowrking in the Cloud (SONiC)

SONiC is an open-source Network Operating System (NOS) built upon the Linux kernel [2], designed to run on network switches equipped with hardware from multiple vendors and supporting a wide range of Application Specific Integraded Circuits (ASICs). Originally developed and production hardened within the data centres of leading cloud service providers, SONiC provides a comprehensive set of networking capabilities, including support for Border Gateway Protocol (BGP) and Remote Direct Memory Access (RDMA). Its modular architecture enables network engineers to customise and extend functionalities to meet specific operational requirements, while benefiting from the robustness and scalability ensured by a broad and active open-source community. SONiC represents a significant advancement in disaggregated networking, combining vendor interoperability with the agility of cloud-native software design.

4.1 Introducion

Network switches were originally introduced as hardware devices designed to interconnect multiple computers within a Local Area Network (LAN) and to forward Ethernet frames using physical addresses. Operating at the data-link layer, switches form the foundation upon which the network layer is built. The latter enables connectivity across the Internet, with routing functionalities typically executed by routers running protocols such as Open Shortest Path First (OSPF) and Border Gateway Protocol (BGP).

Traditionally, the functionality of switches has been restricted to dataplane forwarding, as they were shipped with closed-source, proprietary Network Operative Systems (NOSs). These systems are conceptualised, developed, and distributed by a single vendor, who delivers the hardware preinstalled with the corresponding operating system. Such vendor lock-in prevents users from modifying the system or installing third-party software. While this behaviour ensures stability and reliability, since vendors rigorously test their software prior to release, it also hinders innovation. In practice, the adoption of new technologies or network scaling is often delayed due to concerns about security, financial costs, and potential downtime [21].

In response, the emergence of bare-metal or white-box switches has introduced greater flexibility for network operators. These devices enable the decoupling of hardware and software, allowing the installation of operating systems independently of the switch vendor. This customisability has attracted both practitioners and major manufacturers, including Dell and HP [22].

SONiC has achieved widespread success in cloud data centres primarily due to its ability to combine scalability, flexibility, and vendor interoperability. A central factor is its modular design, which decouples networking software from the underlying hardware and enables the same software stack to be deployed across heterogeneous switching platforms. This disaggregation

allows operators to avoid vendor lock-in while maintaining consistent operational models across large-scale infrastructures [21].

Another determinant of its success lies in the strength of its open-source community. With contributions from more than 850 members, including cloud providers, hardware vendors, silicon suppliers, and service operators, the ecosystem ensures continuous development and rapid innovation. The availability of enterprise-grade support further strengthens SONiC's position, offering professional services that facilitate its adoption in production-grade environments.

4.2 Architecture

The system architecture of SONiC comprises a collection of modular components that interact through a centralised and scalable infrastructure. This infrastructure is built around a `Redis` database engine [23], a key value store that facilitates a language independent interface, provides mechanisms for data persistence and replication, and enables multi-process communication among all SONiC subsystems. By leveraging the *publisher/subscriber* messaging paradigm offered by the `Redis` engine , applications are able to subscribe exclusively to the data views relevant to their functionality. This design enables developers to avoid low-level implementation details that are not essential to the operation of a given module. SONiC deploys each of its modules in isolated `Docker` containers. This architecture promotes high cohesion among semantically-related components while minimising coupling between logically disjointed ones. Furthermore, each module is developed to remain fully agnostic of platform-specific dependencies, especially those related to hardware-level abstractions. As of the current version, SONiC decomposes its principal functional elements into the following containers:

- `dhcp-relay`
- `pmon`
- `snmp`
- `lldp`
- `bgp`
- `teamd`
- `database`
- `swss`
- `syncd`

Figure 4.1 illustrates a high-level overview of the main functionalities encapsulated within each container and outlines the interactions among them. It is worth noting that not all SONiC applications interact with other internal components; in certain cases, modules obtain their state from external entities. In the diagram, blue arrows denote communications that traverse the centralised `Redis` infrastructure, while black arrows represent alternative mechanisms such as `netlink` sockets or access to the `/sys` virtual filesystem.

Although the majority of SONiC's core functionalities reside within Docker containers, a few essential modules are executed directly on the host Linux system. Notable examples include the configuration utility `sonic-cfggen` and the SONiC command-line interface (`CLI`).

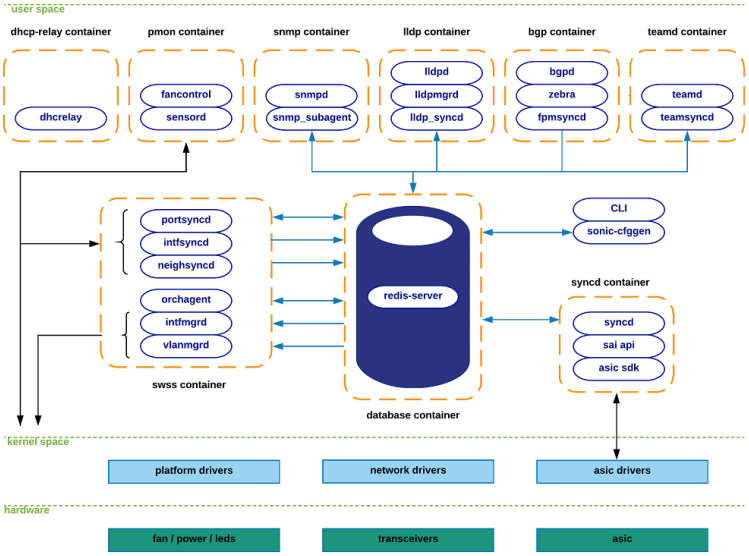

Figure 4.1. SONiC Architecture

4.2.1 The Switch Abstraction Interface (SAI): The Decoupling Layer

The Switch Abstraction Interface, or SAI, is a foundational element of SONiC's architecture and the key enabler of its hardware independence. Defined as a vendor-independent API, SAI provides a consistent, standardized way of controlling switching ASICs. It serves as the critical intermediary, or "translator," between the SONiC software stack and the underlying hardware's specific ASIC drivers.

This decoupling layer is what allows SONiC to run on white-box hardware from multiple vendors. Network hardware vendors implement their own SAI-friendly libraries and software development kits (SDKs), which enables them to develop innovative hardware platforms without needing to build an entire operating system from scratch. This separation of the software and hardware development paths allows for rapid innovation on both fronts and is central to SONiC's ability to prevent vendor lock-in.

4.3 The Inter-Process Communication Framework: Redis, SWSS, and SyncD

At the core of SONiC's modular architecture lies a sophisticated Inter-Process Communication (IPC) framework, designed around a state-driven model. Unlike traditional networking systems, which rely on tightly coupled modules invoking hardware APIs directly, SONiC enables communication through the update of shared states in a centralised database. This architectural choice underpins the containerised paradigm, facilitating independent upgrades, modular scalability, and fault tolerance.

4.3.1 Redis Database

The Redis in-memory database engine, hosted within its own container, functions as the central nervous system of the system. It provides a key-value datastore that consolidates configuration, operational state, and monitoring information into a single, accessible repository. Redis organises this information into five principal databases:

- **APPL_DB:** Stores state generated by application containers, including routes, next-hops, and neighbour entries. It represents the southbound entry point for applications interacting with other SONiC subsystems.

- **CONFIG_DB:** Maintains configuration state such as port settings, interface definitions, and VLAN assignments.

- **STATE_DB:** Records operational state of configured entities and resolves dependencies across subsystems. For instance, it supports the resolution of Link Aggregation Group (LAG) dependencies by verifying the availability of underlying physical ports.

- **ASIC_DB:** Holds ASIC-oriented state in a hardware-friendly format to streamline communication between the `syncd` container and ASIC SDKs.

- **COUNTERS_DB:** Collects counters and statistics for ports and interfaces, which may be used for local CLI requests or exported to external telemetry systems.

Switch State Service (SWSS)

The Switch State Service (SWSS) container encompasses a suite of tools that coordinate communication among all SONiC modules. It primarily manages northbound interactions with the application layer, while some dedicated synchronisation daemons, such as `fpmsyncd`, `teamsyncd`, and `lldp_syncd` operate within their respective containers. The overarching role of these processes is to ensure seamless connectivity between applications and the centralised Redis messaging infrastructure.

Within SWSS, two categories of processes can be identified:

- **State Producers:** These daemons capture system events and publish the corresponding state to `APPL_DB`.
  - *Portsyncd:* Listens to port-related netlink events and exports attributes such as speed, lanes, and MTU.
  - *Intfsyncd:* Processes interface-related events, including the creation or modification of IP addresses, and updates `APPL_DB`.
  - *Neighsyncd:* Monitors neighbour-related events derived from ARP resolution, managing information such as MAC addresses and neighbour address families.

- **State Consumers:** These processes subscribe to Redis updates, process incoming state, and enforce configurations on the system. The central component is *Orchagent*, which consolidates state from synchronisation daemons, interprets it, and publishes hardware-compatible state into `ASIC_DB`. In this role, Orchagent acts simultaneously as a consumer of `APPL_DB` and a producer for `ASIC_DB`. Additional consumer processes, such as `IntfMgrd` and `VlanMgrd`, configure interfaces and VLANs within the Linux kernel in response to changes in `APPL_DB`, `CONFIG_DB`, and `STATE_DB`.

4.4 Key Functional Containers: A Detailed Breakdown

This section provides a description of the functionality encapsulated within the most critical Docker containers of SONiC, as well as essential components operating directly from the Linux host system. Each container is dedicated to a specific networking role, thereby reinforcing the modular and service-oriented architecture of SONiC. A more brieef description is presented in table 4.1

4.4.1 Teamd Container

The `teamd` container implements Link Aggregation (LAG) functionality in SONiC devices. It relies on `teamd`, an open-source Linux implementation of the LAG protocol. The auxiliary process `teamsyncd` enables the interaction between `teamd` and southbound subsystems, ensuring synchronisation of link aggregation state across the stack.

4.4.2 Pmon Container

The `pmon` container is responsible for hardware monitoring. It runs the following key processes:

- **sensord:** Periodically logs sensor readings from hardware components and raises alarms when abnormal behaviour is detected.
- **fancontrol:** Collects and manages fan-related state by interacting with platform-specific drivers.

4.4.3 Snmp Container

The `snmp` container provides Simple Network Management Protocol (SNMP) functionality, supporting external monitoring and management systems. It hosts two primary processes:

- **snmpd:** The main SNMP server, responsible for handling incoming SNMP polls from external entities.
- **snmp-agent (sonic_ax_impl):** A SONiC-specific implementation of an AgentX SNMP subagent, which supplies `snmpd` with information collected from Redis databases.

4.4.4 Dhcp-relay Container

The `dhcp-relay` agent forwards DHCP requests from subnets lacking a DHCP server to one or more servers located on different subnets, ensuring transparent address assignment across heterogeneous topologies.

4.4.5 Lldp Container

The `lldp` container provides Link Layer Discovery Protocol (LLDP) functionality. It includes:

- **lldpd:** The daemon that establishes LLDP connections with neighbouring devices, exchanging system capabilities.
- **lldp_syncd:** Uploads LLDP-discovered state into Redis, enabling applications such as SNMP to consume the information.
- **lldpmgr:** Provides incremental configuration capabilities to the LLDP daemon by subscribing to the `STATE_DB`.

4.4.6 Bgp Container

The `bgp` container runs routing stacks such as Quagga or FRR, supporting multiple routing protocols including BGP, OSPF, IS-IS, and LDP. Despite its name, its scope extends beyond BGP alone. The main processes are:

- **bgpd:** Implements the BGP protocol. It receives routing state from external peers via TCP/UDP and propagates it to the forwarding plane through the `zebra`/`fpmsyncd` interface.

- **zebra:** Functions as a routing manager, maintaining the kernel routing table, resolving interfaces, and providing route redistribution services. Zebra computes the FIB and pushes it to both the kernel and the southbound Forwarding Plane Manager (FPM).
- **fpmsyncd:** Collects the FIB state generated by Zebra and writes it into the `APPL_DB` within Redis.

4.4.7 Syncd Container

The `syncd` container provides the synchronisation layer between SONiC's software state and the hardware ASIC. Its main components are:

- **syncd:** Executes the synchronisation logic by linking with the vendor-provided ASIC SDK library. It subscribes to the `ASIC_DB` for state updates and publishes hardware state back into Redis.
- **SAI API:** The Switch Abstraction Interface (SAI) defines a vendor-independent API for programming forwarding elements, offering a uniform abstraction layer.
- **ASIC SDK:** Vendors supply an SDK implementation compatible with the SAI specification, typically delivered as a dynamic library invoked by the `syncd` process.

4.4.8 CLI and Configuration Tools

On the host system, two important modules provide user interaction and configuration management:

- **CLI:** The Command-Line Interface component, based on Python's Click library, delivers a flexible mechanism for managing and monitoring SONiC.
- **sonic-cfggen:** A configuration generation utility invoked by the CLI, responsible for applying configuration changes and handling system-level configuration tasks.

Table 4.1. Overview of Key SONiC Containers and Their Functions

Container	**Primary Function**	**Key Processes**	**Description**
`bgp`	Routing and Control Plane	FRR routing stack	Supports routing protocols (BGP, OSPF, IS-IS) via the FRR stack, enabling Layer 3 control-plane services.
`swss`	State Management Hub	`orchagent`, `portsyncd`	Coordinates communication via Redis, ensuring state consistency and translating configurations to `ASIC_DB`.
`syncd`	Hardware Synchronisation	`syncd` process	Interfaces with hardware by applying updates from `ASIC_DB` through the SAI API.
`database`	Central Data Store	Redis daemon	Hosts the Redis key-value datastore, storing configuration, operational state, and counters across all components.
`pmon`	Hardware Monitoring	`sensord`, `fancontrol`	Collects telemetry from fans and sensors, logging data and triggering alarms on anomalies.
`teamd`	Link Aggregation (LAG)	`teamd`, `teamsyncd`	Manages LAG groups and coordinates with other networking services.
`lldp`	Device Discovery	`lldpd`, `lldp_syncd`	Implements LLDP to advertise and discover neighbouring devices.
`snmp`	Network Management	`snmpd`, `snmp-agent`	Provides SNMP services for monitoring via external management systems.
`dhcp-relay`	DHCP Service	N/A	Relays DHCP requests across subnets lacking local DHCP servers.

Chapter 5

Design

The objective of this work is to implement a network path attestation framework that enables the verification of router trustworthiness through remote attestation. This chapter presents the architecture, components, and design decisions that support this goal. The framework integrates SONiC as the network operating system, TPM as the hardware root of trust, and Keylime as the attestation orchestration platform. The system operates in a virtualized environment to facilitate experimentation and validation of the attestation workflow.

5.1 Design Overview

The proposed architecture addresses the fundamental challenge of verifying whether a network device can be trusted to forward sensitive traffic. The solution combines three key mechanisms: network state measurement, which captures the configuration of network interfaces, routing tables, and other kernel-level networking parameters in a canonical form; TPM-based integrity reporting, which extends measured network configurations into a dedicated PCR to create cryptographic evidence of the system state; and remote attestation via Keylime, which leverages a challenge-response protocol to verify that the measured state matches approved reference values.

Figure 5.1 illustrates the high-level architecture, showing the interaction between the SONiC node (attested device), the Keylime components (agent, registrar, verifier), and the TPM.

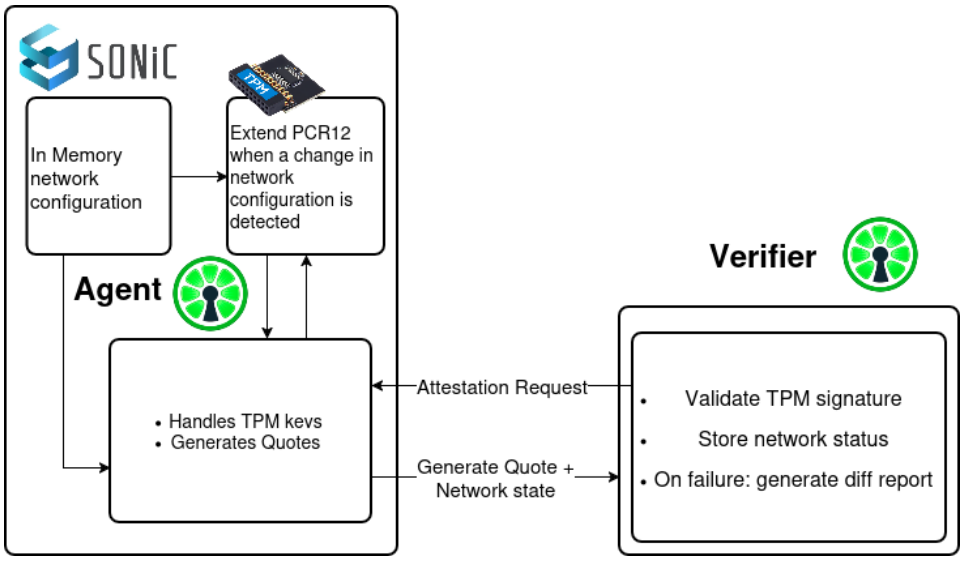

Figure 5.1. High-level architecture of the network attestation framework

5.2 Network Configuration Measurement

The network configuration measurement module captures a complete snapshot of the network stack state and produces a deterministic cryptographic measurement that reflects the system configuration. The module employs the pyroute2 library to interface with the Linux kernel netlink protocol, enabling direct access to network subsystem data structures.

5.2.1 Measurement Categories

The measurement process systematically captures six distinct categories of network configuration data, along with firewall rules, ensuring comprehensive coverage of the network stack state.

Network Interfaces

The module enumerates all network interfaces present in the system, including physical interfaces, virtual interfaces, bridges, VLANs, and tunnel devices. For each interface, the following stable configuration attributes are captured:

- Interface name and operational parameters
- MAC address (filtered for interfaces with dynamic addresses)
- Maximum Transmission Unit (MTU)
- Link type and layer 2 encapsulation
- Interface-specific configuration parameters

IP Address Assignment

All IPv4 and IPv6 addresses assigned to network interfaces are captured, with the following details:

- IP address and prefix length
- Address scope (global, site, link, or host)
- Interface association
- Address flags and attributes

Routing Tables

The module captures routing table entries for both IPv4 and IPv6 protocols. For each route, the measurement includes:

- Destination prefix and prefix length
- Gateway address
- Outgoing interface
- Route metric and priority
- Routing protocol (static, kernel, dynamic)
- Route type (unicast, multicast, blackhole, unreachable)

Routing Policy Rules

Policy-based routing rules are captured to document the logic determining routing table selection. These rules specify conditions based on source address, destination address, interface, or packet markings that determine which routing table is consulted for forwarding decisions.

VLAN Configuration

Virtual LAN interfaces are identified and their configuration parameters are measured, including:

- VLAN ID
- Parent physical interface
- Encapsulation protocol (802.1Q, 802.1ad)
- VLAN-specific attributes

Tunnel Interfaces

Tunnel configurations are captured for various encapsulation technologies, including VXLAN, GRE, SIT (IPv6-in-IPv4), and IPsec tunnels. The measurement records:

- Tunnel type and encapsulation protocol
- Local and remote endpoint addresses
- Tunnel-specific parameters (e.g., VXLAN VNI, GRE key)
- Encapsulation options

5.2.2 Firewall Rules Measurement

In addition to network configuration, the module captures the complete firewall ruleset to ensure that packet filtering policies are included in the attestation measurement.

iptables Rules

For IPv4 packet filtering, the module collects rules from all standard iptables tables:

- **filter**: Default packet filtering (INPUT, OUTPUT, FORWARD chains)
- **nat**: Network Address Translation rules
- **mangle**: Packet alteration rules
- **raw**: Connection tracking exemptions
- **security**: Mandatory Access Control rules

For each table, the measurement captures chain policies and rule specifications. Packet and byte counters are explicitly removed from the measurement to maintain stability, as these counters increment with traffic and do not reflect configuration changes.

ip6tables Rules

IPv6 firewall rules are collected using an identical approach to iptables, capturing rules from all standard ip6tables tables with counter removal to ensure measurement stability.

nftables Detection

The module checks for the presence of nftables, the successor to iptables. If nftables is active, the complete ruleset is captured as part of the measurement.

5.2.3 Measurement Stability and Normalization

A critical requirement for attestation is measurement determinism: identical configurations must produce identical hash values. The module implements comprehensive filtering and normalization to ensure stable measurements.

Exclusion of Dynamic Data

The module explicitly excludes dynamic and volatile fields that change at runtime without reflecting configuration modifications:

- Network interface statistics (packet counters, byte counters, error counts)
- Operational state indicators (link carrier status, interface up/down state)
- Runtime-assigned identifiers (interface indices, which vary across boots)
- Dynamically assigned MAC addresses on virtual interfaces
- Firewall rule packet and byte counters
- Timestamps and event markers
- Neighbor table entries (ARP cache)
- Route cache information
- Bridge timers (hello timer, topology change timer)

Canonical JSON Encoding

All collected data is serialized to canonical JSON format with sorted keys and consistent separators. This ensures that structurally identical configurations produce byte-identical JSON output, regardless of collection order.

Attribute Normalization

Netlink messages contain attribute lists that may appear in arbitrary order. The module converts these lists to dictionaries with sorted keys, ensuring deterministic serialization. Circular references in data structures are detected and replaced with sentinel values to prevent infinite recursion during serialization.

5.2.4 Hash Computation and PCR Extension

After collecting and normalizing all measurement data, the module performs the following operations:

1. Serializes the complete measurement structure to canonical JSON format
2. Computes the SHA-256 hash of the canonical representation
3. Compares the computed hash with the previous measurement hash

4. Extends TPM PCR 12 with the new hash if the configuration has changed

This approach ensures that PCR extension occurs only when actual configuration changes are detected, preventing unnecessary attestation failures due to routine system activity. The measurement data, metadata, and hash value are written atomically to the filesystem, enabling verification systems to retrieve the detailed configuration state that corresponds to the PCR value.

5.2.5 Atomic Write and Persistence

After computing the measurement hash and extending the PCR, the module writes the collected data and metadata to disk using atomic file operations. This ensures that partial writes do not occur in the event of a system crash or interruption.

The module generates three output files. First, `kernel_network.json` contains the complete network configuration in normalized JSON format. Second, `meta.json` includes metadata such as the hostname, timestamp, hash value, PCR index, and extension status. Third, `measurement_hash.txt` stores the SHA256 hash in plaintext for easy reference. This hash is also used to check whether the network has changed compared to the previous measurement. All files are stored in `/tmp/lib/attest/kernel/current/` on the SONiC node, which is created at boot time through a systemd service.

5.3 Integration Architecture

5.3.1 SONiC-Keylime Integration

The integration between SONiC and Keylime is achieved by deploying the measurement module and the Keylime agent on the SONiC node. This integration enables continuous monitoring of network configuration state through hardware-based attestation.

Measurement Triggering

The measurement module operates on a timer-based schedule to ensure continuous monitoring of network configuration state. Upon system boot, the module delays measurement for 2 minutes to allow all network configurations to stabilize in memory. Following this initialization period, the module automatically captures the network state every 10 seconds, computing the configuration hash and extending PCR 12 when changes are detected.

This decoupled approach separates measurement from attestation verification: the measurement module continuously monitors and records configuration state, while the Keylime verifier requests TPM quotes on-demand to validate system integrity. When the verifier initiates an attestation round, the agent retrieves the most recent measurement data and requests a TPM quote over the selected PCRs, including PCR 12.

Attestation Workflow

The attestation process follows a structured sequence of operations:

1. The measurement module collects network configuration data from the kernel via netlink
2. The collected data is normalized and hashed using SHA-256 to produce a deterministic measurement
3. The hash is extended into PCR 12 of the TPM, creating cryptographic evidence bound to the current network state

4. The Keylime verifier sends a quote request to the agent
5. The agent requests a TPM quote over the selected PCRs, including PCR 12
6. The TPM generates and signs the quote
7. The agent transmits the quote, PCR values, and current network state to the verifier
8. The verifier validates the quote signature and compares PCR 12 against the expected reference value
9. The verifier updates the trust state of the agent based on the appraisal result

If the network configuration changes (e.g., VLAN addition, routing table modification), the measurement hash changes, causing PCR 12 to reflect a new state. During the next attestation round, the verifier detects the divergence from the expected baseline and classifies the node as untrusted unless the new state has been explicitly approved.

5.3.2 Network State Verification

To assess whether a SONiC node is trusted, the attestation process extends beyond PCR validation to include comprehensive network state analysis. During each attestation round, nodes transmit their complete network configuration state through a dedicated HTTP endpoint, encompassing the measurement categories described in Section 5.2.1.

While PCR values provide cryptographic evidence of configuration state integrity, they do not convey semantic information about the specific changes that occurred. A PCR mismatch indicates that *something* changed, but not *what* changed or *how*. To address this limitation, the verifier maintains a comprehensive record of each node's network state in the Cloud Verifier database alongside the corresponding PCR baseline.

5.3.3 Configuration Change Analysis

When an attestation failure occurs, the verifier performs a differential analysis by comparing the current network state against the previously trusted state retrieved from the database. The system tracks three fundamental change types across all network configuration categories:

- **Addition**: New configuration elements introduced (e.g., IP addresses assigned, routes added, firewall rules inserted, VLAN interfaces created)
- **Removal**: Configuration elements deleted (e.g., IP addresses unassigned, routes deleted, firewall chains removed, tunnel interfaces destroyed)
- **Modification**: Existing configuration elements altered (e.g., interface MTU changed, firewall rule parameters updated, routing rule priorities adjusted)

The differential analysis operates at two levels of detail, controlled by the `network_status_diff_verbose` configuration parameter.

Summary Mode

In summary mode, the system provides high-level change statistics for each affected category, including the number of routes, addresses, firewall rules, links, VLANs, and tunnels that were added, removed, or modified.

Verbose Mode

When verbose analysis is enabled, the system generates detailed semantic descriptions for each detected change:

- **Route changes**: Destination prefixes, gateways, outgoing interfaces, and metrics
- **Address changes**: Specific IP addresses with interface association and address family
- **Firewall modifications**: Rule specifications including source/destination addresses, ports, protocols, and actions within each table and chain
- **Link modifications**: Interface operational status, MTU adjustments, flag modifications, and MAC address updates
- **Tunnel and VLAN changes**: Tunnel types, endpoints, encapsulation parameters, and VLAN identifiers
- **Routing policy changes**: Rule priorities, source/destination selectors, and routing table associations

The differential analysis output is structured as a JSON object containing:

- `has_changes`: Boolean indicating whether differences were detected
- `summary`: Array of human-readable change descriptions
- `changes`: Field-level comparison showing old and new values for each modified category
- `details`: (verbose mode only) Array of detailed change objects with specific modification information

Figure 5.2 shows an example of the verifier output when network configuration changes are detected during an attestation failure. This semantic analysis enables administrators to determine whether detected changes represent legitimate updates requiring baseline adjustment or unauthorized modifications indicating a security incident.

Figure 5.2. Cloud Verifier differential analysis output identifying network configuration changes that triggered attestation failure. The report details added IP addresses, modified network interfaces, and new routing table entries detected on the agent.

Policy Update and Trust Recovery

If a configuration change was authorized, the network administrator can update the reference value in the verifier's policy to reflect the new baseline. After the policy update, the next attestation round succeeds, and the agent is reclassified as trusted. The change report is stored as part of the attestation failure event, providing forensic evidence for incident investigation and compliance auditing.

5.3.4 Communication Flows

Figure 5.3 illustrates the complete sequence of interactions during a typical attestation round, showing the coordination between the measurement module, Keylime agent, TPM, and verifier.

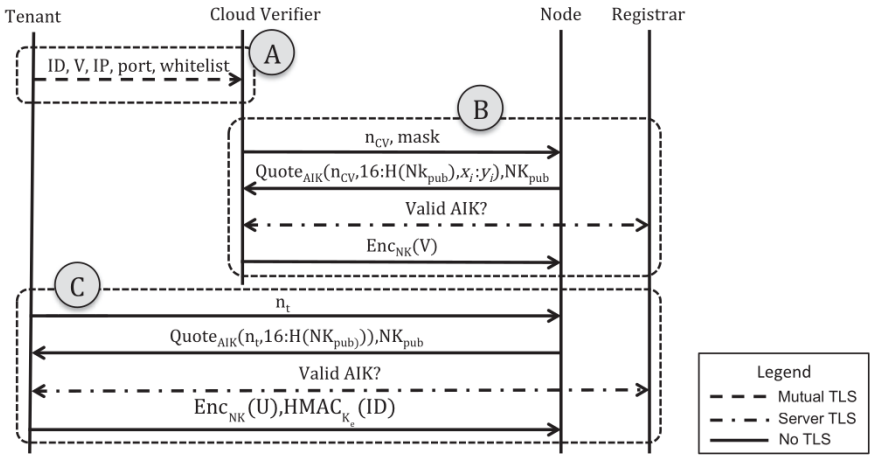

Figure 5.3. Sequence diagram showing the attestation workflow between SONiC node components and the Keylime verifier

5.4 Trust Evaluation Policy

5.4.1 Golden Measurement Generation

Reference values are generated by measuring the network configuration when the system is in a known-good state. The process involves configuring the SONiC node with the desired network settings (interfaces, addresses, routes, VLANs), executing the measurement module to capture the configuration and compute the hash, recording the resulting PCR 12 value after extension, and storing this value as the reference measurement in the verifier's policy.

5.4.2 Policy Updates

If the network configuration must be changed for operational reasons, the reference value must be updated accordingly. The verifier provides an API to modify policies, allowing administrators to replace the old reference value with the new one.

5.4.3 Multiple Reference Values

In some cases, multiple configurations may be acceptable. For example, a router may have different valid states depending on whether certain optional services are enabled. The verifier can

be configured to accept any one of a set of reference values, providing flexibility while maintaining security.

5.4.4 Appraisal Policy

The verifier's appraisal policy determines the conditions under which an agent is classified as trusted or untrusted.

Policy Types

Three primary policy types are supported. Strict matching requires that PCR 12 exactly match the reference value, with any deviation resulting in an untrusted classification. Allowable ranges provide more sophisticated policies that may allow certain dynamic variations; for example, if the measurement includes a timestamp, the policy can specify a tolerance window within which slight differences are acceptable. Multi-PCR evaluation enables the verifier to evaluate multiple PCRs in combination; for instance, if PCRs 0–7 indicate a secure boot state and PCR 12 indicates a valid network configuration, the agent is trusted only if both sets of PCRs are valid.

Chapter 6

Implementation

This chapter details the implementation of the proposed network path attestation solution, focusing on the integration of TPM-based attestation mechanisms within the SONiC network operating system. The solution builds upon modifications to both the SONiC agent environment and the Keylime remote attestation framework to enable real-time verification of network device integrity. The chapter begins by examining the custom measurement module integrated into SONiC nodes, which monitors network configuration state and extends TPM Platform Configuration Registers (PCRs) to reflect configuration changes. It then describes the modifications introduced to the Keylime agent running within SONiC. Finally, the chapter details the enhancements to the Keylime verifier infrastructure, specifically the integration of network-specific verification logic that enables the system to correlate TPM quotes with semantic network states.

6.1 Agent Modifications within SONiC

The integration of remote attestation capabilities into SONiC requires substantial modifications to the Keylime agent to accommodate the network-specific measurement requirements and the unique operational environment of network devices. This section describes the implementation of these modifications, which encompass both the measurement collection mechanism and the agent's interaction with the TPM.

6.1.1 Measurement Module Architecture

The core modification within the SONiC agent environment consists of a custom Python-based measurement module designed to monitor network configuration state and generate cryptographic evidence of configuration integrity. This module operates as a standalone Python service that executes periodically within the SONiC environment.

The implementation employs a timer-based architecture utilising `systemd` units to manage execution scheduling and resource constraints. Two distinct timer intervals govern the measurement process: an initial boot delay of 2 minutes allows the SONiC system to complete initialisation and load all network configurations into memory, while subsequent measurements occur every 10 seconds to capture configuration changes with minimal latency. This dual-timer approach ensures initial system stability while maintaining continuous monitoring without imposing excessive computational overhead.

Data Structures and Storage

The measurement module maintains network configuration state through a hierarchical data structure. The `kernel_state` dictionary serves as the root container, organising data into clearly delineated sections for links, addresses, routes, routing rules, VLANs, tunnels, `iptables` rules, and `ip6tables` rules.

The module outputs three distinct files atomically to the `/tmp/attest/kernel/current` directory:

- `kernel_network.json`: Contains the complete network state snapshot in canonical JSON format with sorted keys to ensure reproducibility.
- `meta.json`: Records measurement metadata including timestamps, the SHA-256 hash of the network state, PCR extension status, and collection method identifiers.
- `measurement_hash.txt`: Provides direct access to the current measurement hash for rapid comparison operations.

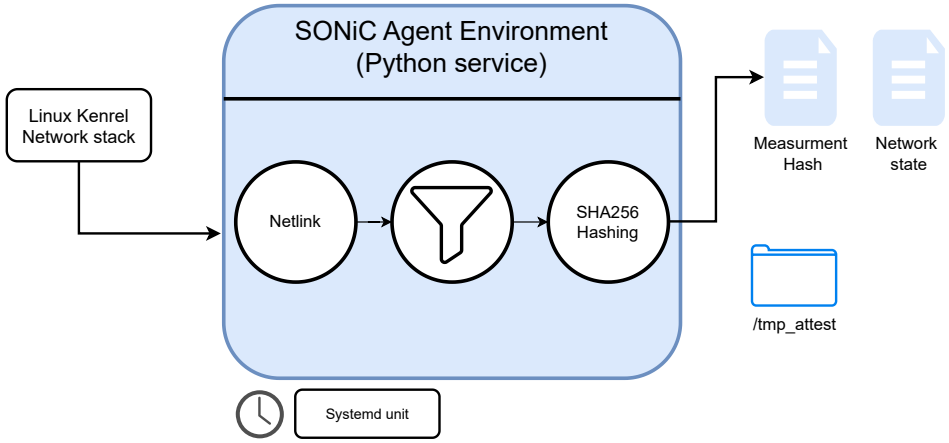

Figure 6.1. Measurment workflow

6.1.2 Data Collection and Normalisation

Configuration normalisation constitutes a critical component of the measurement module, transforming dynamic kernel data structures into stable, reproducible measurements suitable for cryptographic attestation. The normalisation process addresses fundamental challenges inherent in measuring live network systems, where operational state changes continuously while configuration state remains constant.

Monitored Network Parameters

The measurement module captures configuration data through direct interaction with the Linux kernel networking subsystem via the Netlink protocol and the `pyroute2` library. Key monitored parameters include:

- **Link Layer:** MTU settings, administrative state, and master-slave relationships.
- **IP Addresses:** IPv4/IPv6 assignments, excluding link-local addresses.
- **Routing:** Unicast and multicast routes, filtering kernel-generated dynamic routes.
- **Security Policies:** Full `iptables` and `ip6tables` rulesets.

Canonicalisation and Dynamic Field Exclusion

To ensure consistent measurements, the module implements a rigorous canonicalisation algorithm. The `remove_dynamic_fields` function eliminates fields that vary based on operational state rather than configuration intent. This includes:

1. **Statistics:** Packet and byte counters are stripped from interfaces and firewall rules.
2. **Operational State:** Attributes like `IFLA_OPERSTATE` (link status) are removed.
3. **Temporal Data:** Timestamps and object creation times are excluded.
4. **Generated Identifiers:** Randomised MAC addresses on virtual interfaces (e.g., Docker bridges) are filtered.

Finally, data is serialised to JSON with sorted keys to guarantee that semantically equivalent configurations produce bitwise identical strings for hashing.

6.1.3 Agent Integration and Communication

The Keylime agent running within SONiC interacts with the measurement module to manage the attestation evidence generation process. Upon initialisation, the agent establishes communication channels with both the TPM device and the measurement module.

The agent implements a polling mechanism that periodically queries the measurement module. When configuration changes are detected, the agent coordinates the TPM quote generation process, requesting a signed attestation that includes the extended PCR 12 value. This evidence is serialised and transmitted to the Keylime verifier through secure channels.

6.2 Verifier Enhancement for Network State Correlation

The standard Keylime verifier validates attestation quotes by comparing TPM PCR values against expected measurements and processing runtime integrity data such as IMA logs. However, this baseline functionality lacks the ability to correlate PCR extensions with semantic network state information. To address this limitation, the verifier was extended to retrieve and analyse network configuration data from attested nodes, enabling detection of unauthorised network changes that trigger PCR modifications.

6.2.1 Network Status Retrieval Procedure

The enhanced verifier implements a dedicated function `invoke_get_network_status` that contacts the SONiC measurement server running on each attested node. This function executes before requesting attestation quotes, establishing a temporal correlation between network state and subsequent PCR measurements.

The retrieval procedure operates as follows. First, the verifier constructs an HTTP GET request to the measurement server endpoint, using configurable port and path parameters. The default configuration targets port 5000 with the `/measurement` endpoint, though these values can be customised per-agent or globally through verifier configuration. The request uses HTTP rather than HTTPS because the measurement data itself is not security-critical; the TPM quote provides cryptographic integrity protection for the measurements that matter.

Upon receiving a successful HTTP 200 response, the verifier parses the JSON payload containing network configuration data. This payload includes kernel state information with details on network interfaces, addresses, routes, VLANs, firewall rules, and tunnels. The verifier validates the JSON structure and logs successful retrieval operations. If the request fails due to network errors, HTTP error codes, or JSON parsing failures, the verifier logs appropriate warnings but

continues with the attestation process. This design choice ensures that temporary measurement server unavailability does not prevent attestation of nodes where the underlying TPM hardware remains functional.

The network status data is stored temporarily in memory during quote processing. The verifier does not persist this data to its database until attestation succeeds, implementing an optimistic approach where only validated network states are retained as reference points.

6.2.2 State Comparison and Change Detection

When attestation fails, the verifier performs a comparative analysis to determine whether network configuration changes contributed to the failure. This analysis implements a differential checking procedure that compares the current network state against the last successful attestation state.

The procedure begins by retrieving fresh network status data immediately after attestation failure. This second retrieval captures the most current network state, accounting for any changes that may have occurred between the initial retrieval and quote validation. The verifier then queries its database to obtain the network status stored during the previous successful attestation for the same agent.

The comparison function `diff_network_status` analyses multiple network configuration dimensions, including network addresses, routing tables, routing rules, firewall rules (both `iptables` and `ip6tables`), network links, tunnels, VLANs, and `nftables` configurations. For each dimension, the function performs deep equality checking to identify modifications. The comparison handles different data structures appropriately: for list-based configurations, it checks both length and content; for dictionary-based configurations such as firewall rules, it counts rule entries and identifies added or removed keys.

The comparison produces three outputs:

1. A boolean flag indicating whether any changes were detected.
2. A structured dictionary containing detailed change information for each modified network dimension.
3. A human-readable summary describing the nature of changes.

For firewall rules, the summary includes rule count differentials. For other configurations, it identifies whether entries were added, removed, or modified.

6.2.3 Network State Storage and Failure Correlation

Upon successful attestation, the verifier persists the current network status to the agent's database record through the `last_successful_network_status` field. This field stores the complete network configuration as JSON, establishing a baseline for future comparisons. The storage operation occurs within a database transaction, ensuring atomicity with other attestation state updates.

When attestation failures occur, the verifier augments the failure event with network status information. If network changes are detected, the failure event includes the comparison summary, detailed change information, and both current and previous network states. This contextual data enables operators to determine whether attestation failures stem from unauthorised network modifications or other integrity violations. If no network changes are detected, the verifier records this explicitly, helping rule out network configuration as the cause of attestation failure.

The verifier logs all network status operations at appropriate severity levels. Successful retrievals and storage operations generate informational logs, while failures to retrieve network data or database errors produce warnings. Network change detection during attestation failures triggers warning-level logs that include the change summary, providing immediate visibility into configuration modifications that may correlate with security events.

This enhancement preserves the verifier's core attestation functionality while adding network-aware diagnostics. The verifier continues to validate quotes, process IMA logs, and enforce attestation policies according to standard Keylime procedures, with network status analysis serving as supplementary forensic information rather than a primary attestation mechanism.

Chapter 7

Tests

The present chapter concerns the testing and validation phase of the proposed network path attestation architecture. A series of tests has been devised to ensure the system's correctness, reliability, and performance under various operational conditions. Functional tests verify the correct behaviour of the remote attestation process, particularly focusing on scenarios involving legitimate network configurations and unauthorised modifications. The evaluation encompasses the system's capacity to detect configuration changes and respond appropriately by updating node trust status. Performance tests analyse attestation times and measurement overhead, thereby assessing the system's efficiency and scalability in relation to network complexity.

7.1 Testbed

The prepared testbed for the proposed solution is composed of two machines:

- A host machine running the Keylime Registrar and Keylime Verifier components. The machine operates Ubuntu Server 24.04 LTS and serves as the attestation orchestration platform.
- A network node running SONiC (Software for Open Networking in the Cloud) as the attester machine equipped with a TPM hardware module. This node executes the Keylime agent alongside custom measurement modules that monitor network configuration state. The node operates SONiC version 202505 with the integrated Python measurement pipeline.

Both machines are connected via network infrastructure allowing mTLS communication between Keylime components.

7.2 Functional Tests

The first step in the testing process is to understand the correctness of the proposed solution through functional testing. When the SONiC node maintains an unmodified network configuration matching baseline measurements, the Keylime Verifier is expected to classify the node as trusted. Conversely, when unauthorised network modifications occur, the system must detect the configuration change through PCR value divergence and classify the node as untrusted.

7.2.1 Initial Deployment and Baseline Attestation

The functional test begins with complete system deployment following the procedures documented in the User Manual (Appendix A). This includes:

1. Keylime Registrar and Verifier installation on the host machine
2. TPM provisioning and certificate configuration
3. SONiC node preparation with measurement module installation
4. Service activation using systemd timers for periodic measurement
5. Initial agent registration with the Verifier

Following deployment, the system enters a 2-minute stabilisation period to allow network services to initialise completely. During this period, the measurement module performs its first configuration capture without extending PCR 12, establishing the baseline state.

After stabilisation, the measurement module begins periodic execution every 10 seconds, capturing approximately 90KB of network configuration data in 300-500 milliseconds per cycle.

To initiate the attestation process, the SONiC node is registered with the Verifier using the tenant command-line interface, specifying the expected PCR 12 value corresponding to the baseline network configuration:

```
sudo keylime_tenant -c add -t 192.168.122.76 \
  -u 31b26a7c-bf78-4790-8d12-7f15b9f097f2 \
  --tpm_policy '{"12": "7EA6573EEC5355757E8B4AF220D431BF435C4368661B63F779DF01A0284DAFE8"}'
```

With the agent registered, the Verifier initiates periodic attestation cycles. During baseline operation with an unmodified network configuration, the Verifier successfully validates the TPM quote against expected PCR values and classifies the SONiC node as trusted.

7.2.2 Network Modification Test

This test validates the system's capacity to detect unauthorised network configuration changes. The test procedure modifies the SONiC node's network state by executing the `vlan-set.sh` script, which introduces new VLAN interfaces and tunnel configurations not present in the baseline state:

```
bash vlan-set.sh
```

The `vlan-set.sh` script performs several types of network configuration modifications. For VLAN creation, the script executes commands such as:

```
config vlan add 100
config vlan member add 100 Ethernet0
```

For tunnel interface configuration, typical operations include:

```
config interface ip add Tunnel0 192.168.10.1/24
config interface startup Tunnel0
```

Additional routing protocol modifications are performed through commands like:

```
config bgp neighbor add 10.0.0.2 remote-as 65001
config route add prefix 172.16.0.0/16 nexthop 10.0.0.1
```

Upon detecting the configuration change through its periodic monitoring cycle, the measurement module captures the modified network state, normalises the configuration data, and extends PCR 12 with the new measurement. This PCR extension occurs within 10 seconds of the configuration modification, given the measurement interval.

When the Verifier performs its next attestation request, the TPM quote contains a PCR 12 value that diverges from the expected baseline. The Verifier detects this discrepancy during quote validation and classifies the SONiC node as untrusted. The attestation failure is logged by the Verifier, and the node's trust status is updated accordingly in the system state. Additionally, the enhanced Verifier retrieves detailed information about the network configuration changes that caused the attestation failure:

```
keylime.tpm - ERROR - PCR #12:
cf6049d53a35136841579ce9e46477dafec99c152c97c0650d287e3dc50b5e4c
from quote (from agent 31b26a7c-bf78-4790-8d12-7f15b9f097f2)
does not match expected value
['7ea6573eec5355757e8b4af220d431bf435c4368661b63f779df01a0284dafe8']

keylime.verifier - INFO - Attestation failed for agent
31b26a7c-bf78-4790-8d12-7f15b9f097f2, fetching current network status

keylime.verifier - INFO - Contacting agent 31b26a7c-bf78-4790-8d12-7f15b9f097f2
network status endpoint: http://192.168.122.76:5000/measurement

keylime.verifier - INFO - Successfully retrieved network status from agent
31b26a7c-bf78-4790-8d12-7f15b9f097f2

keylime.verifier - WARNING - Network status changes detected
for agent 31b26a7c-bf78-4790-8d12-7f15b9f097f2 during attestation failure:
addresses: 7 added, links: 1 modified, routes: 21 added,
tunnels: 7 added, vlans: 3 added
```

The log output demonstrates that the Verifier not only detects the attestation failure through PCR mismatch, but also retrieves semantic network state information from the agent. This additional context provides detailed information about the specific changes that occurred, including the addition of IP addresses, modifications to network links, creation of routes, tunnels, and VLANs. This semantic correlation capability extends beyond standard TPM attestation by enabling administrators to understand precisely what configuration changes triggered the attestation failure.

This test confirms that the architecture successfully detects network configuration modifications within the measurement interval and propagates the trust status change through the attestation framework.

7.3 Performance Tests

Performance evaluation focuses on quantifying the temporal characteristics of the attestation architecture and identifying potential bottlenecks in the measurement and verification pipeline.

7.3.1 Network Complexity Impact on Measurement Performance

The measurement module's execution time is directly influenced by the volume of network configuration data processed. To quantify this relationship and assess the system's scalability, a controlled performance evaluation was conducted using a custom test framework that systematically increases network configuration complexity while measuring the resulting impact on measurement execution time and output data size.

Test Methodology

The performance test employs an iterative approach where each iteration adds a fixed quantity of network configuration elements before executing the measurement module. The test framework incrementally adds the following configuration types at each iteration:

- **VLAN Interfaces**: 10 VLAN interfaces per iteration, created on the base physical interface using 802.1Q tagging with sequential VLAN IDs
- **Static Routes**: 100 routing table entries per iteration, using dummy destination subnets in the 10.0.0.0/8 range with fabricated gateway addresses
- **IPv4 Firewall Rules**: 50 iptables rules per iteration, configured as TCP port-specific ACCEPT rules in the INPUT chain
- **IPv6 Firewall Rules**: 25 ip6tables rules per iteration, similarly configured for IPv6 traffic filtering

Each test iteration follows a standardised sequence: configuration elements are added to the system, the measurement module is executed, and performance metrics are captured. The measurement execution time is recorded using high-precision timing, while the output file size represents the volume of normalised configuration data processed. Between iterations, a brief stabilisation period ensures that all configuration changes are fully propagated through the node internal memory.

The test framework automatically manages the complete lifecycle of test configurations, including systematic clean up of all added network elements upon test completion to restore the system to its original state. This approach ensures that performance measurements reflect only the intended complexity variations without interference from residual test artifacts.

Performance Results

Table 7.1 presents the measurement performance across twenty-one test iterations, demonstrating the correlation between configuration data size and execution time. The File Size column represents the total volume of normalised configuration data processed in each iteration, while Execution Time quantifies the complete measurement cycle duration from data retrieval through PCR extension. The Efficiency metric, calculated as milliseconds per kilobyte, provides insight into the measurement module's computational scaling characteristics.

Iteration	File Size (KB)	Exec Time (ms)	Efficiency (ms/KB)
1	82.5	362.7	4.395
3	122.5	381.8	3.118
5	177.9	435.7	2.449
7	248.6	526.6	2.119
99	335.1	533.7	1.593
11	437.1	629.1	1.439
13	554.7	720.2	1.298
15	687.8	791.6	1.151
17	836.0	911.8	1.091
19	1000.2	1044.2	1.044
20	1088.2	1059.1	0.973

Table 7.1. SONiC Measurement Performance Across Network Complexity Levels

The results reveal sub-linear scaling behaviour: as network complexity increases from 82.5 KB to 1088.2 KB, execution time grows from 362.7 ms to 1059.1 ms. The efficiency metric improves from 4.395 ms/KB to 0.973 ms/KB, indicating that the measurement module exhibits better

performance characteristics at higher data volumes. This behaviour suggests that fixed-overhead operations (Redis connection establishment, TPM initialisation, PCR extension) constitute a significant portion of total execution time at lower complexity levels, while variable costs associated with data processing scale efficiently with configuration size.

Figure 7.1 illustrates the efficiency convergence behaviour observed across the test iterations. The plot demonstrates that efficiency values exhibit high variability during initial iterations, where small configuration sizes amplify the impact of fixed-overhead operations. As network complexity increases beyond iteration 5, efficiency stabilises around the mean value of 1.969 ms/KB with reduced variance, remaining within one standard deviation of the mean.

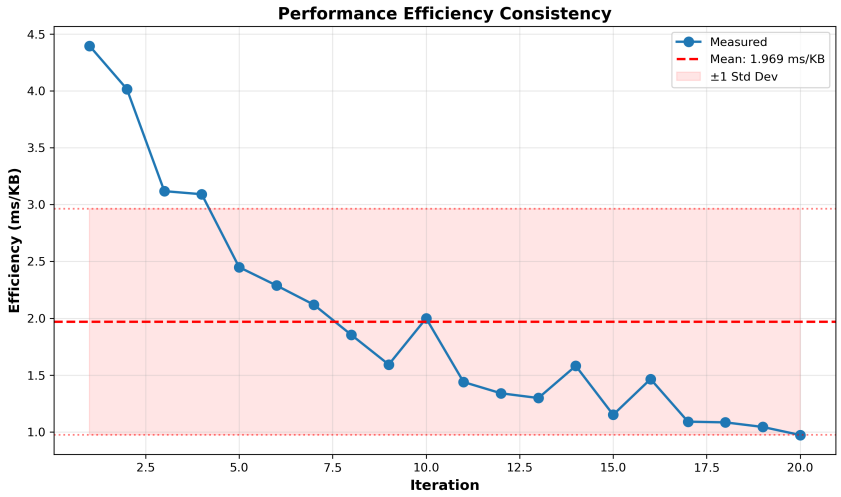

Figure 7.1. Measurement efficiency convergence across network complexity iterations

The convergence to a consistent efficiency metric indicates that the measurement module's performance becomes predictable and linearly scalable once configuration size exceeds approximately 180 KB. This characteristic ensures that deployment in production networks with substantial configuration complexity will exhibit stable, deterministic measurement performance.

For the largest tested configuration (1088.2 KB), the measurement completes in approximately one second, maintaining compatibility with the ten-second measurement interval configured in the deployment. This performance margin ensures that even complex network configurations can be continuously monitored without measurement cycles overlapping or accumulating processing backlogs. The sub-linear scaling behaviour and efficiency stabilisation demonstrate that the system is suitable for deployment in large-scale network environments where configuration complexity may significantly exceed the tested range.

7.3.2 Impact of Network Status Verification on Attestation Performance

To assess the impact of integrating network-status verification into the Keylime attestation workflow, a custom measurement script was employed to capture end-to-end attestation performance across multiple configurations. The script continuously monitors the progress of the verifier, records precise timing information for each attestation round, and correlates these timings with the size of the network measurement transmitted by the agent. This approach enables a controlled and repeatable evaluation of how increasing the complexity or size of the attested data influences the overall attestation latency.

Each experiment consists of **100 attestation iterations**, repeated for all three configurations considered in this study: the *No Check* baseline, the *Normal* 61.5 KB measurement, and the *Large* 134 KB measurement. For every iteration, the script measures the time between two

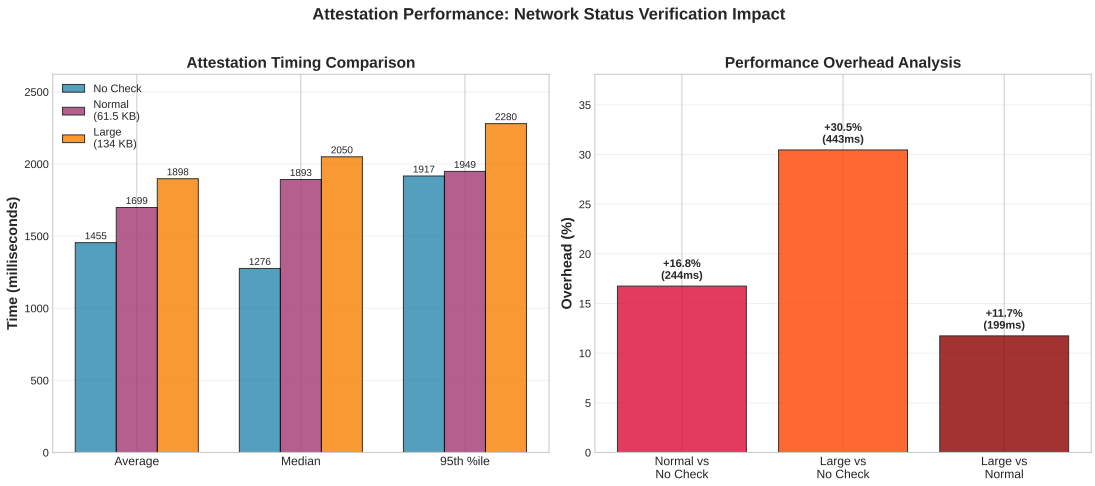

Figure 7.2. Attestation Timing Comparison and Overhead Analysis.

consecutive increments of Keylime's internal `attestation_count`, which serves as a precise indicator that a new round of attestation has been completed. In addition, the script retrieves the measurement size from the agent's network-status endpoint, allowing each datapoint to include both temporal and payload size information. The core logic of this procedure is drawn directly from the implementation in the measurement script.

As attestation performance is influenced by the computational resources available on the host machine, the experiments were executed under controlled conditions. Effort was taken to minimise interference from background processes so that the measured overheads could be attributed reliably to the network-status verification mechanism rather than unrelated system activity. While resource constraints can never be fully eliminated, stabilising the execution environment ensures a consistent baseline for comparing the overhead introduced by different measurement sizes.

Methodology Summary

The evaluation procedure follows the steps below:

- **Agent Registration**: The script registers the agent with the Keylime verifier, initiating continuous attestation.
- **Polling Loop**: For each iteration, the script polls the verifier until the `attestation_count` increments, signalling completion of a new attestation round.
- **Timestamp Measurement**: A high-resolution monotonic timer measures the elapsed time required for that attestation cycle.
- **Measurement Retrieval**: The script queries the agent's measurement endpoint and records the size (in bytes/KB) of the network status evidence.
- **Result Storage**: Timing and measurement data are appended to a dataset for analysis (average, median, percentiles).
- **Repetition**: Steps are repeated until **100 successful attestations** are gathered for the current configuration.
- **Cleanup**: The agent is removed from the verifier to ensure no residual state affects subsequent tests.

The experimental results highlight the quantitative effect of introducing network-status verification inside the Keylime attestation workflow. Three configurations were evaluated: *No Check* (baseline), *Normal* verification with a 61.5 KB measurement, and *Large* verification with a 134 KB measurement. As shown in Figure 7.2, the introduction of the normal measurement induces a measurable increase in attestation latency, rising from an average of 1455 ms in the baseline to 1699 ms, corresponding to a 16.8 percent overhead. When the larger network dataset is used, the average attestation time further increases to 1898 ms, for a total overhead of 30.5 percent relative to the baseline. Even when compared to the normal measurement, the large configuration still introduces an additional 11.7 percent overhead, demonstrating that attestation time scales proportionally to the size of the network evidence.

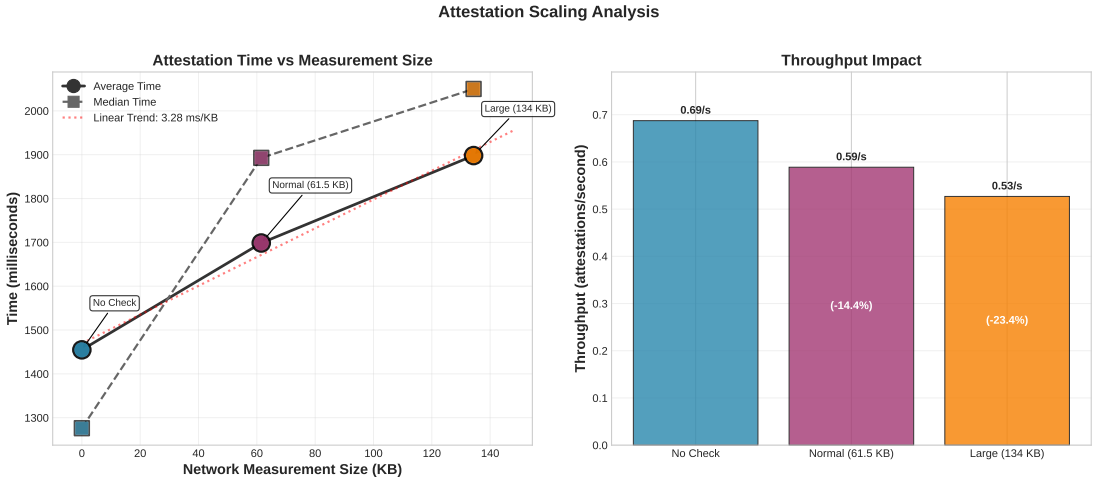

Figure 7.3. Attestation Time Scaling with Measurement Size and Throughput Impact.

The scaling analysis in Figure 7.3 reinforces this linear relationship. By sweeping the measurement size from 0 KB to roughly 140 KB, the attestation time exhibits a clear linear trend of approximately 3.28 ms per kilobyte, confirming that the dominant contributor to the overhead is the increasing size of the network evidence. This behaviour is also visible in the throughput degradation: the baseline achieves 0.69 attestations per second, which decreases to 0.59/s (a 14.4 percent reduction) under the normal measurement size and falls to 0.53/s (a 23.4 percent reduction) with the larger dataset. Overall, the results show that while network-status verification introduces non-trivial overhead, its linear and predictable scaling makes it suitable for continuous attestation scenarios, aiding performance planning and resource provisioning.

Chapter 8

Conclusion and Future Work

This thesis has presented a complete framework for network path attestation that integrates Software for Open Networking in the Cloud (SONiC), Trusted Platform Module (TPM) hardware, and the Keylime remote attestation platform in order to strengthen the security and integrity of network infrastructure. The work demonstrates that hardware-rooted trust mechanisms, traditionally applied to host systems, can be extended to network devices to provide verifiable guarantees regarding router configuration state.

A primary contribution of this research is the design of a custom measurement module capable of capturing a detailed representation of network configuration directly from the Linux kernel using netlink interfaces. Through filtering and normalisation, the module excludes volatile runtime data while retaining stable configuration elements such as interfaces, routing tables, firewall rules, VLANs, tunnels, and bridging information. The canonical JSON encoding ensures deterministic and byte-identical measurements, allowing TPM-backed attestation to operate reliably. By extending PCR-12 with the measurement output, the system binds configuration state to hardware-protected trust roots and produces cryptographic evidence of integrity.

A second contribution lies in extending the TPM attestation workflow to incorporate semantic correlation of configuration changes. Rather than simply indicating a difference from the baseline, the enhanced Keylime Verifier retrieves structured information that identifies the specific network elements that were modified. This significantly improves the interpretability of attestation failures and supports more informed and efficient security decision-making.

A third contribution is the integration of this measurement pipeline into a virtualised SONiC environment, providing a reproducible platform for experimentation. The architecture separates continuous measurement from on-demand TPM verification: the measurement module operates on a ten-second cycle after a two-minute stabilisation period, while Keylime performs verification independently. This avoids bottlenecks and supports scalable deployment.

The experimental evaluation confirmed that the system detects unauthorised configuration changes within the measurement interval. Tests involving VLAN creation, routing modifications, firewall rule updates, and tunnel configuration all triggered PCR divergences and resulted in the attester being classified as untrusted. The Verifier's ability to retrieve semantic change information further enhanced the clarity of attestation results.

Performance testing demonstrated sub-linear scaling with respect to configuration size. As measurement size increased from 82.5 KB to 1088.2 KB, execution time rose from 362.7 ms to 1059.1 ms, while efficiency improved from 4.395 ms/KB to 0.973 ms/KB. This indicates that fixed overheads dominate at low complexity, while variable processing costs scale efficiently. Efficiency stabilised around 1.969 ms/KB beyond 180 KB, showing predictable performance suitable for large network deployments. Even at the highest tested configuration size, measurement completed well within the ten-second interval, ensuring that periodic monitoring can be sustained without delays or cycle overlap.

Additional performance evaluation focusing on attestation overhead showed that integrating network-status verification introduces a measurable but controlled cost. Baseline attestation

averaged 1455 ms, compared with 1699 ms for a 61.5 KB measurement and 1898 ms for a 134 KB measurement. The increase reflects a linear relationship of roughly 3.28 ms/KB. Throughput decreased from 0.69 to 0.53 attestations per second across configurations, but remained within acceptable operational bounds for continuous monitoring.

While the framework demonstrates robust performance, several limitations remain. It measures static configuration rather than dynamic routing state, leaving transient misconfigurations or runtime behaviour undetected. Extending the scope to include routing protocol state, such as BGP session parameters and forwarding tables, would provide stronger assurances but poses challenges in managing dynamic and legitimate changes. In addition, experiments relied on virtualised TPMs, which do not fully reproduce the security characteristics of discrete hardware modules.

The system performs intra-device attestation but does not implement the full Trusted Path Routing model described in IETF specifications. Key elements such as the Stamped Passport mechanism, where routers mutually attest one another using fresh randomness and Verifier-signed results, remain future work. The current implementation also does not construct end-to-end trusted paths or explore inter-domain attestation across autonomous systems.

Several promising research directions arise from this work. Implementing full Trusted Path Routing with Stamped Passports would enable cryptographically verifiable adjacencies. Integrating link-layer authentication mechanisms, such as IEEE 802.1X or MACsec, would provide fresh entropy for secure adjacency establishment. Extending measurements to include routing protocol state would improve security against routing manipulation but would require advanced policy logic. Further work could also investigate distributed Verifier architectures with redundancy or consensus, as well as trusted execution environments for Verifier logic. Inter-domain attestation remains an open problem requiring compatibility with global routing protocols. Integration with Proof of Transit mechanisms offers another avenue to verify both router trust and actual packet traversal behaviour.

Bibliography

[1] H. Birkholz, E. Voit, P. C. Liu, D. Lopez, and M. Chen, "Trusted Path Routing", Internet-Draft draft-voit-rats-trustworthy-path-routing-12, Internet Engineering Task Force, July 2025. Work in Progress

[2] Sonic Foundation, https://sonicfoundation.dev/

[3] A. Martin, "The Ten Page Introduction to Trusted Computing", IEEE Security & Privacy, vol. 6, no. 6, 2008, pp. 74–84, DOI 10.1109/MSP.2008.147

[4] Trusted Computing Group, "Trusted Platform Module 2.0 Library Part 0: Introduction", 2025, https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-0-Version-184_pub.pdf

[5] F. B. Schneider, ed., "Trust in Cyberspace", National Academy Press, 1998, ISBN: 9780309173988

[6] Trusted Computing Group, "Trusted Platform Module 2.0 Library Part 1: Architecture", 2025, https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-1_Architecture-V185-RC2_10July2025.pdf

[7] Trusted Computing Group, "TCG PC Client Platform Firmware Profile Specification", 2019, https://trustedcomputinggroup.org/wp-content/uploads/TCG_PCClient_PFP_r1p05_05_3feb20.pdf

[8] Trusted Computing Group, "TPM 2.0 Keys for Device Identity and Attestation", 2019, https://trustedcomputinggroup.org/wp-content/uploads/TPM-2p0-Keys-for-Device-Identity-and-Attestation_v1_r12_pub10082021.pdf

[9] IEEE, "IEEE Standard for Local and Metropolitan Area Networks - Secure Device Identity", IEEE Std 802.1AR-2018 (Revision of IEEE Std 802.1AR-2009), 2018, pp. 1–73, DOI 10.1109/IEEESTD.2018.8423794

[10] W. Arthur, D. Challener, and K. Goldman, "A Practical Guide to TPM 2.0: Using the New Trusted Platform Module in the New Age of Security", Apress, 2015, ISBN: 978-1-4302-6584-9

[11] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of Remote Attestation", International Journal of Information Security, vol. 10, no. 2, 2011, pp. 63–81, DOI 10.1007/s10207-011-0124-7

[12] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan, "Remote ATtestation procedureS (RATS) Architecture." RFC-9334, January 2023, DOI 10.17487/RFC9334

[13] N. Schear, P. T. Cable, T. M. Moyer, B. Richard, and R. Rudd, "Bootstrapping and maintaining trust in the cloud", Proceedings of the 32nd Annual Conference on Computer Security Applications, New York (USA), December 5, 2016, pp. 65–77, DOI 10.1145/2991079.2991104

[14] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2." RFC-5246, August 2008, DOI 10.17487/RFC5246

[15] P. Psenak, S. Hegde, C. Filsfils, K. Talaulikar, and A. Gulko, "IGP Flexible Algorithm." RFC 9350, February 2023, DOI 10.17487/RFC9350

[16] F. Brockners, S. Bhandari, T. Mizrahi, S. Dara, and S. Youell, "Proof of Transit", Internet-Draft draft-ietf-sfc-proof-of-transit-08, Internet Engineering Task Force, November 2020. Work in Progress

[17] T. Hara and M. Sasabe, "eBPF-Based Ordered Proof of Transit for Trustworthy Service Function Chaining", IEEE Transactions on Network and Service Management, vol. 22, no. 4, 2025, pp. 3138–3149, DOI 10.1109/TNSM.2025.3550333

[18] A. Perrig, P. Szalachowski, R. M. Reischuk, and L. Chuat, "SCION: A Secure Internet Architecture", Springer Publishing Company, Incorporated, 2017

[19] D. Naylor, M. K. Mukerjee, P. Agyapong, R. Grandl, R. Kang, M. Machado, S. Brown, C. Doucette, H.-C. Hsiao, D. Han, T. H.-J. Kim, H. Lim, C. Ovon, D. Zhou, S. B. Lee, Y.-H. Lin, C. Stuart, D. Barrett, A. Akella, D. Andersen, J. Byers, L. Dabbish, M. Kaminsky, S. Kiesler, J. Peha, A. Perrig, S. Seshan, M. Sirbu, and P. Steenkiste, "XIA: architecting a more trustworthy and evolvable internet", SIGCOMM Comput. Commun. Rev., vol. 44, July 2014, pp. 50–57, DOI 10.1145/2656877.2656885

[20] C. Krähenbühl, M. Wyss, D. Basin, V. Lenders, A. Perrig, and M. Strohmeier, "FABRID: Flexible Attestation-Based Routing for Inter-Domain Networks", Proceedings of the USENIX Security Symposium, Zürich, Switzerland, August 23, 2023

[21] Canonical Ltd., "SONiC: The open source network operating system for modern data centers." Ubuntu Blog, May 2021, Accessed: 2025-08-23

[22] A. AlSabeh, E. Kfoury, J. Crichigno, and E. Bou-Harb, "Leveraging SONiC Functionalities in Disaggregated Network Switches", International Conference on Telecommunications and Signal Processing (TSP), Milan (Italy), July 07-09, 2020, pp. 457–460, DOI 10.1109/TSP49548.2020.9163508

[23] Redis, https://redis.io/docs/latest/develop/pubsub/

Appendix A

User Manual

This appendix defines the commands required to set up the environment for the SONiC network path attestation within a SONiC virtual machine emulated with QEMU. The setting consists of an Attester node (SONiC vm) and a Verifier that will reside on the host machine.

A.1 Obtain SONiC

Download a `sonic-vs.img` either by compiling it directly from the official SONiC repository[1] or by retrieving a prebuilt image.

A.2 Start the TPM Emulator

First, create a directory under `/var/lib/swtpm` to store the TPM state:

```
sudo mkdir -p /var/lib/swtpm/mytpm1
sudo swtpm_setup --tpm2 \
  --tpmstate /var/lib/swtpm/mytpm1 \
  --pcr-banks sha256 \
  --create-ek-cert --create-platform-cert \
  --lock-nvram
```

Then start the TPM socket for communication between SONiC and the emulator:

```
sudo swtpm socket --tpm2 \
  --tpmstate dir=/var/lib/swtpm/mytpm1 \
  --ctrl type=unixio,path=/var/lib/swtpm/mytpm1/sock \
  --log level=20
```

Finally, launch the SONiC virtual machine with QEMU and enable port forwarding for host:guest communication:

```
sudo qemu-system-x86_64 \
  -display gtk \
  -accel kvm -m 4096 -boot d -bios bios-256k.bin -boot menu=on \
```

[1] https://github.com/sonic-net/sonic-buildimage

```
  -chardev socket,id=chrtpm,path=/var/lib/swtpm/mytpm1/sock \
  -tpmdev emulator,id=tpm0,chardev=chrtpm \
  -device tpm-tis,tpmdev=tpm0 \
  -netdev user,id=net0,hostfwd=tcp::8087-:8087 \
  -device e1000,netdev=net0 \
  -monitor stdio \
  sonic-vs.img
```

A.3 Attestation Setup in SONiC

A.3.1 TPM2 Tools Installation

Install the required dependencies:

```
sudo apt install libssl-dev swig python3-pip autoconf \
    autoconf-archive libglib2.0-dev libtool pkg-config \
    libjson-c-dev libcurl4-gnutls-dev tpm2-tools tpm2-abrmd
```

A.3.2 Keylime Agent Installation

Install dependencies and the Rust toolchain:

```
apt-get install libclang-dev libssl-dev libtss2-dev \
    libzmq3-dev pkg-config -y
pip install keylime
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
export PATH="$HOME/.cargo/bin:$PATH"
```

Clone and test the Rust Keylime agent:

```
git clone https://github.com/keylime/rust-keylime.git
cd rust-keylime
cargo test
```

A.3.3 Configuring Keylime Agent

Edit the agent configuration file:

```
sudo nano keylime-agent.conf
```

Set the IP addresses in the `[agent]` section:

```
ip = "192.168.122.1"
contact_ip = "192.168.122.1"
registrar_ip = "192.168.60.1"
```

Create required directories and files with appropriate permissions:

```
sudo touch /var/lib/keylime/agent_data.json
sudo chmod 660 /var/lib/keylime/agent_data.json
sudo chown keylime:tss /var/lib/keylime/agent_data.json
sudo chmod 750 /var/lib/keylime
sudo chown keylime:tss /var/lib/keylime
sudo chmod 644 /var/lib/keylime/cv_ca/cacert.crt
sudo chown keylime:tss /var/lib/keylime/cv_ca/cacert.crt
sudo chmod 750 /var/lib/keylime/cv_ca
sudo chown keylime:tss /var/lib/keylime/cv_ca
```

A.3.4 Network Measurement System Setup

Download the measurement scripts:

```
wget https://[repository]/sonic-kernel-measure.py
wget https://[repository]/sonic-measure-setup.sh
wget https://[repository]/sonic_server.py
wget https://[repository]/sonic-kernel-measure.service
wget https://[repository]/sonic-kernel-measure.timer
chmod +x sonic-measure-setup.sh
```

Run the installer to configure the measurement system:

```
sudo ./sonic-measure-setup.sh
```

A.3.5 Starting the Measurement Server

Install Flask and start the measurement server:

```
pip3 install flask requests
sudo python3 /usr/local/bin/sonic_server.py &
```

A.3.6 Running the Keylime Agent

Launch the Keylime agent:

```
RUST_LOG=keylime_agent=trace cargo run --bin keylime_agent
```

A.4 Verifier Machine

A.4.1 Keylime Installation from Source

Clone the Keylime repository:

```
git clone https://github.com/keylime/keylime.git
cd keylime
```

Copy the modified components for network attestation:

```
cp /path/to/modified/cloud_verifier_tornado.py keylime/
cp /path/to/modified/cv_database.sqlite keylime/
```

Run the installer:

```
sudo ./installer.sh
```

A.4.2 Configuring Verifier and Registrar

Edit the verifier configuration:

```
sudo nano /etc/keylime/verifier.conf
```

Set the SONiC agent IP address:

```
[cloud_verifier]
cloudverifier_ip = "0.0.0.0"
cloudverifier_port = 8881
agent_ip = "<SONIC_MACHINE_IP>"
agent_port = 9002
```

Edit the registrar configuration:

```
sudo nano /etc/keylime/registrar.conf
```

Set the agent address:

```
[registrar]
registrar_ip = "0.0.0.0"
registrar_port = 8890
agent_ip = "<SONIC_MACHINE_IP>"
```

A.4.3 Starting the Services

Enable and start the Keylime services:

```
sudo systemctl start keylime_registrar
sudo systemctl start keylime_verifier
sudo systemctl enable keylime_registrar
sudo systemctl enable keylime_verifier
```

Verify service status:

```
sudo systemctl status keylime_registrar
sudo systemctl status keylime_verifier
```

Note: Ensure that the firewall allows connections on ports 8881 (Verifier) and 8890 (Registrar) from the SONiC machine.

A.5 Managing Attestation with Keylime Tenant

The Keylime tenant provides the command-line interface for managing agents and their attestation policies. This section describes the essential commands for adding, monitoring, updating, and removing agents from the attestation framework.

A.5.1 Adding an Agent

To add an agent to the verifier with a custom TPM policy:

```
sudo keylime_tenant -c add \
  -t <AGENT_IP> \
  -u <AGENT_UUID> \
  --tpm_policy '{"12": "<PCR12_HASH>"}'
```

The `--tpm_policy` parameter specifies the expected PCR values. PCR 12 is used for network configuration attestation. Replace `<PCR12_HASH>` with the expected SHA-256 hash of the network state.

A.5.2 Monitoring Agent Status

To list all agents currently registered with the verifier:

```
sudo keylime_tenant -c cvlist
```

To check the attestation status of a specific agent:

```
sudo keylime_tenant -c cvstatus -u <AGENT_UUID>
```

A.5.3 Updating Agent Policy

To update the attestation policy for an existing agent:

```
sudo keylime_tenant -c update \
  -t <AGENT_IP> \
  -u <AGENT_UUID> \
  --tpm_policy '{"12": "<NEW_PCR12_HASH>"}'
```

This command is used when the expected network configuration changes and a new PCR value needs to be validated.

A.5.4 Removing an Agent

To remove an agent from attestation monitoring:

```
sudo keylime_tenant -c delete \
  -t <AGENT_IP> \
  -u <AGENT_UUID>
```

Appendix B

Developer Manual

This manual documents the implementation of network attestation system for SONiC integrated with Keylime. The system extends TPM-based attestation to include runtime network configuration state verification.

B.0.1 Architecture Overview

The attestation system consists of three main components running on the agent machine and one integration with the Keylime verifier:

1. `sonic-kernel-measure.py`: Collects network state from the kernel
2. `sonic_server.py`: REST API server serving measurements
3. `sonic_verifier.py`: Client tool for testing and verification
4. `cloud_verifier_tornado.py`: Keylime verifier integration

B.0.2 Network Measurement Collection

The `sonic-kernel-measure.py` script captures complete network and firewall configuration using netlink sockets (via pyroute2) and iptables commands. The script filters all dynamic data to ensure measurement stability for attestation purposes.

Data Collection Process

The measurement script collects network state using pyroute2's IPRoute interface:

```
with IPRoute() as ip:
    kernel_state = {
        "links": collect_links(ip),
        "addresses": collect_addresses(ip),
        "routes": collect_routes(ip),
        "routing_rules": collect_rules(ip),
        "vlans": collect_vlan_info(ip),
        "tunnels": collect_tunnel_info(ip)
    }

# Collect firewall rules separately
kernel_state["iptables"] = collect_iptables_rules()
kernel_state["ip6tables"] = collect_ip6tables_rules()
```

Dynamic Field Filtering

To ensure hash stability, the script removes all dynamic fields from netlink attributes:

```
DYNAMIC_ATTRS = {
    'RTA_CACHEINFO', # Routing cache statistics
    'IFLA_STATS', # Interface packet counters
    'IFLA_STATS64', # 64-bit interface statistics
    'IFA_CACHEINFO', # Address validation lifetimes
}

def normalize_attrs_list(attrs_list):
    """Convert attrs tuples to sorted dictionaries,
    filtering dynamic fields"""
    return sorted([
        {'key': k, 'value': sanitize_value(v)}
        for k, v in attrs_list
        if k not in DYNAMIC_ATTRS
    ], key=lambda x: x['key'])
```

Firewall Rules Collection

The script collects iptables rules using `iptables-save` and removes packet/byte counters:

```
def remove_counters(rule_spec):
    """Remove packet/byte counters from iptables rules.
    Counters appear in format: -c 123 456 before the rule"""
    if rule_spec.startswith('-c '):
        parts = rule_spec.split(None, 3)
        if len(parts) >= 4:
            return parts[3]
        else:
            return ''
    return rule_spec

def collect_iptables_rules():
    """Collect iptables rules for all standard tables"""
    tables = ['filter', 'nat', 'mangle', 'raw', 'security']
    iptables_data = {}

    for table in tables:
        result = subprocess.run(
            ['iptables-save', '-t', table],
            capture_output=True, text=True, timeout=5
        )

        if result.returncode == 0:
            rules = parse_iptables_save(result.stdout, table)
            iptables_data[table] = rules

    return iptables_data
```

Hash Computation and PCR Extension

After collecting the network state, the script computes a SHA256 hash and extends TPM PCR 12:

```
def compute_hash(data: dict) -> str:
    """Compute SHA256 hash of JSON data in canonical form"""
    canonical_json = json.dumps(data, sort_keys=True,
                                separators=(',', ':'))
    hash_obj = hashlib.sha256(canonical_json.encode('utf-8'))
    return hash_obj.hexdigest()

def extend_pcr(pcr_index: int, hash_value: str) -> bool:
    """Extend TPM PCR with the given hash value"""
    tpm_available, tpm2_pcrextend = check_tpm_available()
    if not tpm_available:
        return False

    pcr_spec = f"{pcr_index}:sha256={hash_value}"
    result = subprocess.run(
        [tpm2_pcrextend, pcr_spec],
        capture_output=True, text=True, timeout=10
    )
    return result.returncode == 0
```

The script extends PCR 12 only if the hash has changed or no previous measurement exists:

```
# Read previous hash
prev_hash = None
if prev_hash_file.exists():
    with prev_hash_file.open('r') as f:
        prev_hash = f.readline().strip()

if prev_hash is None:
    print("[INFO] No previous hash found; extending PCR.")
    pcr_extended = extend_pcr(PCR_INDEX, data_hash)
elif prev_hash == data_hash:
    print("[INFO] Hash unchanged; skipping PCR extension.")
    pcr_extended = False
else:
    print("[INFO] Hash changed; extending PCR.")
    pcr_extended = extend_pcr(PCR_INDEX, data_hash)
```

Output Files

The measurement script writes three files to `OUT_DIR` (default: `/tmp/attest/kernel/current`):

- `kernel_network.json`: Complete network state data
- `meta.json`: Measurement metadata including hash and PCR info
- `measurement_hash.txt`: SHA256 hash for quick reference

Example `meta.json` structure:

```
{
    "hostname": "sonic-agent",
    "timestamp_epoch_ms": 1732464000000,
    "timestamp_iso_utc": "2024-11-24T12:00:00Z",
    "time_measure": 362,
    "collection_method": "pyroute2/netlink + iptables",
    "measurement_mode": "stable_config_only",
    "data_hash_sha256": "a3f5e9c...",
    "pcr_index": 12,
    "pcr_extended": true
}
```

B.0.3 REST API Server

The `sonic_server.py` Flask application executes the measurement script on-demand and serves results via HTTP. It runs on port 5000 alongside the Keylime agent.

API Endpoints

The server exposes three endpoints:

GET /measurement Execute measurement and return network state:

```
@app.route('/measurement', methods=['GET'])
def get_measurement():
    try:
        kernel_state, meta, duration_ms = run_measurement()

        response = {
            'kernel_state': kernel_state,
            'metadata': meta,
            'server_timestamp': datetime.now().isoformat(),
            'measurement_duration_ms': duration_ms
        }
        return jsonify(response)

    except Exception as e:
        return jsonify({
            'error': str(e),
            'timestamp': datetime.now().isoformat()
        }), 500
```

GET /info Server configuration information:

```
@app.route('/info', methods=['GET'])
def server_info():
    return jsonify({
        'measurement_script': MEASUREMENT_SCRIPT,
        'server_port': SERVER_PORT
    })
```

Measurement Execution

The core function executes the measurement script and reads the generated files:

```
def run_measurement():
    """Execute measurement script and return
    (kernel_state, meta, duration_ms)"""
    start_time = time.perf_counter()

    env = os.environ.copy()
    env['OUT_DIR'] = '/tmp/attest/kernel/current'
    Path(env['OUT_DIR']).mkdir(parents=True, exist_ok=True)

    # Execute measurement script
    result = subprocess.run(
        ['python3', MEASUREMENT_SCRIPT],
        env=env, capture_output=True,
        text=True, timeout=30
    )

    if result.returncode != 0:
        raise Exception(f"Measurement failed: {result.stderr}")

    # Read generated files
    kernel_file = Path(env['OUT_DIR']) / 'kernel_network.json'
    meta_file = Path(env['OUT_DIR']) / 'meta.json'

    with kernel_file.open('r') as f:
        kernel_state = json.load(f)
    with meta_file.open('r') as f:
        meta = json.load(f)

    duration_ms = round((time.perf_counter() - start_time) * 1000, 2)
    return kernel_state, meta, duration_ms
```

B.0.4 Verification Client

The `sonic_verifier.py` script provides a standalone client for testing the measurement server. It can be used independently or as a reference for integration.

SONICVerifier Class

The main class provides methods for interacting with the measurement server:

```
class SONICVerifier:
    def __init__(self, server_url: str, timeout: int = 30):
        self.server_url = server_url.rstrip('/')
        self.timeout = timeout

    def check_health(self) -> Dict[str, Any]:
        """Check server health status"""
        response = requests.get(
            f"{self.server_url}/health",
            timeout=self.timeout
        )
```

```
        response.raise_for_status()
        return response.json()

    def get_measurement(self, force: bool = False) -> Optional[Dict]:
        """Request measurement from server"""
        response = requests.get(
            f"{self.server_url}/measurement",
            timeout=self.timeout
        )
        response.raise_for_status()
        return response.json()

    def verify_measurement(self, data: Dict[str, Any]) -> bool:
        """Verify measurement data integrity"""
        # Check required fields
        required_fields = ['kernel_state', 'metadata']
        for field in required_fields:
            if field not in data:
                return False

        # Check kernel_state structure
        kernel_state = data['kernel_state']
        expected_keys = ['links', 'addresses', 'routes',
                         'routing_rules', 'vlans', 'tunnels','iptables', '
                             ip6tables']

        for key in expected_keys:
            if key not in kernel_state:
                return False

        # Check metadata has hash
        if 'data_hash_sha256' not in data['metadata']:
            return False

        return True
```

Command-Line Usage

The verifier can be invoked from the command line with various options:

```
# Basic usage
python3 sonic_verifier.py --server http://192.168.1.100:5000

# Health check only
python3 sonic_verifier.py --server http://localhost:5000 --health

# Save measurement to file
python3 sonic_verifier.py --output measurement.json

# Verify only (no summary display)
python3 sonic_verifier.py --verify-only
```

B.0.5 Keylime Verifier Integration

The Keylime verifier has been modified to retrieve network measurements during the attestation process. The changes are in the `cloud_verifier_tornado.py` file.

New Network Status Retrieval Function

A new asynchronous function contacts the SONIC server on the agent machine:

```
async def invoke_get_network_status(
    agent: Dict[str, Any],
    custom_port: int = 5000,
    custom_endpoint: str = "/measurement",
    timeout: float = DEFAULT_TIMEOUT,
) -> Optional[Dict[str, Any]]:
    """Contact agent's SONIC endpoint to retrieve
    network status information"""
    try:
        url = f"http://{agent['ip']}:{custom_port}{custom_endpoint}"

        logger.info(
            "Contacting agent %s network status endpoint: %s",
            agent['agent_id'], url
        )

        res = tornado_requests.request(
            "GET", url, timeout=timeout,
        )
        response = await res

        if response.status_code == 200:
            network_status = json.loads(response.body)
            logger.info(
                "Successfully retrieved network status from agent %s",
                agent['agent_id']
            )
            return network_status
        else:
            logger.warning(
                "Network status request failed for agent %s: HTTP %d",
                agent['agent_id'], response.status_code
            )
            return None

    except Exception as e:
        logger.error(
            "Exception retrieving network status from agent %s: %s",
            agent['agent_id'], str(e)
        )
        return None
```

Integration with Quote Retrieval

The `invoke_get_quote()` function has been extended to retrieve network status after successfully obtaining the TPM quote:

```
async def invoke_get_quote(agent: Dict[str, Any], ...) -> None:
    # ... existing code to get TPM quote ...

    else: # Success case - quote retrieved successfully
        try:
            json_response = json.loads(response.body)

            # NEW: Retrieve network status after successful quote
            network_status = await invoke_get_network_status(
                agent,
                custom_port=5000,
                custom_endpoint="/measurement",
                timeout=timeout
            )

            if network_status:
                # Store in agent data for verification
                agent['network_status'] = network_status
                logger.info(
                    "Network status retrieved and stored for agent %s",
                    agent['agent_id']
                )
            else:
                logger.warning(
                    "Could not retrieve network status for agent %s",
                    agent['agent_id']
                )

            # Continue with existing quote verification...
            if "provide_V" not in agent:
                agent["provide_V"] = True

            # ... rest of existing verification code ...
```

The network status data is stored in the agent dictionary and can be accessed during policy verification. The expected hash from the measurement metadata can be compared against the TPM PCR 12 value to verify network configuration integrity:

```
# Extract expected hash from measurement
expected_hash = agent['network_status']['metadata']['data_hash_sha256']

# Compare against policy or PCR value
# Policy verification logic here...
```