OPENPCC

A Standard for Private and Secure Inference in the Cloud

Authored by CONFIDENT SECURITY

OpenPCC

Confident Security Technical Staff

November 2025

This paper introduces OpenPCC, a security and privacy standard for providing provably-confidential cloud compute to clients. Designed for multi-modal AI models and other inference engines, OpenPCC describes a system for inference on private or otherwise sensitive data such that inputs and outputs remain completely hidden from all participants. The standard also specifies an anonymization scheme to protect the privacy of customers from side-channel leaks, including API timing, payment side channels, and metadata analysis. *A fully-featured Apache 2.0 open-source implementation written in Go accompanies this document.*

Available at `https://github.com/openpcc/openpcc`

1. Cloud Inference (In)security

As AI becomes more powerful and accessible, the stakes around data privacy and protection are higher than ever. For instance, a single employee, seeking to leverage AI's ability to read and understand a PDF, can easily upload a confidential document to an LLM and, in doing so, mistakenly expose PII or trade secrets. Worse, these private data may be stored and used to train and improve future models, eroding any data-related competitive advantages an enterprise has.

Data privacy risks are not new, but AI's capabilities and prevalence amplify them dramatically. These risks are no longer hypothetical:

- A 2025 survey found that in enterprise environments, 40% of files uploaded into GenAI tools contain PII or PCI data, and 45% of all employees are using GenAI tools.
- Anthropic recently began defaulting to including all user-submitted data in training. On major AI providers today, users must opt-out of data training rather than a more user-friendly opt-in.
- Meta, Grok, Anthropic, and OpenAI have all suffered leaks that exposed hundreds of thousands of very confidential chat logs to the public Internet, and search engines subsequently indexed them.
- In an extremely high-profile case, OpenAI was legally forced to retain "deleted" and "temporary" prompts and responses by a judge, violating user expectations and revealing the capability is indeed possible – the servers retain private prompt data. In this instance, two parties were at risk: the users *and the operator* (OpenAI) due to legal proceedings.

These risks exist in current AI environments because private prompts and responses are available alongside user identifiers that can be linked to names, email addresses, and other personal information. In any system where user prompts and identities are accessible, there is an incentive – and eventual legal compulsion – to log and retain that information. Even when AI providers intend to use stored data only for analytics, customer support, debugging, or training, the only true way to guarantee that prompts and personal data remain - now and in the future - is to ensure they're never accessible to anyone in the first place. If stored, user prompts and identities can be exposed in a security breach by insiders, other users, or even nation-state actors.

The Trust Problem

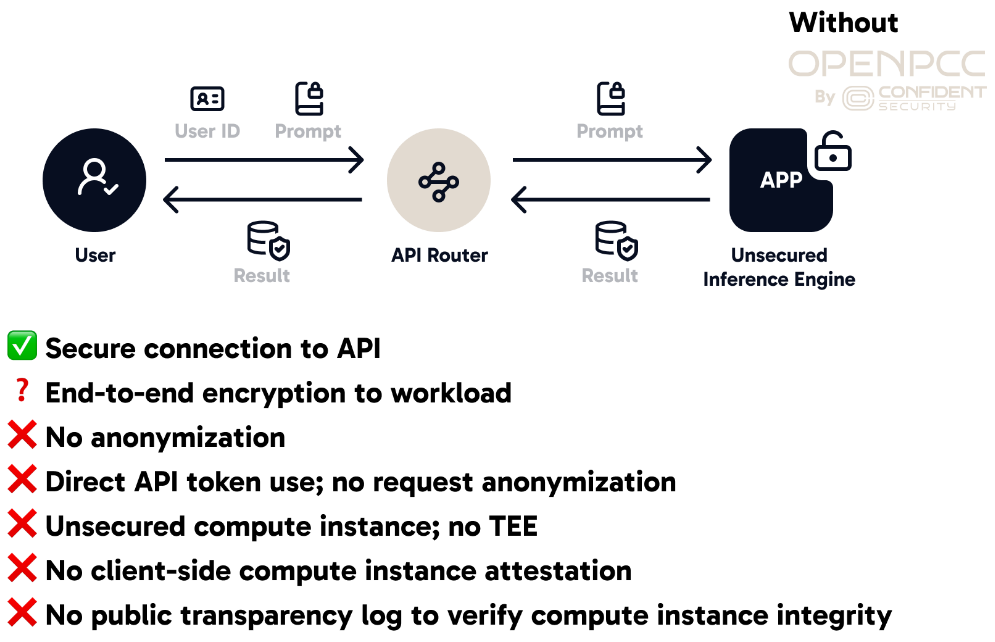
Beneath data privacy concerns lies a fundamental challenge of developing trust among the following parties:

- **Data owners (user)** - do not want to share private or proprietary data but do not have the ability to run the best (proprietary) models themselves nor have the budget to afford AI hardware.

- **Model owners** - do not want to leak or disclose proprietary model weights, may want to use additional proprietary data for training, and must ensure models are used safely.
- **Software operators** - may integrate model weights with an inference engine running on cloud hardware but are really intermediaries assuming all risk.
- **Hardware providers** - lease access to compute resources, are responsible for up-to-date hardware, and control physical access to hardware.

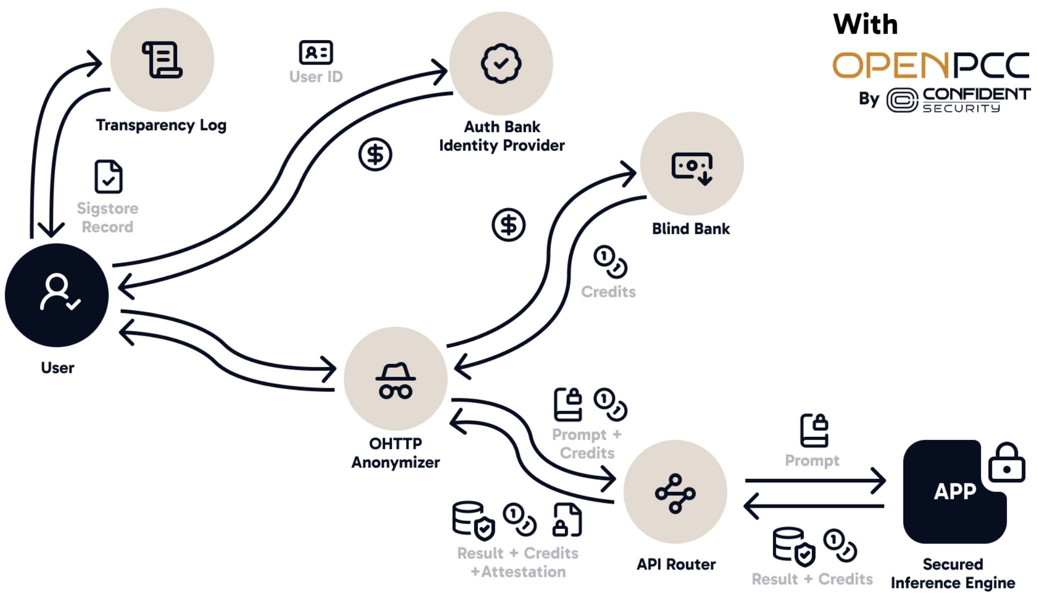
Each party has varying degrees of trust in the others, and any breach by one compromises them all. Note that their incentives are misaligned, but they will all feel the consequences of a data breach!

Approaches to AI data privacy often rely on shifting trust among the parties mentioned above. But so far, these methods have consistently fallen short. Contractual promises will be broken, redaction and obfuscation are incomplete and prone to leaks, and self-hosting/on-premise is prohibitively expensive given the high cost of AI hardware and. Even then, proprietary models often remain inaccessible. The only approach that truly addresses all of these challenges is a combination of full anonymization and full encryption. To guarantee user prompts and responses cannot be subpoenaed, stolen, used for training, or leaked, they must not be stored at all. To guarantee users and their data cannot be identified or targeted, they must remain anonymous.



✅ Secure connection to API
❓ End-to-end encryption to workload
❌ No anonymization
❌ Direct API token use; no request anonymization
❌ Unsecured compute instance; no TEE
❌ No client-side compute instance attestation
❌ No public transparency log to verify compute instance integrity

FIGURE 1. Legacy Inference Providers



✅ **Secure connection to API**
✅ **End-to-end encryption to workload**
✅ **Privacy-aware anonymization scheme**
✅ **Blind single-use token for anonymized requests**
✅ **Hardened compute instance inside of a TEE with a TPM 2.0**
✅ **Compute instance attestation, tied to secure hardware elements**
✅ **Transparency log records to verify attestation bundles**

FIGURE 2. OpenPCC Inference Providers

2. The OpenPCC Standard

OpenPCC is a standard for providing cloud inference while protecting user data from platform actors, intermediaries, and external attackers. Simply put, no party but the client and the inference engine the client chooses can decrypt the content of a compute request or its response. A compute request serves as the central API operation in which a client sends an encrypted inference prompt to a target inference engine hosted on a ComputeNode instance. Inference prompts contain plaintext or plaintext-encoded data submitted by a client to an inference engine, and the inference engine transforms the prompt into a plaintext or plaintext-encoded inference result.

Within the OpenPCC standard, user data includes all private or otherwise sensitive data a user shares with an OpenPCC component. In each case, only the single recipient component may decrypt the submitted ciphertext. Every other component cannot access the user's identity, the user's request, or both. This restriction applies to inference prompts, inference results, selected inference engines, and side channels such as the timing or frequency of API calls. A ComputeNode instance, the terminal endpoint processing a compute request, acts as the sole component that decrypts inference prompts and evaluates them using the inference engine selected by the client.

OpenPCC is inspired by, extends, and improves upon Apple Inc.'s PCC architecture, incorporating a mechanism for anonymized usage tracking while protecting the privacy of customers from side-channel leaks, including API timing, payment side channels, and data metadata analysis, to ensure provably and verifiably private inference. This design ensures that private data used in inference is inaccessible, even to insider threats.

OpenPCC exceeds the current state-of-the-art by satisfying the rigorous requirements of the PCC reference architecture, chaining to secure hardware elements, and protecting user privacy at every stage of the protocol stack. Current solutions provide mechanisms to encrypt prompts, results, and other workloads but fail to protect user identity from de-anonymization due to side-channel analysis or compromise of a single system actor in the platform. Other inference environments isolate workloads within layers of virtualization and fail to anchor secure hardware elements or attestation flow, offering only a subset of the assurances OpenPCC provides.

Previous work

OpenPCC offers several key differences from existing projects like Edgeless, Tinfoil, Project Oak, and RA-TLS implementations. These differences are a consequence of 4 practical requirements:

1. Anonymization to protect end-user privacy is a primary goal

2. Enterprise scale and performance is a priority
3. Maximal interoperability with existing technology stacks is key to adoption
4. Prompt decryption must fail if the evidence at time of decryption does not match the evidence at time of collection. This is a generalization of solving a TOCTOU security flaw. Here, we consider time of evidence collection, time of evidence verification (TOC), time of request encryption, and time of decryption followed by workload execution (TOU).

   We satisfy these requirements by:

- Using OHTTP + blind signatures
- Tying prompt HPKE to a private key resident in a TPM with a Policy that denies access on any change in attested values
- Clients verify evidence asynchronously and encrypt for multiple compute nodes to handle JIT routing and availability decisions
- Only inference nodes are confidential and attested. The rest of the stack does not need attestation (e.g., for routing) and so can use existing high-scale, high-performance software and services (e.g., Cloudflare) that an implementer is already using for working with HTTP, load balancing, databases, etc.

In the following sections, we discuss the full specification and standards.

2.1. System Goals

An OpenPCC-compliant system is designed to accomplish a specific set of interrelated goals that, when taken together, provide complete privacy and security for clients seeking cloud computation on their behalf.

- Stateless computation on personal user data: User data must be inaccessible to all platform actors apart from the inference engine itself.
- Enforceable guarantees: users must be able to analyze and verify the security and privacy guarantees of OpenPCC.
- No privileged runtime access: there must be no privileged access utilities (ssh, serial console) that allow anyone to bypass privacy guarantees.
- Non-targetability: an attacker must not be able to target specific OpenPCC users or de-anonymize user traffic.
- Verifiable transparency: users and security researchers should be able to verify the stated guarantees of OpenPCC.

2.1.1. Stateless Computation on Personal User Data

OpenPCC implementations must prevent persistence of any user data once an inference request is complete. All Compute Request content, including the prompts and results, are decrypted by a single ephemeral process on the ComputeNode instance running in an isolated sandbox environment. Inference engines are isolated from system services and host system processes through Linux kernel security modules (e.g., SELinux) and are restricted to performing file system operations on an ephemeral disk encrypted using a Trusted Platform Module (TPM). This ensures that all inference engine content is inaccessible across boot cycles. Network requests are limited to the requisite set of ports and address ranges necessary for OpenPCC to route inference requests – no user data is made accessible through diagnostic or remote debug services. Inference engines are run in a secure sandbox, encapsulated in a Trusted Execution Environment (TEE) with a TPM 2.0, using a hardened compute image that must be attested and verified before any inference request can be routed to it.

2.1.2. Enforceable Guarantees

Users of the OpenPCC system must be able to enforce the security and privacy constraints that are provided. The hardware security elements of an OpenPCC ComputeNode must sign an attestation bundle, and that bundle is presented to the client. This attestation bundle includes evidence artifacts that verify the hardware, firmware, and software configuration of the compute instance, including proof that the inference model in use matches what the user requested and that the content the user sends to the inference model is inaccessible to any actor other than the inference engine itself.

Before issuing a compute request, the client verifies the attestation bundle presented by each ComputeNode. The client may then enforce their security and privacy constraints by rejecting ComputeNode instances that do not satisfy their criteria.

2.1.3. No Privileged Runtime Access

It is cryptographically verifiable that no third party can access sensitive user data. This data includes prompts, model outputs, and any intermediate data within the inference engine. This constraint dictates the attestation flow, the compute image hardening, and the end-to-end protocol design of OpenPCC. The implementer may provide a set of client-side tooling that allows users to verify this assertion given a set of trust anchors.

OpenPCC considers every actor in the protocol suite untrusted, including (but not limited to) the implementer and any third-party cloud or hosting providers. Each component of the system must be designed to tolerate compromise of its dependent components. Every component should be designed according to the principle of least authority and must only have access to information and operations strictly required to perform its singular function. Implementers are required to go to great lengths to prune all outside interfaces to the compute instance itself (e.g., no SSH or no remote shell access), and these steps must be verifiable in the publicly available compute image in the OpenPCC transparency log. Using the content of an attestation bundle and transparency log records, a client must be able to verify that the desired ComputeNode instance is running without privileged runtime access, making this an enforceable guarantee as well.

2.1.4. Non-Targetability

Non-targetability means that an attacker must not be capable of selectively targeting a user or particular set of users, including but not limited to route users to compromised ComputeNodes. Satisfying non-targetability makes any attack on OpenPCC more difficult for an attacker, as this design eliminates single points of failure and forces an attacker to compromise the entire system, even to target one user.

An anonymous credit system allows users to pay-as-you-go, while detaching their true identity from the Credits withdrawn and any subsequent dispatched inference requests. OpenPCC standardizes this process using RSA Blind Signatures with Public Metadata (RSAPBSSA).

To increase user privacy, OpenPCC operators may offer a multi-tenant environment where a single physical node can serve requests from many users across different models. To preserve security, the system isolates tenants from one another by running each request inside a separate secure sandbox. Additionally, the system routes calls to an inference engine through an Oblivious HTTP (OHTTP) relay. This ensures that network requests remain both anonymized and end-to-end encrypted along their path to and from the inference engine.

These protections combine to ensure an attacker cannot target any specific OpenPCC user to compromise the privacy of their prompts and requests.

2.1.5. Verifiable Transparency

The ComputeNode attestation flow must allow clients to verify that the system securely isolates their allocated computing environment and binds it to a secure hardware root-of-trust, chain-of-trust, and execution environment (TEE + TPM). With this verification, clients can trust that the operating system image, firmware, and userspace applications remain immutable and match the publicly auditable container images, which the system lists in a public transparency log.

2.2. Architecture Overview

With those goals in mind, let's begin to examine the specific components of the OpenPCC standard. OpenPCC uses many IETF standards and publications, including HTTP (RFC

2616), Oblivious HTTP (RFC 9458), Remote Attestation Procedures (RATS) Architecture (RFC 9334), as well as many others. We reference them where possible and implementers SHOULD be familiar with them.

At the highest level, an OpenPCC request ensures both security and anonymity: clients encrypt requests directly to a target set of appropriate ComputeNodes. The client then sends the encrypted requests through an anonymizing proxy, which strips identifying request metadata, through an accounting proxy, which ensures the request includes unspent anonymous compute credits, and lastly, a service proxy, which randomly routes the request arrives to a healthy inference engine. The inference engine can only decrypt the request in the presence of a hardware security element guaranteeing the virtual machine contains no remote access or request logging software. The hardware-attested inference engine then encrypts its response directly to the requesting client's public key, sending the response back through the same proxy chain.

An OpenPCC-compliant system produces fully anonymous inference requests that no one can read or associate with a particular user, even if a component in this system has somehow been compromised. Each proxy component reads only the specific data required for its task, and hides all other request data from itself. The ComputeNode component decrypts the client request only with assistance from a trusted element, and the trusted element ensures that each running machine image remains auditable via a public append-only transparency log.

Let's look briefly at each component of the overall system before we examine in depth how the system provides fully private networking, completely anonymous accounting, and attested and verified secure compute.

2.2.1. Client

The *Client* is an SDK for inclusion in user-facing software and includes the code needed to encode and send the necessary HTTP requests to all other components of the OpenPCC system. Clients can be written in any language and run on any architecture. Example clients are available for Go, Python, JavaScript, and TypeScript.

2.2.2. AuthBank

The *AuthBank* service handles user identity, authentication, and secure payment of user funds into a prepaid balance that a client can later use to authorize compute requests. The AuthBank server converts external funding into OpenPCC Credits for clients. If the client leaves credits unused, they may return OpenPCC Credits to the AuthBank, which then converts the unused credits back into a balance for later use.

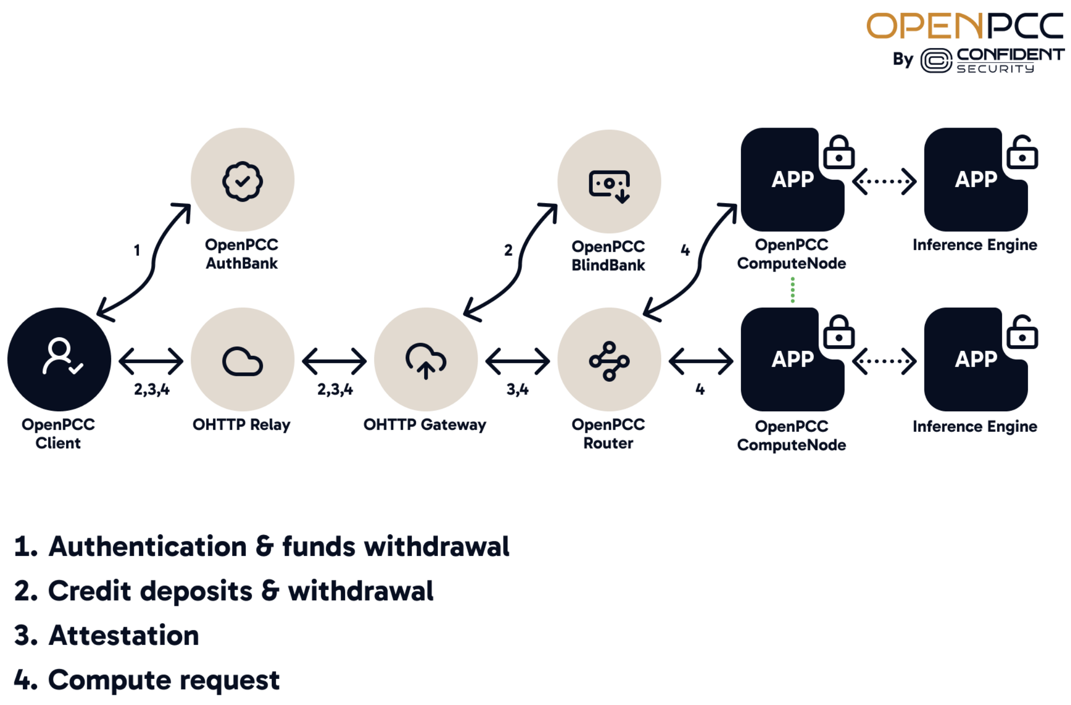


FIGURE 3. OpenPCC Component Architecture

The AuthBank serves as the single component of the OpenPCC system that ties a user's identity to their requests, since accounts and payment are both strongly linked to a specific user identity. For this reason, communication between the client and AuthBank is not typically protected beyond standard HTTP TLS encryption.

In addition to handling user authentication and payments, the AuthBank provides the public keys of the OpenPCC Gateway, allowing clients to encrypt requests such that the Relay operator cannot read them.

2.2.3. Relay

The *Relay* is an HTTP proxy server managed by a third party. By routing all traffic to the Gateway through the Relay, OpenPCC ensures the Gateway can never see user-identifying request metadata. Since the Relay does not possess the keys needed to decrypt the contents of any request it proxies, the Relay operator cannot view request contents.

2.2.4. Gateway

The *Gateway* is a managed service component that accepts proxied OHTTP requests and removes the outermost layer of encryption. The Gateway then forwards the inner request, which carries another layer of encryption, to the intended OpenPCC service, either the BlindBank or Router.

2.2.5. BlindBank

The *BlindBank* service manages OpenPCC Credits tied to ephemeral blind accounts but disconnected from real user identities. The BlindBank allocates ephemeral accounts at a client's request, deposits Credits, and withdraws Credits. By design, the BlindBank server does not maintain any knowledge of real user identities. To protect user identities, all traffic to the BlindBank server(s) routes through the Relay, which strips identifying information from the request(s).

Clients redeem and withdraw Credits from the anonymous bank based only on a (long, unguessable) account token. The BlindBank acts as an intermediary between withdrawing Credits from the auth server (which knows your identity) and communicating with the compute servers. The BlindBank issues Credits to the client as RSA Blind Signatures with Public Metadata (RSAPBSSA).

2.2.6. Router

The Router service dispatches inference requests from a user, thereby consuming Credits. The Router consumes the portion of Credits necessary to perform the computation and returns a refund Credit bundle to the user containing the remaining unused portion of their Credit bundle sent to the Router. The ComputeNode may include the amount of expected refund Credits to verify that the Router issued the correct refund amount. The Router server does not maintain any knowledge of real user identities; therefore, all requests made to the Router server(s) use OHTTP.

2.2.7. ComputeNode

The ComputeNode communicates with the Router to service inference requests from users. The ComputeNode service serves as the final endpoint of an inference request. Each ComputeNode TPM attests to the inference engines and models available on that node. By hosting multiple inference engines and models, up to capacity, ComputeNodes can further protect the anonymity of individual Compute Requests.

2.3. Fully Private Networking

To preserve user privacy, OpenPCC utilizes a cryptographic scheme that both encrypts user data end-to-end and obscures the relationship between a user and an API request.

The protocol must not only protect the prompts and responses exchanged with the inference engine but also potential side channels that could reveal equally sensitive information. The OpenPCC privacy model protects against leakage of:

- Content of any inference request or response
- Connections between a user's identity and a specific inference engine
- Per-user counts for inference requests or other system usage

The system provides this protection through a combined approach: it authorizes inference requests anonymously, routes network requests without attaching any user metadata, and maintains end-to-end encryption between clients and the inference engine that services those requests and returns responses.

2.3.1. Anonymous Authorization via Platform Credits

OpenPCC uses a blind signature scheme on top of route anonymization to disconnect Credits from real user identities. Users spend these Credits as secure bearer tokens to purchase compute resources, and the system issues a refund for any residual balance. This approach shifts the management of Credits to the user and provides fine-grained control of when and how Credits are spent, allowing users to further obfuscate their transactions by adding random delays between payments.

2.3.2. Private Network Requests via Oblivious HTTP

OpenPCC uses Oblivious HTTP, described in RFC 9458, to anonymize all network traffic between the client and OpenPCC services other than AuthBank. OHTTP prevents OSI Layer 1-4 metadata from revealing any association between an API call and a real user identity.

OHTTP requires two endpoints: (1) a relay that acts as a standard HTTP proxy for encrypted requests and removes all identifying request metadata, and (2) a gateway that receives requests from the relay and decrypts their true contents. To generate anonymized requests, a client encrypts its request with the gateway's public key, and then sends that request to the relay. The relay then forwards the request to the gateway. A third party manages the relay, shifting the trust in this component away from the OpenPCC implementers and removing them as a single point of failure in the anonymization scheme.

When a message exits the OHTTP gateway and reaches the target OpenPCC service, the service cannot correlate the payload with a real user identity, as shown in the diagram below.

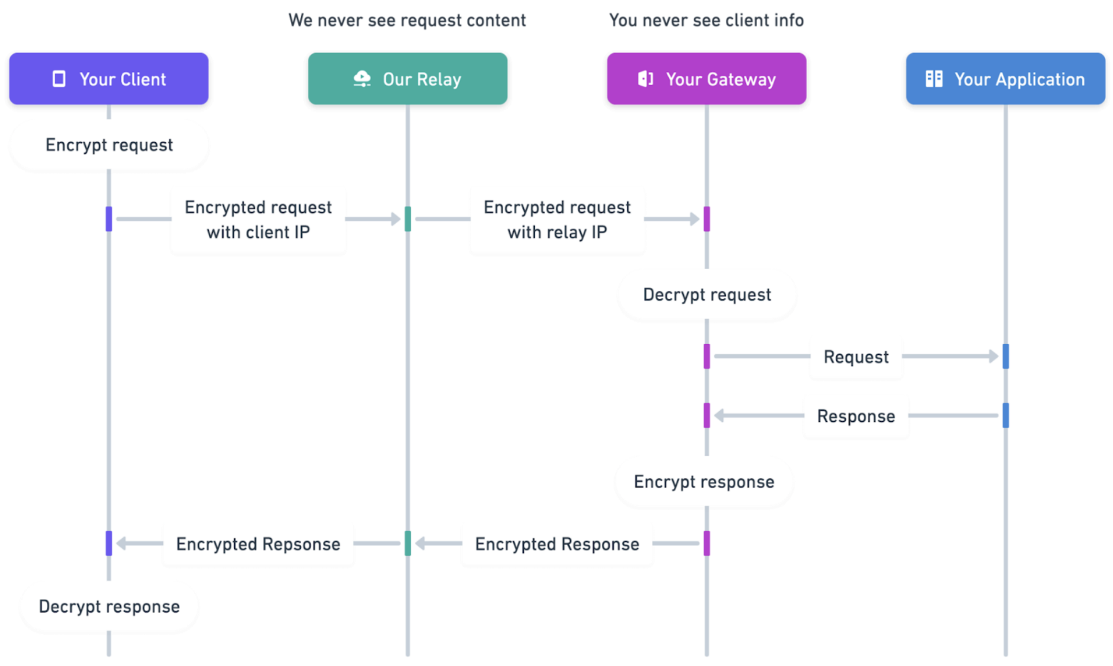


FIGURE 4. Oblivious HTTP Request & Response

Third party OHTTP relay providers have a business incentive in direct opposition to collusion with an implementer or developer of OpenPCC (or any other platform using their OHTTP relays) to de-anonymize traffic, and so de-anonymization of OHTTP traffic is only feasible by compromising both the recipient's managed services and the third party

OHTTP relay infrastructure. As of the initial release of OpenPCC, available OHTTP relay providers include Fastly, Cloudflare, and Oblivious.network.

2.3.3. End-to-End Encryption

To preserve full confidentiality, the Gateway and Router components are unable to decrypt or view the inference request itself. To achieve this requirement, the client and compute node create another layer of encryption – a session key derived through HPKE. The client derives the session key from the public portion of a key in the TPM (the Request Encryption Key or REK), and the private portion of the client’s key. The compute instance attests the REK using the Attestation Key (AK) from the TPM. Both the AK and REK are included in the compute instance's attestation bundle presented to the user. Using the REK and derived session key ensures that no intermediate parties may decrypt the innermost contents – only the user’s client and the compute instance may decrypt the contents of an inference request or response.

The inference request encodes the inference request as BHTTP and encrypts it using a random per-request Data Encryption Key (DEK). This encryption uses an independent AEAD instance compatible with HPKE. Then, the client uses HPKE and a new instance of the same AEAD algorithm to encrypt the DEK of each candidate node's public REK, producing an encapsulated key for each candidate ComputeNode. The client keeps those HPKE contexts and uses them later to decrypt the response.

The Router forwards these encapsulated keys together with the encrypted user request to the candidate ComputeNode that it chooses to fulfill the request. Upon receiving the encrypted request from the Router, the ComputeNode uses the private REK resident on its TPM to decrypt the DEK using HPKE. The ComputeNode then uses the DEK to decrypt the user request, decodes and validates the request, and dispatches it to the inference engine.

The inference engine encodes the response with BHTTP and encrypts it with a key derived from the HPKE context. As part of the response back to the client, the Compute Node adds its identity to the response metadata. When the client receives the response, it locates the HPKE context used to encrypt the DEK and matches it to the identity in the response metadata. The client then derives the response encryption key from that HPKE context, and uses that key to decrypt and decode the response before presenting it to the user.

2.4. Anonymous Accounting

The OpenPCC system uses a pay-as-you-go model to charge users. It implements this model through an internal currency system where users purchase tokens and deposit them into an anonymized banking account. To purchase inference requests, users withdraw Credits from the bank and spend those Credits on inference. When users have unused Credits, the system refunds them, and users can exchange those Credits back for real-world funds.

2.4.1. Banking Flow

Credits. Credits serve as the in-platform currency in OpenPCC. Each Credit contains two parts: a private component and public metadata. The public metadata encodes the Credit's value, while the private component includes a nonce and a quantized timestamp. The nonce guarantees that each Credit is unique, and the timestamp limits the Credit's lifespan. OpenPCC maintains a centralized record of used nonces to track when users redeem Credits. The limited lifespan prevents the system from having to store all nonces indefinitely. Each Credit represents its value as a floating-point number within the range 0 to 33,285,996,544 (31×2^3), and can express any number in this range with no more than 6.25% relative error.

In a standard blind signature process, the signer remains unaware of the content they sign. To address this limitation, OpenPCC uses the RSA Blind Signatures with Public Metadata (RSAPBSSA) scheme, which allows the signer to encode certain data publicly within the signature.

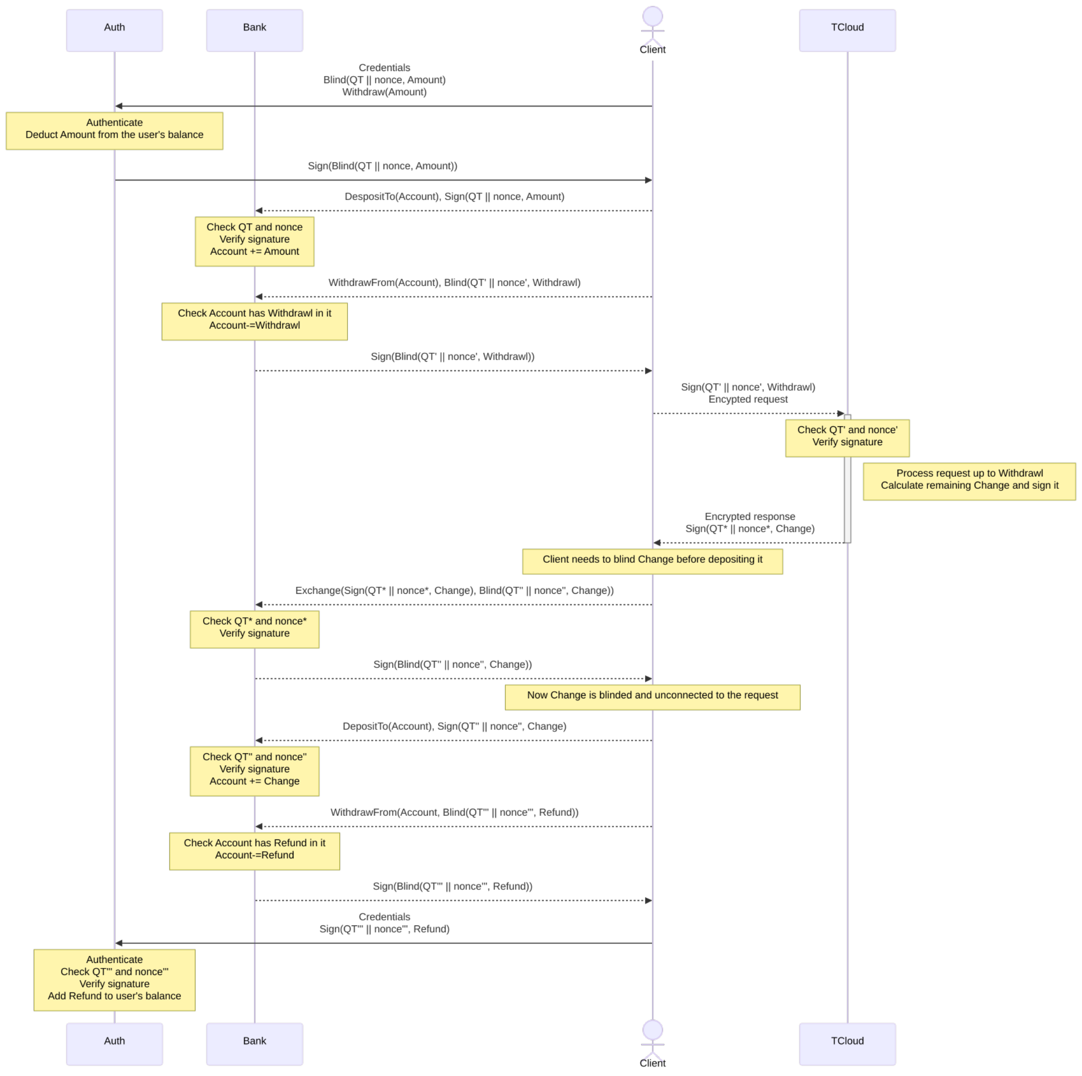
In this signing process, the user blinds the Credit before sending it to the relevant server for signing. After the server returns the blinded and signed Credit, the user unblinds it. The unblinded Credit includes a signature that the currency public key can verify, but no one can link the unblinded signed Credit to the original blinded Credit that the server signed. The public metadata extension (RSAPBSSA) lets Credits include an unblinded value associated with them.

The following sequence diagram outlines the complete banking flow. Note that QT is quantized time, a timestamp rounded to the nearest hour to reduce the risk of client identification, and the nonce serves as a single-use random value generated by the client.

2.4.2. Spend-Refund Flow

When spending Credits, the user first chooses the maximum spend for a computation and then supplies a Credit of that value. While performing inference, the system caps the available Credit at the value the user supplied, and returns a Credit with the unused balance to the user. Returned Credits undergo quantization to minimize information revealed by refunds. The user unblinds the returned Credit and exchanges it at the BlindBank for a blind Credit before depositing it.

The refund amount exposes the client to a timing attack. After unblinding the Credit, the



Auth
Bank
Client
TCloud
Credentials
Blind(QT || nonce, Amount)
Withdraw(Amount)
Authenticate
Deduct Amount from the user's balance
Sign(Blind(QT || nonce, Amount))
DespositTo(Account), Sign(QT || nonce, Amount)
Check QT and nonce
Verify signature
Account += Amount
WithdrawFrom(Account), Blind(QT' || nonce', Withdrawl)
Check Account has Withdrawl in it
Account-=Withdrawl
Sign(Blind(QT' || nonce', Withdrawl))
Sign(QT' || nonce', Withdrawl)
Encypted request
Check QT' and nonce'
Verify signature
Process request up to Withdrawl
Calculate remaining Change and sign it
Encrypted response
Sign(QT* || nonce*, Change)
Client needs to blind Change before depositing it
Exchange(Sign(QT* || nonce*, Change), Blind(QT'' || nonce'', Change))
Check QT* and nonce*
Verify signature
Sign(Blind(QT'' || nonce'', Change))
Now Change is blinded and unconnected to the request
DepositTo(Account), Sign(QT'' || nonce'', Change)
Check QT'' and nonce''
Verify signature
Account += Change
WithdrawFrom(Account, Blind(QT''' || nonce''', Refund))
Check Account has Refund in it
Account-=Refund
Sign(Blind(QT''' || nonce''', Refund))
Credentials
Sign(QT''' || nonce''', Refund)
Authenticate
Check QT''' and nonce'''
Verify signature
Add Refund to user's balance
Auth
Bank
Client
TCloud

FIGURE 5. The Complete Banking Flow

client should delay before depositing the refund into their account. OpenPCC includes mechanisms in the client tooling to support this behavior.

2.4.3. Privacy Considerations

OpenPCC's primary goal is to provide provably confidential computing. This necessarily extends to the currency system. OpenPCC assures user privacy by using an anonymized currency model, with OHTTP mediating all routes. The OpenPCC currency system prevents an attacker (including insider threats) from being able to deanonymize transactions by correlating:

1. A transaction to a user account
2. A user to a given inference engine
3. A user to a given prompt
4. A user to an allocated compute resource

To satisfy the above constraints, let's first look at the potential side channels in a naïve banking system.

Withdrawing Funds . When a user issues a withdrawal request with the AuthBank service, the system does not need to blind the AuthBank server to the user's identity, because the AuthBank must know the user's identity to issue the requested funds.

However, when the AuthBank service withdraws funds and returns Credits to the user, it must anonymize the issued Credit bundle so that the bundle serves as an anonymized bearer token for the allocated Credits. OpenPCC uses RSA Blind Signatures to achieve this.

Depositing Credits . When the user deposits Credits into the internal banking service, the BlindBank service cannot correlate the anonymized banking account to a true user identity. To ensure this, the client uses OHTTP to strip all identifiable metadata from the requests to the BlindBank. Additionally, the system assigns each anonymous banking account a random pseudonym identifier, and the user can generate ephemeral banking accounts to isolate different transactions.

Withdrawing Credits . When the user withdraws Credits from the internal banking service, the system ties the transaction to the ephemeral account pseudonym where the user deposited funds, and the transaction occurs over OHTTP. The BlindBank service issues a Credit bundle to the user using an RSA Blind Signature, which anonymizes the Credit bundle from the account pseudonym.

Requesting Inference . When the user issues an inference request, they forward a withdrawn Credit bundle and their inference request parameters (requested inference engine, prompt, etc.) to the Router service over OHTTP. The Router dispatches the request to the chosen inference engine and collects the result along with the amount of compute resources used. The system then calculates a refund Credit for the user by subtracting the compute cost from the submitted Credit bundle, applying stochastic rounding, and returning the result and refund to the user.

Weak Side Channels . This anonymization scheme mitigates side channel leaks, and tolerates compromise of ideally all but one component of the system. In practice, some components could be compromised as a pair to reveal certain information, if the client is not careful about API call timing and Credit allocation. None of these weaknesses should be possible with a well-behaved client wallet (described below) and sufficient OpenPCC network traffic.

- ***AuthBank* and *BlindBank***: If an attacker inspects both the *AuthBank* and *BlindBank* , two potential side channels arise:
- **Timing**: Correlating the time between withdrawing funds from AuthBank and depositing them into BlindBank can connect a real user identity to a banking pseudonym.
- **Funds**: Correlating the exact volume of funds withdrawn from AuthBank and deposited into BlindBank can similarly reveal a connection.
- ***BlindBank* and *Router***: If an attacker inspects both the *BlindBank* and *Router* two potential side channels arise:
- **Timing**: Correlating the time between withdrawing Credits from BlindBank and submitting an inference request to the Router can connect a compute request to a banking pseudonym.
- **Credits**: Correlating the exact volume of Credits withdrawn from BlindBank and submitted to the Router can similarly reveal a connection.

Note that the quantization scheme for Credits reduces the risk of cross-actor correlation, as Credit spends follow a smaller distribution of values. Moreover, these side channel leaks require simultaneous compromise of two or more OpenPCC services to correlate operations with user identities, which significantly reduces the risk of user de-anonymization.

Client Wallet . The client wallet holds Credits directly and maintains references to pseudonym accounts in the *BlindBank*. The wallet minimizes timing and correlation attacks by introducing delays and leveraging the *BlindBank*'s anonymity protections, which the wallet accesses via OHTTP. If desired, the client may use a locally managed wallet to store Credits and maintain anonymity.

Wallet Flow. The usage flow for the client-side wallet is as follows:

1. The client retrieves sufficient Credit from the *AuthBank* server to cover the expected number (N) of default-sized requests.
2. The client waits a random time period to obscure the source of this auth Credit.
3. The client deposits the auth Credit into a new *BlindBank* account.
4. The client withdraws N Credits of the default size from this *BlindBank* account, waits a random time, and then makes them available for use.

The Client exchanges refunded Credits at the *BlindBank*, then waits a random time before depositing them in a new blind account.

The Client manages pseudonymous accounts to:

1. Return excess Credit to the AuthBank if an account holds more than a set limit.
2. Consolidate accounts with insufficient Credit into a single new account.

To reduce correlations, the Client deposits one or more Credits into each pseudonym account, then withdraws one or more Credits until the account is empty, after which the account may not be reused.

2.5. Attested Secure Compute

Within OpenPCC, every secure ComputeNode encapsulates the inference engine inside a hardened virtual machine, within a Trusted Execution Environment (TEE) bound to a TPM. OpenPCC explicitly trusts secrets or artifacts received through a TPM, SecureBoot, or TEE, and these artifacts form the OpenPCC Trusted Compute Base (TCB) only after the system validates them through their respective chain-of-trust.

This document describes in-depth an implementation for the OpenPCC ComputeNode, called Confident-Compute. The image implements all hardening steps we believe are required for an OpenPCC-compliant ComputeNode and applies additional hardening steps to enforce the principle of defense-in-depth.

Note, however, that, by design, any implementer of the OpenPCC standard may create their own ComputeNode that follows the protocols defined herein. It may offer different workloads or different security assurances. When discussing the ComputeNode in the remainder of this document, we attempt to delineate the Confident-Compute implementation from the expectations of a generic ComputeNode that is compliant with the OpenPCC standard and meets the security requirements therein.

Confident-Compute uses a heavily modified Linux base image, security-hardened to make critical portions of the file system immutable and protect sensitive user data. After building

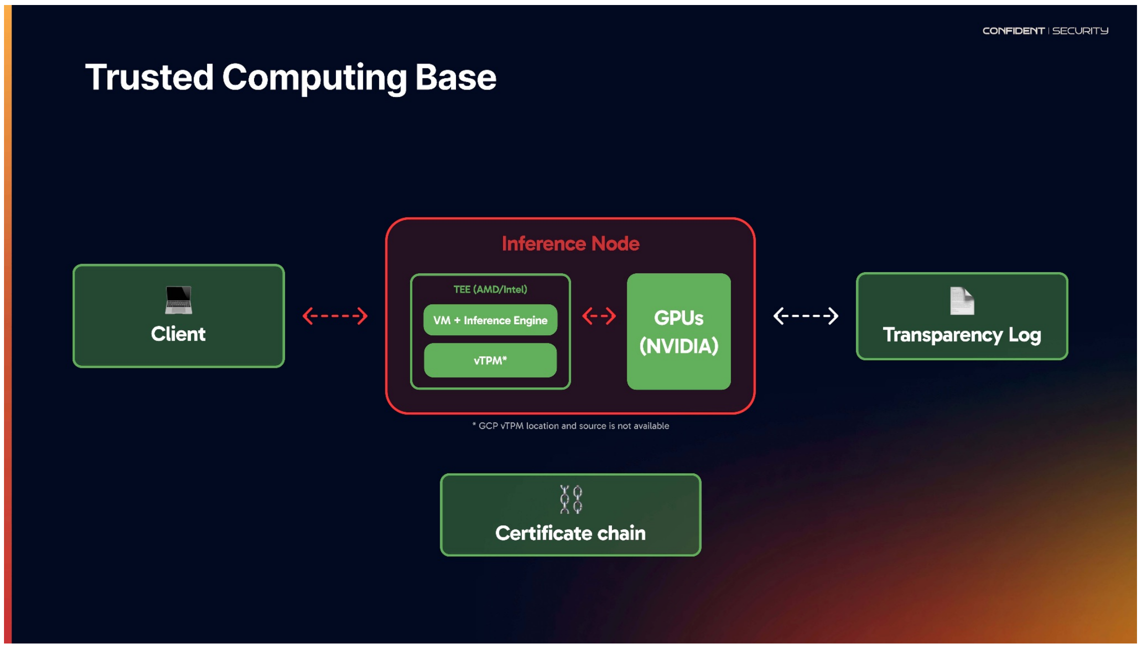


FIGURE 6. The Components of OpenPCC's Trusted Computing Base

the image, the system uploads the hardened Confident-Compute image's signature and security metadata to a public transparency log, which allows users to inspect and verify the integrity and contents of each image. All ComputeNode instances must include the following hardening elements:

- Trusted Execution Environment (TEE):
  - Both Intel (TDX) and AMD (SEV-SNP) instances of Confident-Compute use hardware security modules to encrypt memory, protect from hardware side channels, and provide a trust anchor for hardware attestation.
- Trusted Platform Module (TPM) 2.0:
  - All Confident-Compute instances have an associated ephemeral virtual TPM (TPM) that the instance uses to hold key material used by OpenPCC, sign attestation bundles to prove endpoint security, verify the correctness of the SecureBoot process, and provide hardware true random number generators (TRNG).
- Immutable file system:
  - Confident-Compute includes an immutable bootloader and root file system, which the system attests at runtime. This design enables the client to verify *ComputeNode* instance content via an attestation flow, chained to an image manifest entry in the OpenPCC transparency log.

2.5.1. Trusted Execution Environment

The Confident-Compute instances rely on a TEE and TPM 2.0 bundle. The TEE provides transparent encryption for data held in RAM, adds hardware security features, and includes an attestation mechanism that allows clients to verify the security of the TEE. On AMD machines the instances use Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP). On Intel machines the instances use Trust Domain Extensions (TDX).

2.5.2. Multi-Tenant Isolation

Confident-Compute recommends multi-tenancy for inference engines. This means that an individual inference engine may run on a single hardened Confident-Compute instance and host an arbitrary number of client inference queries. This setup poses several potential privacy and security issues, which Confident-Compute addresses through specific design considerations.

Confident-Compute and OpenPCC support a wide range of inference engines while explicitly isolating each engine and minimizing the actions each engine can perform. Confident-Compute heavily leverages Linux kernel security modules, such as SELinux, dm-verity, and dm-crypt, to isolate process-level operations. The OpenPCC client-side attestation tooling allows users to verify all these hardening steps.

2.5.3. General Security Considerations

Operating in a multi-tenant fashion widens the attack surface of the inference engine, as a compromise by one user may introduce compromise for other users of the same inference engine. To address this, we invested significant effort in hardening the Confident-Compute instance, limiting the "blast radius" of any critical exploit of an OpenPCC component or a 3rd-party inference engine. Confident-Compute provides sandboxing of the file system, process-level file system access control, an immutable root file system, and strict network controls for each compute worker instance. All of these constraints are attested, and users can verify these constraints using the publicly available compute image on the transparency log.

2.5.4. Hardening Steps

Confident-Compute implements a defense-in-depth approach, with layered security controls to limit the impact of any individually compromised component. The security of each instance is based on hardware security modules and cryptographic chains of trust, ensuring the security and confidentiality of both system and user data.

Confident-Compute heavily modifies the Ubuntu 22.04 base image to include relevant Linux kernel security modules, integrity protect the root and bootloader file systems, minify the userspace utilities to reduce the available attack surface, apply network firewall controls, and bind all modifications to the TPM's Platform Configuration Registers (PCRs). Extending all modifications to the TPM's PCRs allows clients to cryptographically verify each system modification in a manner chained back to the TPM's root of trust.

Modifications begin at the bootloader stage, where the dm-verity kernel module protects the integrity of UEFI and GRUB bootloaders. The dm-verity kernel module also appends the root hash of the bootloader partition's Merkle tree to PCR 8, and it is shown on the Linux kernel command line. The UEFI bootloader mediates the SecureBoot process, with the system extending the SecureBoot state to PCR 7.

Next, the build process applies modifications to the default kernel configuration. The build modifies the initramfs image, the early-stage kernel that bootstraps the pivot to the root file system, to include custom modules that download and integrity check the inference model(s) chosen for the ComputeNode instance, and extends the model digest(s) to PCR 12. The build binds an ephemeral, encrypted temporary file system to the TPM for the /var partition, holding any runtime cache for the inference engine. The system prevents access to this partition between boot cycles by terminating the LUKS key slot held by the TPM.

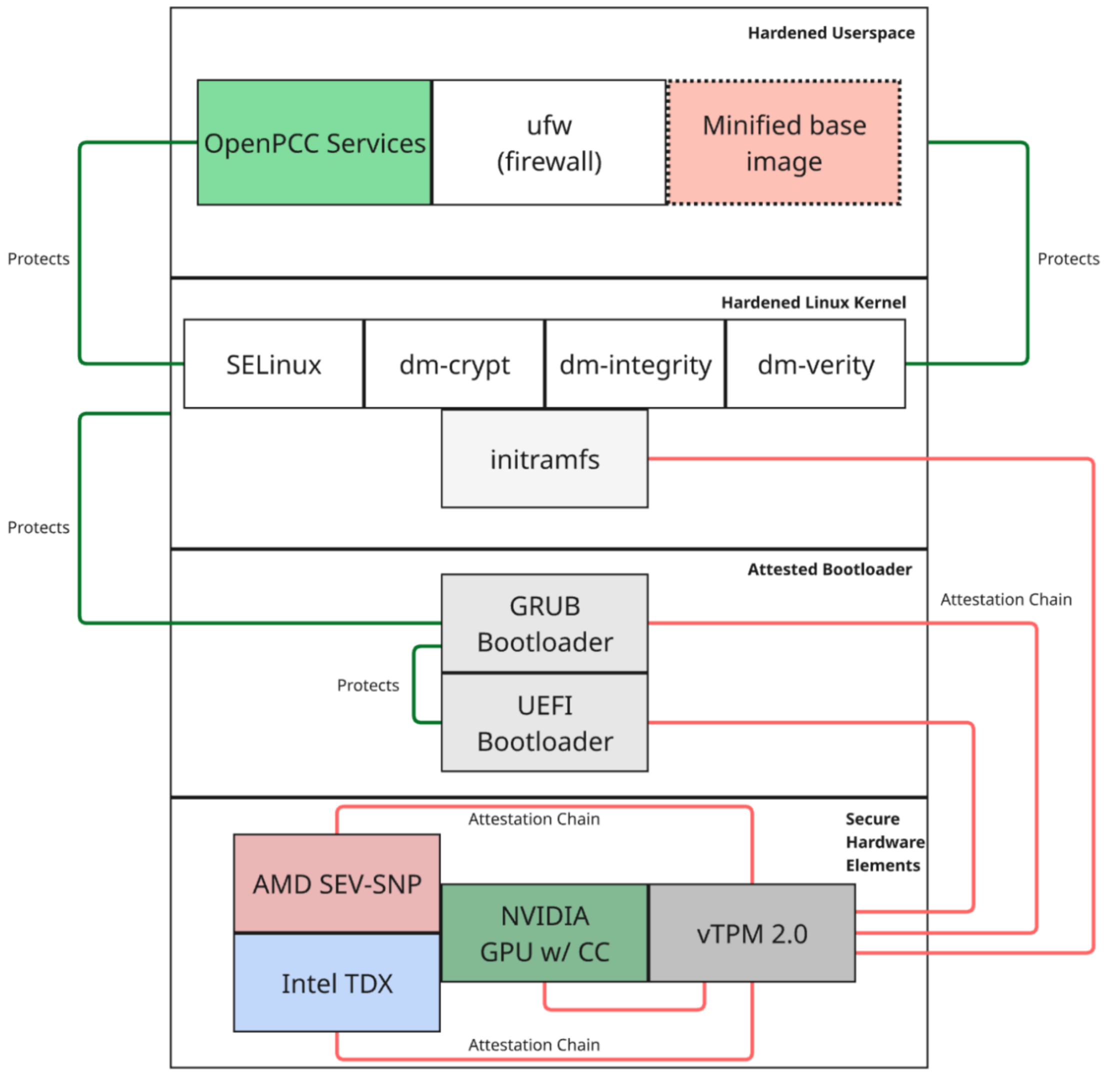


FIGURE 7. Trusted Execution Environment

After the image pivots to the root file system and the primary kernel image, SELinux isolates OpenPCC system services, and ufw establishes and enforces network firewall controls. Together, these tools prevent OpenPCC services and the inference engine from performing network operations not explicitly required for system operation.

Tying everything together: secure hardware elements for the TEE (TDX, SEV-SNP) and NVIDIA GPU complete the attestation chain to the TPM, binding all operating system modifications to the attestation bundle, verifiable by any client requesting inference on the given ComputeNode instance. The previous diagram illustrates this defense-in-depth approach to hardening the Confident-Compute image. Logical groupings of components appear in rectangles, in order of descending privilege. Red lines indicate links in the attestation chain, and green lines indicate that one component directly protects the integrity of another.

SecureBoot. The SecureBoot process of each cloud provider protects the boot chain of the Confident-Compute image, including the UEFI bootloader, GRUB bootloader, and Linux kernel. The system records SecureBoot measurements during the boot process and extends them to the allocated TPM's Platform Configuration Registers (PCRs), which are attested to following the boot process. Bootloader measurements are additionally integrity protected using each cloud provider's variant of Measured Boot, which assesses for deviations in the content of the PCR bank. The bootloader images themselves are integrity protected and attested to as well, using the dm-verity Linux kernel security module.

dm-verity . dm-verity provides integrity checks on a block device by computing a Merkle tree (one-way function tree) for the block device and using that data structure to perform runtime integrity checks on the data being accessed. If the system detects that a read-only sector does not conform to the hashes stored in the dm-verity Merkle tree, dm-verity marks the block as corrupted, and optionally enforces panic-on-corruption or restart-on-corruption.

Confident-Compute uses dm-verity to make the UEFI bootloader, GRUB bootloader, root file system, and the OpenPCC tooling partition (/opt) read-only and immutable. During the build process, the system modifies the base image partition layout to include dm-verity meta partitions to hold Merkle trees of each immutable partition and appends the kernel command line with the root digest of each Merkle tree. An exception is the GRUB bootloader partition, where the build process writes the Merkle root digest to the partition layout itself (since the GRUB bootloader partition holds the kernel command line).

On boot, the system extends these digests to the UEFI event log, which the TPM consumes and extends to the TPM PCRs. The UEFI event log and PCR hash chains form an attestation artifact in the final attestation bundle, allowing clients to verify that the content of the immutable partitions matches the transparency log.

dm-crypt + dm-integrity. dm-crypt is a disk encryption kernel module that operates on a block device. Confident-Compute binds the runtime sandbox read-write partitions /home and /var to dm-crypt using LUKS.

dm-integrity is an extension to dm-crypt that adds an HMAC to each dm-crypt block, so the data can be checked for corruption rather than returning garbage to the caller.

The dm-crypt and dm-integrity Linux kernel modules allow Confident-Compute to provide read-write portions of the file system to hold inference cache and temporary files securely. The boot process binds these special partitions to the allocated TPM and re-encrypts them, so they become wholly inaccessible to anyone, including the Confident Security team, when the image is shut down and the TPM is freed and terminated. Note that because they are re-encrypted, their initial content is ignored and not an attack vector.

SELinux . SELinux is used for granular access control for the image, only allowing permitted actions for the binaries in the /opt partition (i.e., the OpenPCC tooling). For context, SELinux is a hardening kernel subsystem that enforces Mandatory Access Control (MAC) in the compute environment. In typical Linux operating systems, files and processes are protected using Discretionary Access Control (DAC). Under DAC, file access is governed by ownership and permission bits (owner:group:all with read, write, and execute rights). These permissions are visible in the output of a typical ls -l call. This security model can be bypassed by any root process with the commonly given cap_override_dac capability. Therefore, a single compromised root process could freely pivot to inspection of the entire system. This includes examining the process space maps of the inference engine, attaching a debugger to the engine, listening to communication sockets, or creating core dumps, jeopardizing the secure environment.

By contrast, under MAC, SELinux enforces policy-based restrictions at the kernel level. System calls that are not explicitly allowed by the loaded policy are blocked:

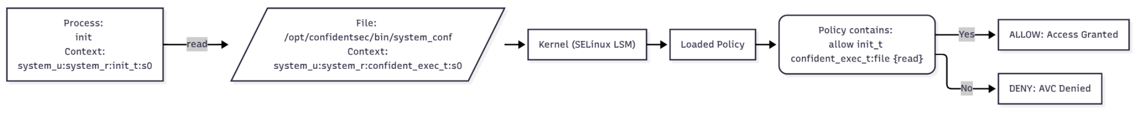


FIGURE 8. SELinux Syscall Filtering

Depending on the policy mode, denials are either logged without enforcement (permissive mode) or result in an immediate “permission denied” response (enforcing mode).

SELinux was selected from the various other forms of hardening/isolation available (e.g., AppArmor, KVM, Docker containerization) due to the monolithic level system-level hardening with no exceptions or high-level processes being immune to the created security environment. AppArmor does not restrict systemwide only on a binary-by-binary basis. In a KVM/Docker environment, it becomes increasingly challenging to ensure the host system cannot inspect compute-engine internals.

Confident-Compute uses SELinux to ensure process isolation and restrict the capabilities of the loaded inference engine, ensuring that only select executables are accessible to the inference engine user. The permissions allowed in Confident-Compute are determined based on an auditing procedure, wherein we run the image in auditing mode, collect the resulting SELinux into an allow list, transform that into appropriate policies, and include them in the compute image, wherein they become immutable. This policy prevents administrative/root privileged users from accessing the compute-engine and protects compute assets from insider, high-privileged threats.

We have added additional SELinux types to the SELinux Project’s reference policy in order to further harden Confident-Compute tooling to defend against the threats discussed in the later section named “Security Considerations”. New types include: confident_config_t, confident_exec_t and confident_internal_exec_t. During boot, the init process domain transitions into confident_exec_t, thereby configuring the program options, models, and cloud meta-data information. This process then transitions into the confident_internal_exec_t for workloads, TPM functions, and TLS termination. This extra buffer domain-specific policy allows only what is expressly needed to run the inference program while ensuring the init process must go through confident_exec_t. In the event an init_t privileged process is compromised, the attack surface into the confident_internal_exec_t domain is limited to the transition program.

The following diagram shows the SELinux domain transition path from init systemctl service initialization through to tpm compute_boot binary execution:

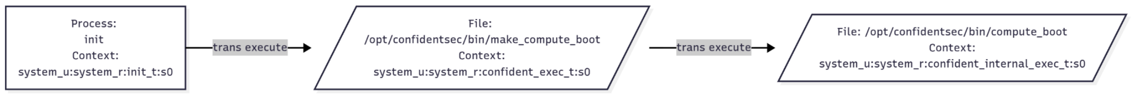

Process:
init
Context:
system_u:system_r:init_t:s0
trans execute
File:
/opt/confidentsec/bin/make_compute_boot
Context:
system_u:system_r:confident_exec_t:s0
trans execute
File: /opt/confidentsec/bin/compute_boot
Context:
system_u:system_r:confident_internal_exec_t:s0

FIGURE 9. Domain Transition Path

This domain transition workflow is reused for the router_com process.

Specific compute hardening granted from SELinux policy:

- Port Binding Restrictions: prevents non-privileged processes from binding to the inference engine socket.
- Directory Access Control: prevents privileged shells from accessing the /opt/ OpenPCC tooling directory.
- Ptrace and Signal Protection: Blocks inter-process tracing and interactions.
- Cloud provisioning sandboxing: Scripts are blocked from modifying accounts through /etc/shadow or authorized_keys

OS Modifications . The Confident-Compute ComputeNode operating system image is a heavily modified version of Ubuntu 22.04 LTS with an immutable root file system, restrictive firewall rules, strong SELinux protections, and ephemeral cryptography and stateless computation techniques that assure the privacy of user data. Note that we selected Ubuntu 22.04 as a base image for first-class NVIDIA driver support. Confident-Compute packages and enables several security controls while disabling non-essential packages from the mainline Linux kernel and Ubuntu LTS. In particular,

- Minimization of the system image to reduce the size of the SBOM and prune unnecessary binaries, services, and configuration.
- Hardening of the kernel and file system using dm-verity, dm-crypt, and dm-integrity .
- Implements a read-only, immutable root partition (/); assures immutability of system binaries, services, and configuration.
- Implements read-only, immutable boot partitions (/boot and /boot/efi); assures immutability of the GRUB & UEFI bootloader.
- Implements a read-only, immutable OpenPCC tooling partition (/opt); assures immutability of the OpenPCC binaries.
- Implements read-write, ephemeral /home and /var partitions, encrypted with a key slot bound to the TPM; ensures that temporary files and model cache are inaccessible once the TPM is garbage collected.
- Implements a read-only, ephemeral model ramdisk; assures the model weights are encrypted (using the TEE protections) while maintaining performance.
- Modifications to the Linux kernel command line:
- dm-verity root hashes, versioning information, and hardening parameters.
- The kernel command line arguments are extended to TPM PCR 8; allows users to verify content against the Sigstore transparency log.
- Implements default-deny firewall rules using ufw; only allows network operations that are essential for OpenPCC on restricted CIDR ranges.

- Removing remote serial console access for the cloud provider
- Disabling ssh

- Disabling Linux kernel rescue shell
- Applying the lockdown, landlock, and Yama kernel security modules

Note that all file system content can be externally audited by examining the OS image present on the Sigstore transparency log.

File System Modifications . The Confident-Compute image has several hardening modifications applied to the base file system. First, the root file system, UEFI bootloader partition, and GRUB bootloader partition are immutable and integrity protected using dm-verity. The dm-verity root hashes are extended to TPM PCRs 5 and 8, which are chained back to the OpenPCC transparency log for client verification. Moreover, the inference models are downloaded to a RAMdisk during the early boot stage (initramfs) and the model digest is extended to PCR 12, which is chained back to the transparency log for verification. Note that the ramdisk is encrypted within the TEE.

The dm-verity protected partitions recorded in the transparency log allow the ComputeNode to attest to the immutable content on these portions of the file system. The publicly available compute image, recorded to the transparency log along with the respective dm-verity root hashes, allows anyone to independently audit the correctness of the file system content and the integrity of the compute image itself. When validating a ComputeNode's attestation result, the client checks for (1) the integrity of the attestation bundle chained to the hardware roots of trust in the TEE and TPM, and (2) the integrity of the content of the attestation bundle by verifying the equivalence of the attestation bundle evidence with the transparency log records. This provides assurance that (a) the attestation result is generated in a valid compute instance, chained to a secure hardware enclave with trusted certificates, and (b) the content of the secure enclave is equivalent to the latest release of the OpenPCC system, chained to the OpenPCC transparency log public keys.

To emphasize the claims here: if an attacker wished to compromise the immutable content of the OpenPCC ComputeNode (i.e., the base system image and OpenPCC system services), they would have to either (i) compromise the OpenPCC transparency log public-key infrastructure to append invalid file system digests *and* tamper with the base image deployed to a ComputeNode, or (ii) compromise the Linux kernel security modules (dm-verity, SELinux) to corrupt the immutable file system without deviating from the content of the transparency log. Employing rigorous key management practices and expiry mechanisms mitigates the former while monitoring for CVEs and regularly re-deploying Confident-Compute and OpenPCC related infrastructure, with all patching performed offline before deployment, mitigates the latter.

Confident-Compute also uses two ephemeral partitions (/home and /var) encrypted using a key bound to the TPM using dm-crypt. These partitions hold cache and other temporary files generated by inference engines and system services. When the compute instance first boots, these partitions are re-encrypted and re-keyed using the ephemeral TPM. This ensures that when the machine reboots, these partitions are inaccessible, as the TPM is terminated upon instance shutdown.

The figure below demonstrates the logical file system layout, including anchors back to the TPM's PCR bank and the transparency log.

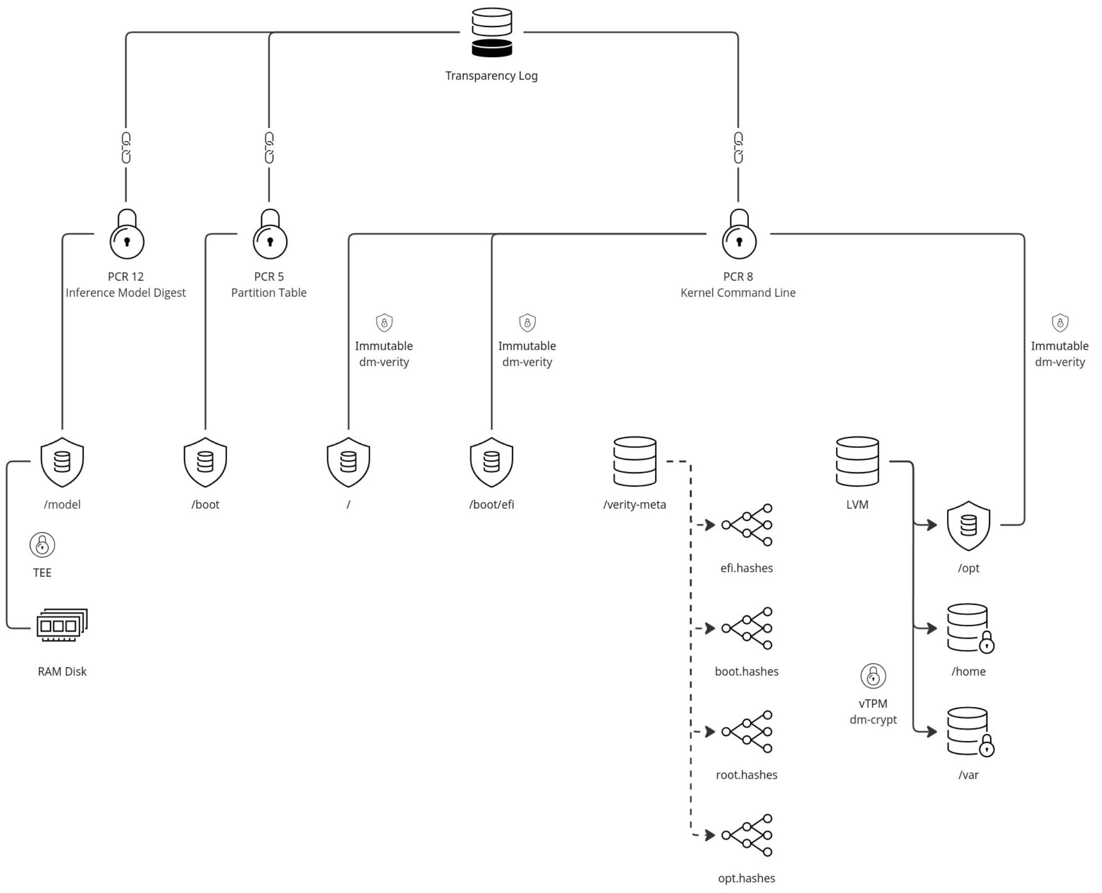


FIGURE 10. Filesystem Layout Ties To Transparency Log

OS Boot Sequence . When the compute instance first boots, it goes through a series of early-stage boot routines (in the initramfs) to:

- Mount the immutable file system using:
– (1) the root hash provided in the kernel command line and,
– (2) the hash provided in the partition table.
- Re-encrypt the /home and /var partitions, binding them to the ephemeral TPM.

- Download the inference model weights and extend them to the TPM's PCR 12 for attestation.

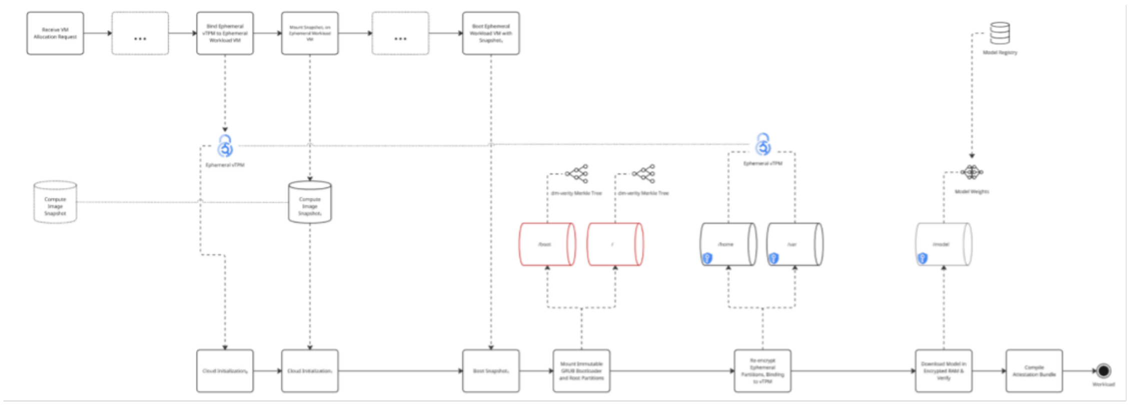

Receive VM Allocation Request
...
Bind Ephemeral vTPM to Ephemeral Workload VM
Mount Snapshot on Ephemeral Workload VM
...
Boot Ephemeral Workload VM with Snapshot
Model Registry
Ephemeral vTPM
Compute Image Snapshot
Compute Image Snapshot
dm-verity Merkle Tree
dm-verity Merkle Tree
Ephemeral vTPM
Model Weights
/boot
/
/home
/var
/model
Cloud Initialization
Cloud Initialization
Boot Snapshot
Mount Immutable GRUB Bootloader and Root Partitions
Re-encrypt Ephemeral Partitions, Binding to vTPM
Download Model in Encrypted RAM & Verify
Compile Attestation Bundle
Workload

FIGURE 11

Attesting and Verifying the Image. The hardened Confident-Compute image plays an important role in the OpenPCC attestation process. When a client verifies the attestation bundle provided by a *ComputeNode*, the image manifest is included in the verification process. This manifest includes the image digest, kernel command line parameters, and the GUID Partition Table for the image. The content included in these fields validates:

1. The immutable partitions of the file system, allowing the client to validate the file system content as a Merkle root hash.
2. The hardening configuration parameters used in the image build process
3. The git commit hash used for each stage of the image build process

Taking these fields in aggregate, along with the TPM and TEE evidence bundles, the client can have confidence that the ComputeNode is running an authentic Confident Compute image that was not tampered with during any stage of the deployment process.

Debugging the Confident-Compute Image. The hardening steps, by definition and by intention, result in severely restricted remote monitoring and debugging in production environments. The build parameters available for debugging non-production images are also included in the Confident-Compute image manifest that is appended to the transparency log, thereby enabling users to verify these images are not used in production.

Software Tooling. The Confident-Compute image is provided as source-available, and individual compute image releases are published to a managed OpenPCC transparency log for validation during attestation and verification routines.

A custom Packer workflow builds the image for each respective hardware provider (GCP, Azure, bare-metal), then publishes all to an image repository and transparency log. Anyone may download the image and inspect the source code for the build process to validate the correctness of the hardening modifications thereof.

2.5.5. Attestation & Verification

The full OpenPCC attestation flow allows Clients to verify (1) that their compute environments are securely isolated, (2) that those environments are bound to secure hardware and execution environment (TEE + TPM) with a verifiable chain-of-trust, and (3) that the operating system image, from the firmware up to userspace applications, is immutable and equivalent to the images for which publicly auditable signatures and security metadata are provided in the OpenPCC transparency log. An OpenPCC service MUST provide attestation results to each client before routing their Compute Request, allowing the client to verify the claims chained to their respective roots of trust.

The attestation system uses a “passport” model, in which the Attester pre-validates an attestation bundle that is presented to the user for verification. We chose this model to allow one-time client-side verification of the attestation bundles for a given window-of-use. The requirements of a valid hardened ComputeNode instance ensures that the ComputeNode instance is unable to deviate from the content of the attestation result after its production. To prevent the reuse of old or invalidated attestation bundles, an OpenPCC Client and ComputeNode leverage TPM mechanisms, including TPM Policy sessions, to expire passports safely and securely.

Roots of Trust. Claims made in an OpenPCC attestation result chain back to a given root of trust. These chains allow secure hardware elements and transparency log content to cryptographically bind to the corresponding piece of evidence being attested to. When a client verifies a piece of evidence, they verify the correctness of the claim and validate the cryptographic binding to the given root of trust. All information necessary to verify

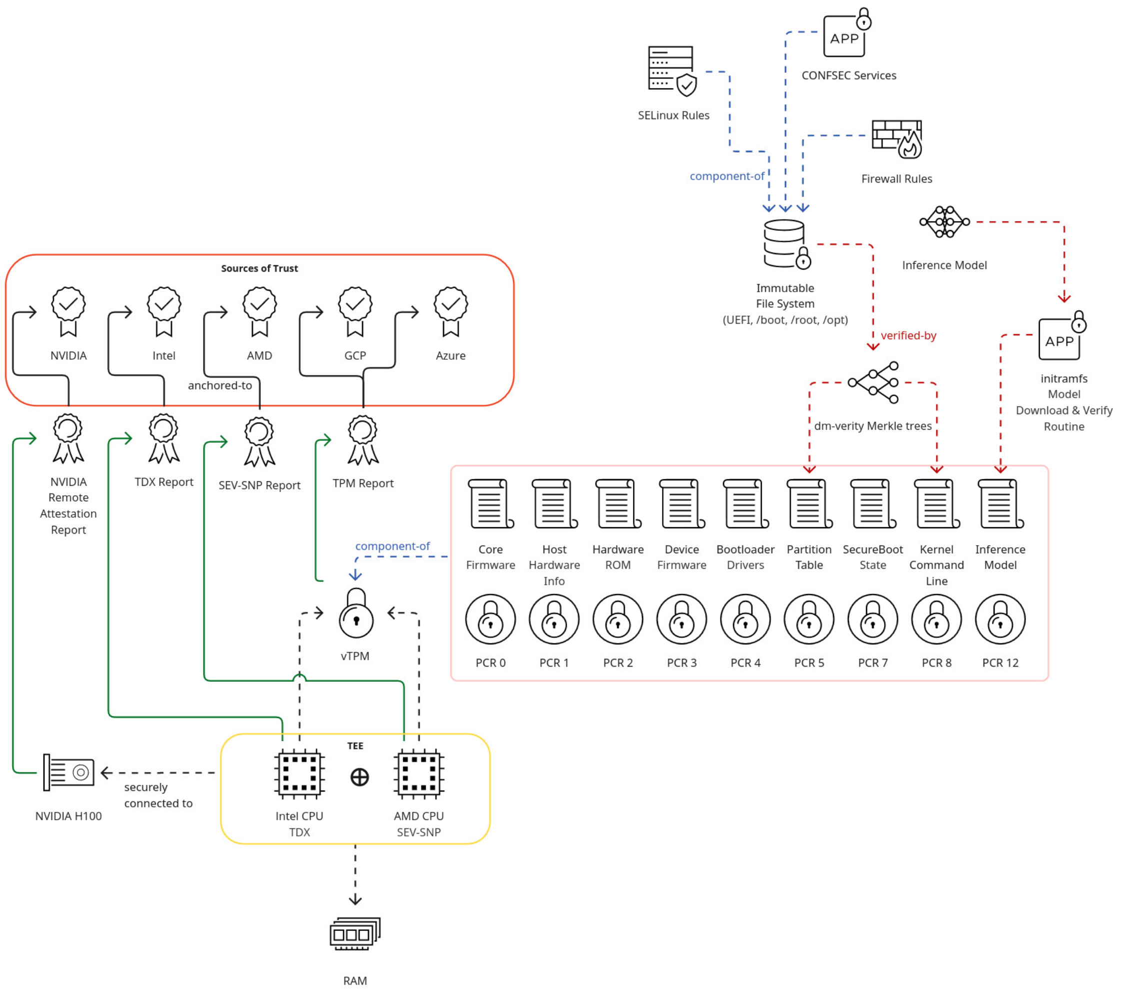


FIGURE 12. Secure Hardware Elements for Compute Attestation

Trust domains are shown as rounded rectangles, including the TPM, TEE, and third party sources of trust. Dashed red arrows denote that an artifact is attested to by the dependent component. Dashed black arrows indicate secure hardware element connections. Solid green arrows denote composition in an attestation evidence type.

an attestation claim up to the root of trust must be included in the transparency log, client-side verification routines, or the attestation result itself.

Confidential Compute The compute environment, regardless of the infrastructure provider, must allocate a Trusted Platform Module 2.0 (TPM), a Trusted Execution Environment (TEE), and NVIDIA GPU(s) in Confidential Computing mode. OpenPCC supports any TEE capable of issuing an attestation result, with current implementations supporting Intel Trust Domain Extensions (TDX) and AMD Secure Encrypted Virtualization-Secure Nested Paging (SEV SNP). As the computing engine is instantiated and the virtual machine boots, the ComputeNode instance collects signed attestation reports, attesting to the co-locality of the TPM, TEE, and GPU, with the TPM signing all attestation reports.

The operating system image includes an immutable file system for the UEFI bootloader, GRUB boot partition, and root file system. The dm-verity Linux kernel module protects these immutable file system partitions, allowing OpenPCC to extend the Linux kernel command line with a digest of the file system and to use a Merkle tree of each immutable partition to detect corruption at runtime. The digests present in the kernel command line are extended to the TPM's Platform Configuration Registers (PCRs), an append-only record that is included in the final attestation bundle for each private compute environment.

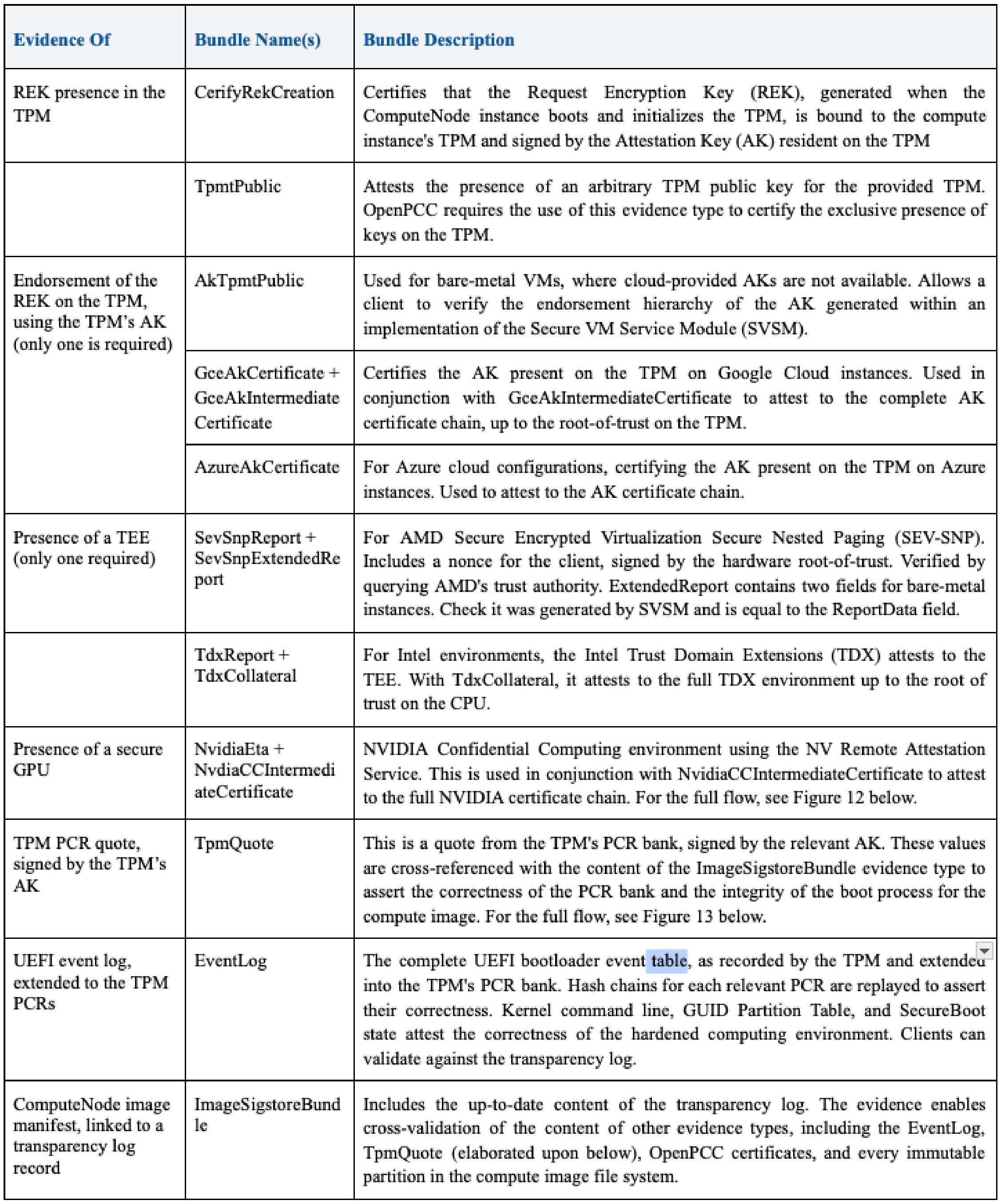
Transparency Log The OpenPCC transparency log, which includes all artifacts necessary to verify the complete bundle of evidence, enables client-side verification of the attestation result. OpenPCC clients must have attestation verification routines to validate the presented evidence bundle, including checking against the transparency log.

Evidence Types. The OpenPCC specification defines several standard evidence types, signed by one or more of the following keys:

- **Attestation Key (AK):** The asymmetric signing key used by the TPM to attest to keys resident in the TPM.
- **Endorsement Key (EK):** The asymmetric signing key used to endorse the AK.
- **Data Encryption Key (DEK)**: The symmetric key used to encrypt inference prompts and outputs.
- **Request Encryption Key (REK)**: The key resident on each ComputeNode’s TPM to encrypt the DEK used for each inference request.

To produce a valid result, any attestation must include the following evidence bundles.

TpmQuote This is a quote from the TPM's PCR bank, signed by the relevant AK. These values are cross-referenced with the content of the ImageSigstoreBundle evidence type



Evidence Of	Bundle Name(s)	Bundle Description
REK presence in the TPM	CerifyRekCreation	Certifies that the Request Encryption Key (REK), generated when the ComputeNode instance boots and initializes the TPM, is bound to the compute instance's TPM and signed by the Attestation Key (AK) resident on the TPM
	TpmtPublic	Attests the presence of an arbitrary TPM public key for the provided TPM. OpenPCC requires the use of this evidence type to certify the exclusive presence of keys on the TPM.
Endorsement of the REK on the TPM, using the TPM's AK (only one is required)	AkTpmtPublic	Used for bare-metal VMs, where cloud-provided AKs are not available. Allows a client to verify the endorsement hierarchy of the AK generated within an implementation of the Secure VM Service Module (SVSM).
	GceAkCertificate + GceAkIntermediate Certificate	Certifies the AK present on the TPM on Google Cloud instances. Used in conjunction with GceAkIntermediateCertificate to attest to the complete AK certificate chain, up to the root-of-trust on the TPM.
	AzureAkCertificate	For Azure cloud configurations, certifying the AK present on the TPM on Azure instances. Used to attest to the AK certificate chain.
Presence of a TEE (only one required)	SevSnpReport + SevSnpExtendedRe port	For AMD Secure Encrypted Virtualization Secure Nested Paging (SEV-SNP). Includes a nonce for the client, signed by the hardware root-of-trust. Verified by querying AMD's trust authority. ExtendedReport contains two fields for bare-metal instances. Check it was generated by SVSM and is equal to the ReportData field.
	TdxReport + TdxCollateral	For Intel environments, the Intel Trust Domain Extensions (TDX) attests to the TEE. With TdxCollateral, it attests to the full TDX environment up to the root of trust on the CPU.
Presence of a secure GPU	NvidiaEta + NvdiaCCIntermedi ateCertificate	NVIDIA Confidential Computing environment using the NV Remote Attestation Service. This is used in conjunction with NvidiaCCIntermediateCertificate to attest to the full NVIDIA certificate chain. For the full flow, see Figure 12 below.
TPM PCR quote, signed by the TPM's AK	TpmQuote	This is a quote from the TPM's PCR bank, signed by the relevant AK. These values are cross-referenced with the content of the ImageSigstoreBundle evidence type to assert the correctness of the PCR bank and the integrity of the boot process for the compute image. For the full flow, see Figure 13 below.
UEFI event log, extended to the TPM PCRs	EventLog	The complete UEFI bootloader event table, as recorded by the TPM and extended into the TPM's PCR bank. Hash chains for each relevant PCR are replayed to assert their correctness. Kernel command line, GUID Partition Table, and SecureBoot state attest the correctness of the hardened computing environment. Clients can validate against the transparency log.
ComputeNode image manifest, linked to a transparency log record	ImageSigstoreBund le	Includes the up-to-date content of the transparency log. The evidence enables cross-validation of the content of other evidence types, including the EventLog, TpmQuote (elaborated upon below), OpenPCC certificates, and every immutable partition in the compute image file system.

Table 1. Minimal Evidence Required for Valid Attestation Results

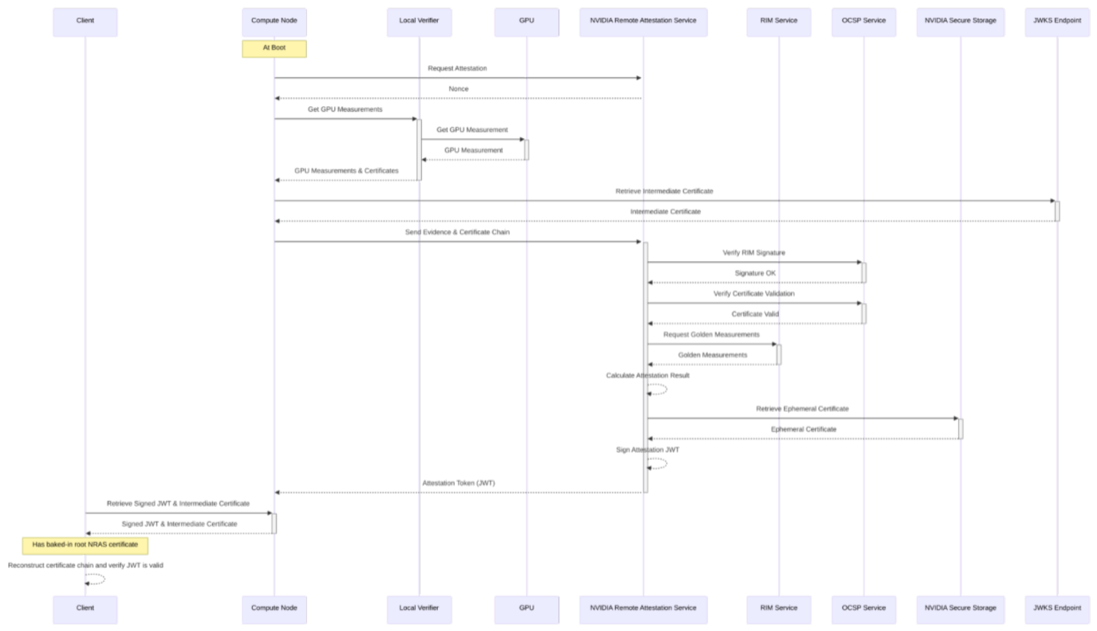

Client
Compute Node
Local Verifier
GPU
NVIDIA Remote Attestation Service
RIM Service
OCSP Service
NVIDIA Secure Storage
JWKS Endpoint
At Boot
Request Attestation
Nonce
Get GPU Measurements
Get GPU Measurement
GPU Measurement
GPU Measurements & Certificates
Retrieve Intermediate Certificate
Intermediate Certificate
Send Evidence & Certificate Chain
Verify RIM Signature
Signature OK
Verify Certificate Validation
Certificate Valid
Request Golden Measurements
Golden Measurements
Calculate Attestation Result
Retrieve Ephemeral Certificate
Ephemeral Certificate
Sign Attestation JWT
Attestation Token (JWT)
Retrieve Signed JWT & Intermediate Certificate
Signed JWT & Intermediate Certificate
Has baked-in root NRAS certificate
Reconstruct certificate chain and verify JWT is valid

FIGURE 13

to assert the correctness of the PCR bank and the integrity of the boot process for the compute image.

OpenPCC operations should validate the following PCR values:

- PCR 0: "Core system firmware executable code"
- PCR 1: "Core system firmware data/host platform configuration; typically contains serial and model numbers"
- PCR 2: "Extended or pluggable executable code; includes option ROMs on pluggable hardware" PCR 3: "Extended or pluggable firmware data; includes information about pluggable hardware" PCR 4: "Boot loader and additional drivers; binaries and extensions loaded by the boot loader" PCR 5: "GPT/Partition table"
– The GUID Partition Table includes versioning information and the dm-verity root hash for the immutable GRUB bootloader partitions.
- PCR 7: "SecureBoot state"
- PCR 8: "Commands and kernel command line"
– The kernel command line includes the dm-verity root hash for all immutable file system partitions, along with image hardening settings and versioning information.
- PCR 12: Inference model digest; extended during the early-stage boot process by a publicly available OpenPCC initramfs module.

Figure 13 demonstrates the boot chain as it pertains to PCR extension. Modification of any component (e.g., core firmware version, hardware configuration) changes the PCR bank's hash chain. The TpmQuote client verification routine must accommodate such changes. Users wishing to implement additional checks on the PCR hash chain, as reflected in the TpmQuote, can extend the OpenPCC client reference implementation to impose additional constraints on the UEFI event log. Moreover, alternative implementations of OpenPCC may extend additional content to the PCR bank, so long as this content is wholly reflected in the variant's implementation of the TpmQuote verification routine.

Verification Routines. When the user wishes to make an inference request, their client first fetches an attestation bundle for the desired compute resource. The Router forwards the attestation bundle to the client. Note that attestation results are generated when a ComputeNode instance initializes. The user then validates the attestation bundle using the included (and auditable) verification routines in the OpenPCC client.

Because Remote Attestation occurs with many different (and often connected) elements of the computing environment, the process is modular, where different hardware and software configurations follow slightly different verification pathways, determined by the evidence that is provided, in order to produce the verification results that will be

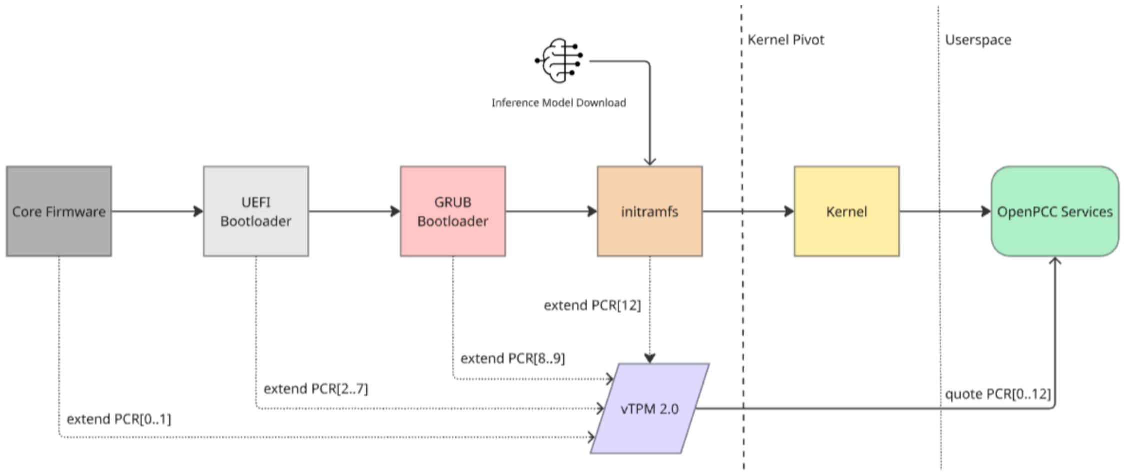


FIGURE 14. TpmQuote Components for Validation

evaluated by the client.

Modular Verification Design To make OpenPCC a fully open and extensible platform, it supports extensible and plug-and-play integration of client verification routines. That is, anyone can fork or patch the reference implementation of the OpenPCC client to include additional or alternative verification routines.

Any user can harden, or relax, the verification routines for clients that wish to check additional or alternative properties of the OpenPCC ComputeNode attestation results. For instance, an implementor can extend the verification routines to include more rigorous checks on the UEFI event table content, constraining the client to approve ComputeNode instances using particular hardware configurations only.

This design choice also allows alternative managed services to federate and support implementation variants of the Confident Compute's ComputeNode reference implementation. The only caveat is that all parties in the federation must accurately tag their evidence types, and provide verification routines to all member clients to prevent namespace collision and diverging verification routines for the client.

Cryptographic Binding to Roots of Trust All evidence types ultimately bind to one of:

1. The OpenPCC transparency log
2. Third party cloud providers or internet public-key infrastructure
3. Hardware roots-of-trust

Figure 14 maps the composition and attestation relationships between all evidence types and the intermediate artifacts they reflect.

The bindings to each root of trust provide a pathway for verifying each individual claim back to an externally managed source of trust, ultimately binding to a hardware component or the OpenPCC transparency log. This way, when verifying a particular piece of evidence, the client does not have to assert trust in any piece of proprietary code or black-box component – all claims chain back to third party roots of trust or open-source verification routines provided in the OpenPCC reference implementation.

In order to compromise any individual piece of evidence, an attacker would need to compromise the PKI of the third party root of trust or secure hardware element that the evidence uses. In aggregate, this type of compromise requires an intractable attack on every relevant piece of evidence in the attestation bundle, as a single failure invalidates the attestation results, and the client will reject them.

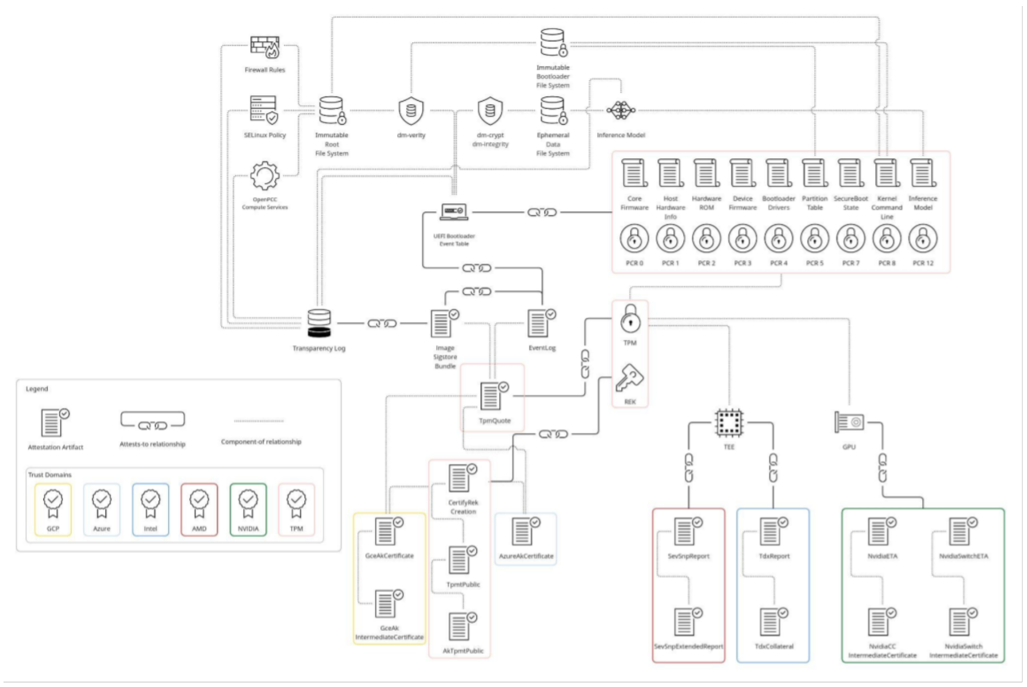

Firewall Rules
SELinux Policy
OpenPCC Compute Services
Immutable Root File System
dm-verity
dm-crypt dm-integrity
Immutable Bootloader File System
Ephemeral Data File System
Inference Model
Core Firmware
Host Hardware Info
Hardware ROM
Device Firmware
Bootloader Drivers
Partition Table
SecureBoot State
Kernel Command Line
Inference Model
PCR 0
PCR 1
PCR 2
PCR 3
PCR 4
PCR 5
PCR 7
PCR 8
PCR 12
UEFI Bootloader Event Table
Transparency Log
Image Sigstore Bundle
EventLog
TPM
REK
TpmQuote
Legend
Attestation Artifact
Attests-to relationship
Component-of relationship
Trust Domains
GCP
Azure
Intel
AMD
NVIDIA
TPM
TEE
GPU
CertifyRek Creation
GceAkCertificate
GceAk IntermediateCertificate
TpmtPublic
AkTpmtPublic
AzureAkCertificate
SevSnpReport
SevSnpExtendedReport
TdxReport
TdxCollateral
NvidiaETA
NvidiaSwitchETA
NvidiaCC IntermediateCertificate
NvidiaSwitch IntermediateCertificate

FIGURE 15

2.6. Security Considerations

The OpenPCC standard provides both privacy and security to end users, even in the case where an individual component of the system has been compromised. This section examines several mechanisms that could allow a component to be compromised and describes the specific mitigation strategies for that scenario that the standard requires.

2.6.1. Supply Chain Providers

The trust guarantees of the OpenPCC system are anchored in the secure hardware elements utilized in (often 3rd-party) hosting environments. OpenPCC implementers should continuously monitor for advisories on leaked key material for the secure elements, and implementers may use remote attestation mechanisms to process certificate revocations and updates.

2.6.2. Platform Providers

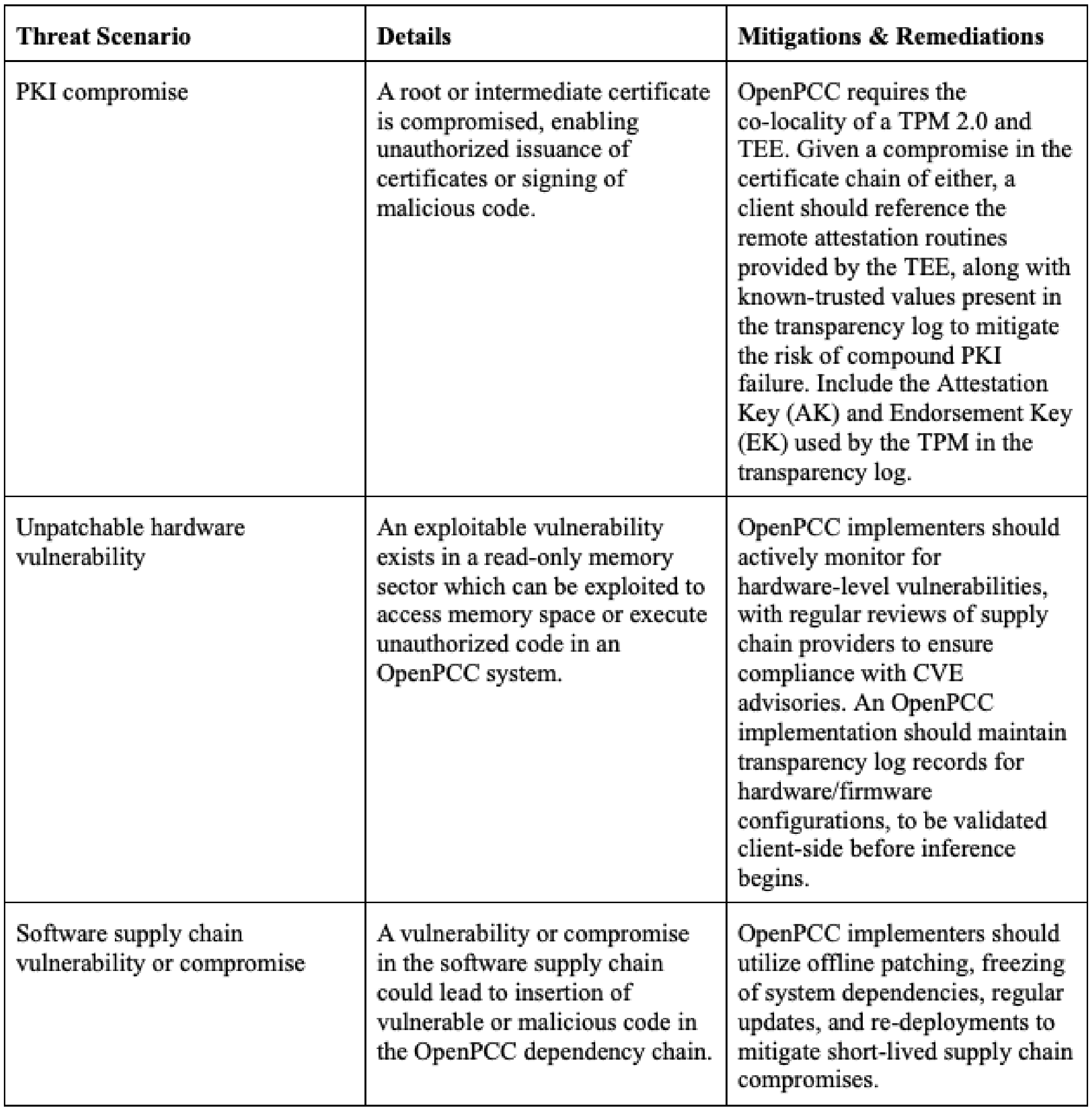
OpenPCC implementations may utilize 3rd-party cloud, infrastructure, and/or hosting providers. We consider these actors implicitly untrusted. The high privilege level of hypervisor access and/or physical access limits userspace actions for mitigation. Instead, OpenPCC requires that a modification of the system at the physical level is reflected in the received client attestation report. Secrets or artifacts received through a cloud provider's TPM, SecureBoot, or TEE are considered explicitly trusted, and form the OpenPCC Trusted Compute Base (TCB), after validation through their respective chain-of-trust.

2.6.3. Inference Engines

In this design, an inference engine used within an OpenPCC system is not trusted. Do not assume that an inference engine is free from vulnerabilities, nor is it trusted to interact directly with entities in the OpenPCC system. This design consideration forces sandboxing and isolation of potential zero-day compromises of the chosen inference engine (SGLang, vLLM, etc.), making the platform more capable of hosting generic workloads as a side effect.

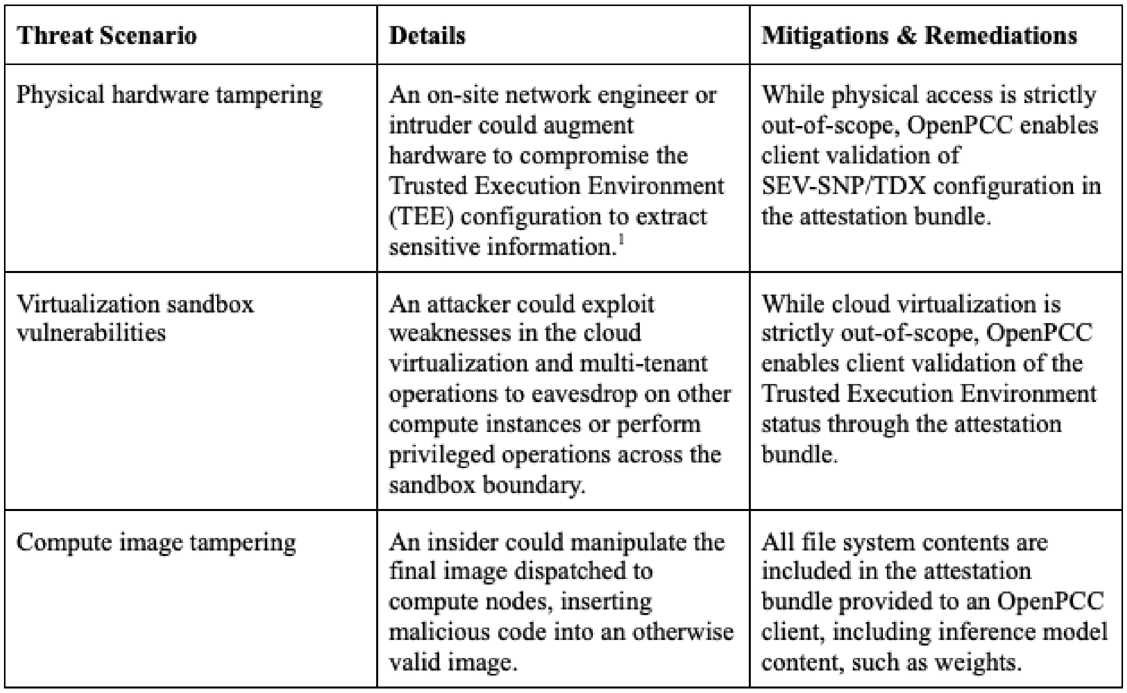
2.6.4. System Administrators

OpenPCC considers and mitigates the following threat scenarios related to insider threats from compromised system administrators and/or developers.



Threat Scenario	Details	Mitigations & Remediations
PKI compromise	A root or intermediate certificate is compromised, enabling unauthorized issuance of certificates or signing of malicious code.	OpenPCC requires the co-locality of a TPM 2.0 and TEE. Given a compromise in the certificate chain of either, a client should reference the remote attestation routines provided by the TEE, along with known-trusted values present in the transparency log to mitigate the risk of compound PKI failure. Include the Attestation Key (AK) and Endorsement Key (EK) used by the TPM in the transparency log.
Unpatchable hardware vulnerability	An exploitable vulnerability exists in a read-only memory sector which can be exploited to access memory space or execute unauthorized code in an OpenPCC system.	OpenPCC implementers should actively monitor for hardware-level vulnerabilities, with regular reviews of supply chain providers to ensure compliance with CVE advisories. An OpenPCC implementation should maintain transparency log records for hardware/firmware configurations, to be validated client-side before inference begins.
Software supply chain vulnerability or compromise	A vulnerability or compromise in the software supply chain could lead to insertion of vulnerable or malicious code in the OpenPCC dependency chain.	OpenPCC implementers should utilize offline patching, freezing of system dependencies, regular updates, and re-deployments to mitigate short-lived supply chain compromises.

Table 2. Supply Chain Threats and Remediations



Threat Scenario	Details	Mitigations & Remediations
Physical hardware tampering	An on-site network engineer or intruder could augment hardware to compromise the Trusted Execution Environment (TEE) configuration to extract sensitive information.[1]	While physical access is strictly out-of-scope, OpenPCC enables client validation of SEV-SNP/TDX configuration in the attestation bundle.
Virtualization sandbox vulnerabilities	An attacker could exploit weaknesses in the cloud virtualization and multi-tenant operations to eavesdrop on other compute instances or perform privileged operations across the sandbox boundary.	While cloud virtualization is strictly out-of-scope, OpenPCC enables client validation of the Trusted Execution Environment status through the attestation bundle.
Compute image tampering	An insider could manipulate the final image dispatched to compute nodes, inserting malicious code into an otherwise valid image.	All file system contents are included in the attestation bundle provided to an OpenPCC client, including inference model content, such as weights.

Table 3. Platform Provider Threats and Remediations

Threat Scenario	**Details**	**Mitigations & Remediations**
Inference engine side-channel (prompt/output leak, prompt injection)	A vulnerable inference engine could leak sensitive user content to another user through the inference engine cache inside its sandbox or to other host system components.	Rigorously isolate inference engines from host system components using available security modules, such as SELinux. OpenPCC implementers should review inference engines hosted on ComputeNode instances for tenant isolation, and perform continuous monitoring for identified inference engine side channel vulnerabilities.
Exploitable vulnerabilities in inference engine	With access to a vulnerable inference engine, an attacker could perform arbitrary operations on the host compute instance.	OpenPCC implementers must use strict application sandboxing and immutable filesystems to limit malicious process escalation.
Sandbox escape	An attacker may leverage a vulnerable inference engine to escape an individual user's inference sandbox and access file system content, or perform other privileged operations.	OpenPCC implementers must use strict application sandboxing and immutable filesystems to limit malicious process escalation.
Undesired network behavior	Misconfigured inference engines and compute instances could perform undesired network operations that could act as a side-channel leak for sensitive data.	OpenPCC implementers should include firewall rules limited to pass only necessary system service traffic. Implementors ese may include restrictions in the instance layer, as well as the cloud-provider layer if applicable to provisioned environments.

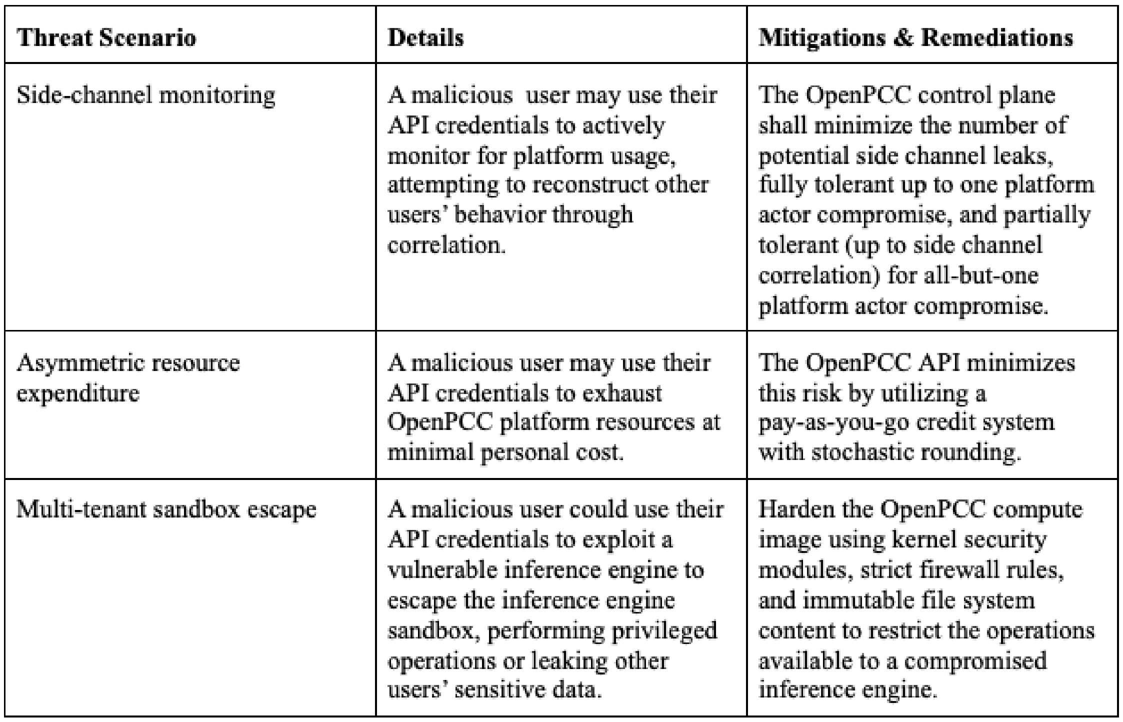
Table 4. Inference Engine Threats and Remediations

Threat Scenario	**Details**	**Mitigations & Remediations**
Remote administration tunnel compromise	An attacker with system administration privileges may use an existing remote tunnel (serial console, ssh, etc.) to compromise the compute instance, if misconfigured.	OpenPCC compute images must disable all identifiable remote management services. This includes, but is not limited to: ssh, serial console, cloud-init. OpenPCC clients should only trust ComputeNode instances with a compliant OpenPCC image manifest record in the transparency log.
Developer credential compromise	Local developer credentials (private keys, auth tokens, etc.), in certain circumstances, may be used to tamper with cloud services and instances.	Restrict OpenPCC related developer credentials following the principle of least authority, and ensure all running compute nodes are inaccessible regardless of possessing private keys/secrets.
Insider threat – malicious software patch	An insider threat may push software patches with vulnerabilities or backdoors into the stack, to exploit at a later time.	to increase the visibility of such compromise attempts, open-source core components of the OpenPCC stack so that anyone may audit it. Moreover, implementers may employ review practices for all patches to OpenPCC components.

Table 5. System Administrator Threats and Remediations

2.6.5. Malicious Users

It is essential to consider malicious users who attempt to exploit an OpenPCC system to acquire side-channel information from other users or directly compromise other OpenPCC infrastructure components. To manage this risk, OpenPCC providers must implement a capability-based permission model to scope organization and user privileges. Organizations and users are only able to see information relevant to the models visible within their scope, and are only able to request inference from those specific models. Additionally, OpenPCC's anonymization scheme must prevent users from being able to track or monitor the usage of other users.



Threat Scenario	Details	Mitigations & Remediations
Side-channel monitoring	A malicious user may use their API credentials to actively monitor for platform usage, attempting to reconstruct other users' behavior through correlation.	The OpenPCC control plane shall minimize the number of potential side channel leaks, fully tolerant up to one platform actor compromise, and partially tolerant (up to side channel correlation) for all-but-one platform actor compromise.
Asymmetric resource expenditure	A malicious user may use their API credentials to exhaust OpenPCC platform resources at minimal personal cost.	The OpenPCC API minimizes this risk by utilizing a pay-as-you-go credit system with stochastic rounding.
Multi-tenant sandbox escape	A malicious user could use their API credentials to exploit a vulnerable inference engine to escape the inference engine sandbox, performing privileged operations or leaking other users' sensitive data.	Harden the OpenPCC compute image using kernel security modules, strict firewall rules, and immutable file system content to restrict the operations available to a compromised inference engine.

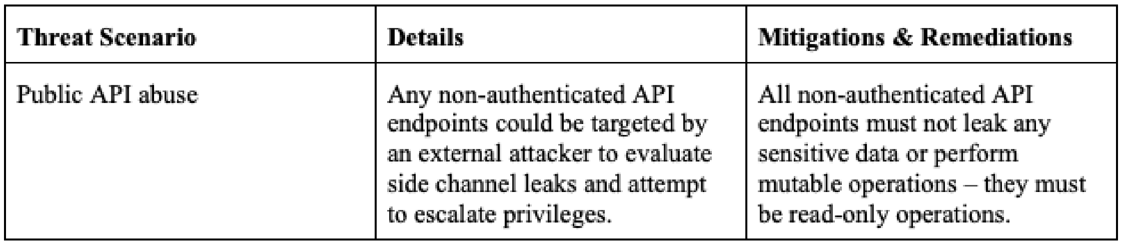
Table 6. Malicious User Threats and Remediations

2.6.6. External Attackers

Threats from External Attackers are modeled as originating from a low-privileged, publicly accessible attack surface.

3. Implementations

The OpenPCC project provides Apache 2.0-licensed open-source implementations of the Client, AuthBank, Relay, Gateway, BlindBank, and Router components, along with the necessary shared libraries, protobufs, and tools, in the OpenPCC GitHub Repo. An



Threat Scenario	Details	Mitigations & Remediations
Public API abuse	Any non-authenticated API endpoints could be targeted by an external attacker to evaluate side channel leaks and attempt to escalate privileges.	All non-authenticated API endpoints must not leak any sensitive data or perform mutable operations – they must be read-only operations.

Table 7. External Attacker Threats and Remediations

OpenPCC implementer will need to adopt the subset of the ComputeNode goals that meets their offering's needs. A production-ready ComputeNode implementation exists under a source-available license from Confident Security.

In production services, an unrelated third party must operate the Relay component to ensure Relay request logs are not available to the primary operator running the other components. At the time of publication, Fastly, Oblivious Network, and Cloudflare offer commercial OHTTP Relay services.

3.1. OpenPCC Client SDKs

Python Client SDK github.com/confidentsecurity/confsec-py

Typescript & Javascript SDK github.com/confidentsecurity/confsec-js

Go Client SDK github.com/openpcc/openpcc

3.2. OpenPCC System Components

AuthBank, Relay, Gateway, BlindBank, Router github.com/openpcc/openpcc

ComputeNode github.com/confidentsecurity/confidentcompute

3.2.1. Cross-component Libraries

Binary HTTP (Go) github.com/confidentsecurity/bhttp

Oblivious HTTP (Go) github.com/confidentsecurity/ohttp

Two-way HPKE (Go) github.com/confidentsecurity/twoway

NVIDIA Trusted Compute (Go) github.com/confidentsecurity/go-nvtrust

3.3. CONFSEC, an End-to-End OpenPCC Service

Confident Security operates CONFSEC, a managed OpenPCC service. It includes a third-party Relay, as well as every other component of the OpenPCC system. As part of CONFSEC, Confident Security provides a source-available ComputeNode, fulfilling the requirements of the Confident-Compute standard described above. For more information about the CONFSEC end-to-end OpenPCC service from Confident Security, see confident.security.

4. Appendix A: OpenPCC HTTP API Reference

All OpenPCC API endpoint requests are serialized as Protobufs using the schema definitions found in the OpenPCC reference implementation. All endpoints except for authentication and identified banking are accessed via OHTTP.

4.1. Auth Config and Withdrawal

The Auth Config and Withdrawal operations are how clients initially verify their identity with OpenPCC and withdraw funds from the *AuthBank*, to be exchanged with Credits. These API endpoints do not route through OHTTP, as a direct binding to user identity is required to perform initial authentication.

4.1.1. Auth Config Details

Using their provisioned API key, negotiated out-of-band, the client issues an Auth Request to the *AuthBank* service, providing their API key. The *AuthBank* must then verify the API key and respond.

The API key must contain the following:

- An organization identifier
- A user identity label
- An expiry timestamp

After validating the client's identity, the *AuthBank* must respond with an AuthConfigResponse, including the following:

- The OHTTP *Relay* URL(s)
- The OHTTP trust bundle for encrypting the OHTTP body so that only the Gateway can decrypt its content.
– Must be present in the transparency log
- The CurrencyKey bundle, used for validating the blinded Credits issued by the *BlindBank*

4.1.2. Auth Withdrawal Details

After receiving the OpenPCC routing configuration and key bundles in the Auth Config operation, a client can then perform an Auth Withdrawal operation to receive Credits for later deposit in a *BlindBank*, along with the *user badge* – a signed capability-based authorization structure.

The Auth Withdrawal request must contain the following:

- The client’s API key
- The amount of funds to withdraw in the form of Credits)

Upon validating the client’s API key and remaining balance, the *AuthBank* must respond with the following:

- Blinded Credits corresponding to the amount of funds withdrawn
- The user badge, denoting which inference models they are able to query
  - Includes the list of permitted models signed by the *AuthBank*

4.1.3. Auth Refund

The Auth Refund operation allows a client to re-deposit Credits into their reserve of funds. Note that this endpoint should not support refunds greater than the total balance of funds tied to their real user identity.

Protocol Details. To issue an Auth Refund request, the client must include the following:

- The client’s API key
- The Credit to refund

Upon validating the client’s API key and remaining balance, the *AuthBank* must replenish the client’s funds. Management of funds is considered out-of-scope for this specification and is left as an implementation detail for managed services implementing OpenPCC.

4.1.4. Bank Refund Deposit, Withdrawal

The Bank Deposit and Withdrawal operations allow a client to deposit Credits withdrawn from the *AuthBank* to a pseudonymous bank account and withdraw Credits from an account, further deidentifying them. These bank accounts serve as a means for clients to split their transactions between different “bins” while being decoupled from real user identity. These transactions occur over OHTTP to strip identifying network metadata.

4.1.5. Bank Deposit Details

The client can deposit Credits into a pseudonym account using the Bank Deposit operation. This allows clients to bin their Credits into pools and further decouple the relationship between the original funds withdrawn and the Credits deposited. **Note that pseudonym accounts tokens are essentially bearer tokens – any user with access to their value(s) is able to operate on the given account(s). For this reason, we caution against re-using account tokens or accidentally publishing them to public repositories.**

The Bank Deposit request must contain the following:

- The pseudonym account token
- The Credits to deposit

The *BlindBank* must then place the deposit in the designated blind bank account, allocating one if necessary. Because of the cryptographic guarantees of the Credits themselves, expectations or management of bank accounts is considered out-of-scope for the OpenPCC specification and left as an implementation detail for compliant *BlindBanks* implementing OpenPCC.

4.1.6. Bank Withdrawal Details

A client can then perform an Bank Withdrawal operation to retrieve blinded Credits from the *BlindBank*.

The Bank Withdrawal request must contain the following:

- The pseudonym account token
- The amount of Credits to withdraw

The *BlindBank* must issue a blinded Credit equal to the amount withdrawn to the client, up to the total amount of Credits present.

4.1.7. Unblinded Full Withdrawal

An Unblinded Full Withdrawal is a variant of the Bank Withdrawal operation, allowing clients to withdraw the full blind bank account balance, with stochastic rounding. This is returned to the client as an unblinded credit, which must be exchanged in the same manner as a refund Credit.

4.2. Compute Request

The Compute Request is the central component of the OpenPCC API. The high-level flow is as follows:

- Login (one-time): The client authenticates with the *AuthBank* and receives Credits.
- Deposit Credits: The client deposits Credits into an anonymous *BlindBank* account.
- Withdraw Credits: The client withdraws Credits from the specified *BlindBank* account.
- List Available Nodes: The client fetches the list of available *ComputeNode* attestation bundles from the Router.
  - Validate Attestation Bundle: The client selects the set of verified *ComputeNode*s they wish to use and validates their attestation bundles, using the information provided in the transparency log.

- Compute Request: The client, after validating the chosen *ComputeNode*(s), dispatches an inference request, providing Credits; the *Router* randomly selects a candidate *ComputeNode* to use, and returns the inference result along with a refund Credit, which can be re-deposited.

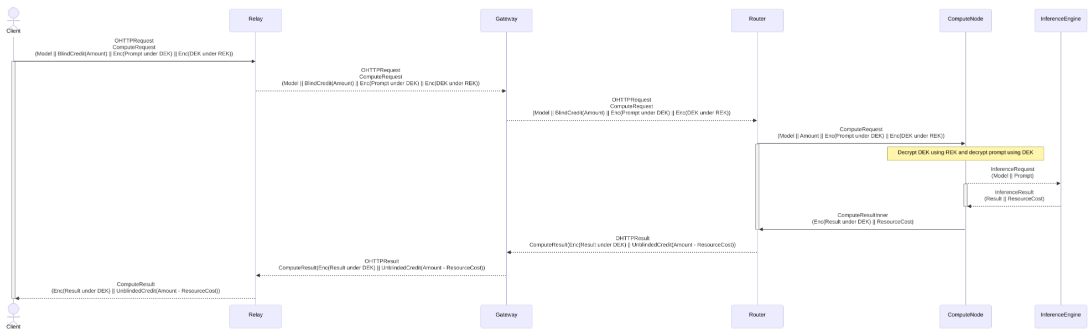

Client
Relay
Gateway
Router
ComputeNode
InferenceEngine
OHTTPRequest
ComputeRequest
(Model || BlindCredit(Amount) || Enc(Prompt under DEK) || Enc(DEK under REK))
OHTTPRequest
ComputeRequest
(Model || BlindCredit(Amount) || Enc(Prompt under DEK) || Enc(DEK under REK))
OHTTPRequest
ComputeRequest
(Model || BlindCredit(Amount) || Enc(Prompt under DEK) || Enc(DEK under REK))
ComputeRequest
(Model || Amount || Enc(Prompt under DEK) || Enc(DEK under REK))
Decrypt DEK using REK and decrypt prompt using DEK
InferenceRequest
(Model || Prompt)
InferenceResult
(Result || ResourceCost)
ComputeResultInner
(Enc(Result under DEK) || ResourceCost)
OHTTPResult
ComputeResult(Enc(Result under DEK) || UnblindedCredit(Amount - ResourceCost))
OHTTPResult
ComputeResult(Enc(Result under DEK) || UnblindedCredit(Amount - ResourceCost))
ComputeResult
(Enc(Result under DEK) || UnblindedCredit(Amount - ResourceCost))
Client
Relay
Gateway
Router
ComputeNode
InferenceEngine

FIGURE 16. Sequence diagram depicting the end-to-end Compute Request network flow, including OHTTP. Note that user prompts are visible only as ciphertext to all parties except the ComputeNode and Inference Engine.
Note: all actions, except for "Login (one-time)" are anonymized over OHTTP.

The Compute Request is the encrypted inference request visible to only the client and the desired *ComputeNode*. Once a client validates the *ComputeNode*(s) attestation bundle, the Compute Request is encrypted by the client and only decrypted and serviced by a specified *ComputeNode*.

Protocol Details This Compute Request API call is itself an HTTP request. A BHTTP encoded version of that HTTP request is encrypted using the per-compute-request Data Encryption Key (DEK).

Each request body itself maps to the target inference engine's request structure. For example, a compute request targeted at a vLLMinstance uses the Open AI Chat Completions API request format in the inner payload.

From the Router's Perspective The Compute Request needs to first be sent through the *Router* node. This requires additional metadata to forward the request correctly. For the *ComputeNode* to be able to decrypt the Compute Request, it needs to access the DEK. The client encrypts the DEK for each *ComputeNode* independently using the Request Encryption Key (REK), which is resident in the *ComputeNode's* TPM. Note that the *Router* can only route to *ComputeNode* instances for which the client is able to generate an encrypted DEK.

The Compute Request, as sent to the Router, is ultimately a streaming HTTP request where the body is the content of the Compute Request and headers contain the target model, Credits, and the REK encrypted DEK for the target *ComputeNode*.

From an OHTTP Perspective To make Compute Requests anonymous, we use OHTTP to blind the IP address (and other routing metadata) of the client making the request. This is a further level of nesting in which the Compute Request, as seen by the Router, is serialized using chunked OHTTP and encrypted using the OHTTP Gateway's public key.

> **OHTTP Relay**
>
> The *Relay* receives the OHTTP request and forwards data to the OHTTP *Gateway*, stripping identifying metadata (source address, timing, headers, etc.).
>
> **OHTTP Gateway**
>
> The *Gateway* de-encapsulates the OHTTP payload and forwards the request to the *Router*.

From the ComputeNode's Perspective The *Router* consumes the allocated Credits and forwards the Compute Request to the target *ComputeNode*. The *ComputeNode* then decrypts the DEK using the REK resident on its TPM.

The *ComputeNode* then uses the DEK to decrypt the payload (prompt) for the Compute Request. The target inference engine is dispatched for the plaintext prompt, and the ComputeNode forwards the encrypted result along with the compute cost back to the *Router*.

The *Router* then issues a refund and terminates the Compute Request by sending the refund and result to the client.

4.2.1. User Permissions & Model Selection

OpenPCC is designed to support plug-and-play inference engines, with first-class support for custom user-provided models that are integrity protected. Currently, OpenPCC targets supporting inference engines that offer access via OpenAI API schema. The selected inference engine must be attested to on the Confident Compute image as immutable file system content. While the initial use-case focuses on a particular HTTP API, OpenPCC's generic HTTP-based protocol generalizes to any available inference engine and even custom user-supplied workloads (e.g., "Bring-Your-Own-Workload").

Custom Models for Existing Inference Engines . A user must specify the model served by an inference engine. Models are available on *ComputeNode* instances, wherein the model weights and metadata are downloaded during an early-stage boot process and a model digest is extended to the compute instance's TPM on PCR 12 for attestation.

OpenPCC allows for a single *ComputeNode* instance to serve multiple models. This mechanism provides an additional mode of obfuscation for the anonymous routing, ensuring that the *Router* cannot directly pinpoint which model a client is requesting, despite knowing the *ComputeNode* to serve the request to.

When requesting a model, the user will query the OpenPCC Router for the set of available inference engine-model pairs. If the desired model is not yet available, OpenPCC recommends an out-of-band mechanism by which the user can upload their custom model.

Selecting a Model . When selecting a model for inference, user clients perform the following steps:

- Authenticate with the AuthBank, get Credits for the session
- Deposit Credits into the BlindBank, get request-sized Credits
- Fetch list of available *ComputeNode* attestation packages from the *Router*, providing a Credit of no value for authentication.
- If the desired model is not available, the user must upload the model out-of-band.
- Validate ComputeNode attestation packages with public keys and transparency log.
- Issue an inference request to the desired ComputeNode through the Router

Privacy Considerations. Users are restricted to querying for attestation packages known to their user account or organization. This prevents malicious users from being able to track usage of domain-specific models that could provide information regarding which users are performing inference and for what purpose they are issuing the inference requests.

User Badges . The *AuthBank* service issues an authorization signature called a badge to a user whenever it authenticates. The badge includes user-level permissions, which may tie to an organizational account if relevant. This badge is included in the Compute Request operation as an HTTP header to authorize the user to query the target model. The badge includes a set of permissions for each user and is signed by the *AuthBank* service. Currently, the user credentials included in a badge are the models they have permission to query and issue Compute Request operations on.

Note that the content of the badge (i.e., the user permissions) are encrypted up until the *ComputeNode*, where they are verified. This prevents intermediate actors from being able to identify a user based on their permissions. The content of a badge is restricted to the models a user is authorized to access, preventing the ComputeNode from directly tying a user badge to a user identity. We recognize that for organizations and users with private models, this can still loosely bind to a user identity. We intend to augment this process to reveal only the models the user is directly requesting. However, the current scheme prevents an attacker from attributing a particular request to an organization or individual, and attributing a particular request to its associated inference model.

However, it is worth noting that OpenPCC provides a set of "public" models that are freely allocated to all users of the platform. Due to the auto-scaling mechanisms used, a malicious user would be able to guess the approximate usage of one of these "public" models based on the number of attestation packages available. For organizations with disproportionately high usage of a particular model(s) relative to other users, this could signal to a malicious user a very rough approximation of model usage when combined with out-of-band information. For this reason, it is recommended that users create private variants of these "public" models with restricted visibility if usage tracking is a concern.

4.3. Go, Typescript, C, and Python Client Libraries

Confident Security provides a Go module and C bindings for the OpenPCC API. Javascript, Typescript, and Python SDKs implementations use a foreign function interface (FFI) to the C bindings.

Documentation and examples are provided in OpenPCC's GitHub repository.