Bellerophon: Non-Interactive Verifier-Free Remote Attestation

Deepak Sirone Jegan
University of Wisconsin–Madison
dsirone@cs.wisc.edu

Ethan Cecchetti
University of Wisconsin–Madison
cecchetti@wisc.edu

Michael Swift
University of Wisconsin–Madison
swift@cs.wisc.edu

***Abstract*—Modern trusted execution environments (TEEs) support trustworthy remote computation with untrusted system software, but launching a job requires verifying the authenticity of the hardware through *remote attestation* and provisioning secrets after launch. Both operations incur round-trip wide-area network communication and require complex always-available replicated infrastructure as part of the software trusted computing base (TCB).**

We present *Bellerophon*, a new remote attestation mechanism that eliminates the need for both trusted verifiers and separate secret provisioning. Bellerophon accomplishes this feat using encrypted binaries embedded with user secrets that can only be decrypted using a manufacturer-provisioned key and only when the TEE is correctly initialized. Moreover, Bellerophon seamlessly integrates with existing approaches to accelerate confidential serverless functions designed to reduce launch overheads. Bellerophon uses Hierarchical Identity-Based Encryption (HIBE) to simplify secret key management and public key distribution, and incorporates a key rotation mechanism for forward security. Our evaluation shows that Bellerophon provides similar security to existing interactive attestation mechanisms with much lower latency.

I. Introduction

Trusted execution environments (TEEs) allow clients to run code securely in a harsh environment: machines controlled by an attacker running malicious system software. Their widespread deployment in recent years has drastically reduced the cost of secure computing in domains including banking [1], [2], health care [3], [4], [5], and blockchains [6], [7], [8]. Major cloud providers, including Amazon, Google and Microsoft, now provide TEE-enabled compute resources [9] based on Intel SGX and TDX and AMD SeV-SNP [10], [11], [12].

Such systems are extremely powerful, but with the TEE enclave itself entirely surrounded by an untrusted operating system, untrusted facilities, and an untrusted network, there is more work to do. To launch an enclave, a client simply sends a piece of code to the cloud provider, which runs it inside a TEE. Since the code passes through the untrusted system, it cannot contain secrets, and before sending secrets, even in encrypted form, a client must verify that (1) the correct code is running, and (2) it is running in a valid enclave that untrusted software cannot pierce, lest an attacker maliciously modify the code or attempt to run it where they can observe its data. Existing TEEs accomplish these goals through *remote attestation*, a protocol that allows a client to verify that the enclave was initialized in the correct state and is running on a genuine TEE-enabled CPU. There are currently two approaches to remote attestation, both with substantial drawbacks.

Traditional attestation, defined by hardware manufacturers like Intel [13], [14] and AMD [15], places the burden of verification fully on the client. The client must receive an attestation from the enclave and verify it with the hardware manufacturer. This model provides very strong security, guarding against an adversarial cloud provider, but clients must run their own verifiers and implement their own secret management, deciding when and how to transmit secrets to a verified enclave. A commercial-grade secret management service such as HashiCorp Vault [16] could be as large as 650K lines of code. Moreover, clients must communicate with each enclave on launch (and to the hardware manufacturer in some attestation schemes [13]), not only requiring them to be online and available, but adding wide-area network latency to the process. This extra latency alone can be prohibitive in applications that require rapid processing of large data, like serverless and high-performance compute workloads. Prior work on protecting serverless functions inside TEEs rely on the client attesting the enclave each time a function is invoked [17], [18], [19] (including the commercial Conclave Cloud Functions [20]). Our measurements show that with a wide-area network latency of 10 ms, attestation increases startup latency 40-400% compared to published AWS cold-start latencies [21].

To solve these scalability and performance problems, commercial deployments, such as those at Microsoft [22] and Amazon [23], place immense trust in the cloud provider. Instead of the client performing their own verification and secret management, the cloud provider does both. Users no longer have to provision secrets to enclaves, and launch avoids wide-area network communication as the cloud provider can place all relevant servers in the same datacenter. However, users must now completely trust the complex services of the cloud provider, reducing the value of TEEs. Intel recently launched a dedicated cloud-based, distributed attestation service called Trust Authority [24] designed to reduce the burden of verification on the users by supporting the offloading of the user's TEE launch policy. In this model, the user still needs to manage secrets and transmit them to the TEE after a successful evaluation of the launch policy by the Trust Authority, and the Trust Authority is a large addition to the user's TCB. Prior work has also used verifiers running inside enclaves [25] running in the same datacenter, removing the infrastructure provider from

the TCB. However, managing the state of verifier enclaves poses the same management challenges as the client running their own verifier, increasing the overall TCB size.

Non-interactive attestation, which does not require online verification, is a promising solution. Previous work on non-interactive remote attestation focused on either deferring the verification of the proof of initial state or relying on periodic verification of the initial state by the remote verifier [26], [27], [28] but does not consider secret provisioning. Furthermore, launch still requires one round trip with the verifier, which is still part of the TCB. Chancel [29] runs a loader enclave that is attested in advance and is entrusted with the secrets needed to decrypt the binaries input by the client. While this approach alleviates the impact of attestation on startup latency, the burden of managing the loader enclaves and provisioning secrets still falls on the client, increasing the size of their TCB.

This work presents Bellerophon, an attestation scheme with the scalability and performance of provider-managed verification and secret management, the traditional threat model trusting the cloud provider only to not deny service, and a smaller trusted computing base (TCB) than prior approaches. The key insight enabling these gains is to encrypt user binaries such that only the TEE inside a genuine and properly-provisioned CPU can correctly decrypt them. As a result, clients can safely include secrets in their initial (encrypted) binary, and be sure *without interactive verification* that only a secure enclave can decrypt them. This assurance eliminates the choice between high trust in the cloud provider and online verification with large network latencies. In addition, it allows binaries to be stored and executed asynchronously, even behind firewalls, at any time at low cost. To maintain compatibility with existing machines capable of running TEEs, Bellerophon does not require any changes to existing cloud machine hardware capable of running TEEs but requires the hardware manufacturer to update their protocols. Each Bellerophon worker machine requires a persistent and tamper-proof key store that is protected from untrusted software, so we assume a hardware Trusted Platform Module (TPM) in our design.

Designing an encryption-based attestation scheme presents important challenges. First, the successful decryption of the user's binary must depend on the TEE being loaded correctly on a platform with the correct firmware version, just as today's TEE's rely on remote verification to run successfully. This requires great care in the design and, as we discuss in Section IV, is not scalable using a standard public-key encryption scheme. Second, as with any protocol sending encrypted secrets through an untrusted channel—in this case a cloud provider—the scheme should be forward secure. That is, a compromise of key material should not leak secrets from prior to the compromise, even if the attacker stored previously encrypted data.

We implement a Bellerophon prototype on Intel SGX enclaves, and show that the Bellerophon's TCB is small: just over 1,000 lines of code for a local architectural enclave for decryption, and 38.6K lines in total including all the other local architectural enclaves, library code and the provisioning server. The addition of a hardware TPM into the TCB is in line with the usage of TPMs for VM attestation in prior academic work and real cloud deployments [30], [31], [32]. Our experiments compare Bellerophon's against SGX-style interactive attestation (also applicable to virtual machine enclaves such as Intel TDX [11] and AMD SEV-SNP [12]) and show that Bellerophon reduces enclave startup latency for a 40 MB TEE binary by 94% when the SGX verifier requires a 10ms network round trip. To demonstrate the effectiveness of Bellerophon in an end-to-end application, we integrate Bellerophon with the reusable enclaves framework [19] to accelerate the startup performance of confidential serverless functions.

The main contributions of the paper are:

- We describe an attestation scheme based on encryption that is non-interactive, verifier-free, and with the same threat model as that of interactive attestation. The core insight is the design of a set of architectural enclaves and efficient key management that aid in performing decryption, in contrast to the signing architectural enclaves used in interactive attestation.
- We design a key-management mechanism for TEEs based on HIBE that removes communication with the hardware manufacturer during attestation.
- We design a load balancing scheme for Bellerophon that enables the infrastructure provider to run an encrypted binary on any machine they own. We also design a re-encryption scheme so that the users do not need to re-encrypt binaries for maintaining forward secrecy.
- We implement a Bellerophon prototype on Intel SGX and show Bellerophon's TCB is smaller and performs better than prior mechanisms by integrating with prior work [19] and running microbenchmarks. Further, we describe how Bellerophon is implemented on Intel TDX (Appendix B) and AMD SEV-SNP (Appendix C).

II. Background

A. TEEs and Remote Attestation

A Trusted Execution Environment (TEE) is a combination of hardware and firmware that provides both confidentiality and integrity protections even against malicious system software. To ensure a specified computation runs correctly, the TEE prevents system software from accessing the TEE's private data or changing the execution flow of code running in a TEE. These guarantees allow a user to be confident that code will execute as intended and not leak any secrets passed in during computation. The running code and its associated data isolated by a TEE is termed an *enclave*.

For these guarantees to be useful, users need to verify that specified code was properly loaded onto trustworthy hardware. This process, known as *remote attestation*, verifies critical properties of both the enclave state and the hardware. The enclave state includes the entire configuration of its address space (page contents, relative offsets, and permissions), defining precisely what it will do if executed correctly. The hardware state includes the security version number of the CPU, ensuring

it properly supports remote attestation and trusted execution, and the version of the associated firmware. The firmware consists of the CPU microcode, and a variety of *architectural enclaves*—software signed by the hardware manufacturer, allowing it to derive the keys needed in the attestation process without external verification, that implements more complex portions of attestation protocols. The full initial hardware and software state is provided by a hardware *measurement*, which returns a hash of all relevant values.

Existing systems perform remote attestation as an interactive protocol. We detail the protocol for Intel SGX [13] and Intel TDX [14] here, please refer to the Appendix for the specifics of AMD SEV-SNP which are similar.

Provisioning. A TEE user must be able to verify that the hardware running a TEE can be trusted to maintain its security, which means that the user can verify that the hardware comes from a trusted manufacturer. To facilitate this proof, each machine running TEEs is manufactured with an embedded secret called a *provisioning key* shared with its manufacturer. The provisioning key is only accessible to a manufacturer-provided *provisioning enclave* and is specific to the current firmware version of the machine. Intel EPID [33] has a provisioning step where an attestation specific secret is generated by the hardware provider and sent back to the machine being provisioned. During provisioning, the machine proves its knowledge of the provisioning key to a provisioning service run by the manufacturer—allowing the service to check the validity of the hardware—and provides its local *firmware version*—ensuring the CPU is not running outdated firmware with known vulnerabilities (since the provisioning key accessible to the provisioning enclave depends on the currently running firmware version). For Intel SGX EPID, the secret is a group signing key that can be used to generate signatures over a hash of the initial state of an enclave. To store the secret across reboots, the processor encrypts the secret key with a local sealing key known only to the processor and only accessible with the current firmware version, and stores the encrypted key on local untrusted storage. At the end of provisioning, a machine has a signing key tied to a specific manufacturer and firmware version. Provisioning provides the opportunity for the hardware manufacturer to check for additional attributes of the running machine such as machine ownership which is required for the design of Bellerophon.

In Intel ECDSA-based attestation [34] and AMD SEV-SNP attestation [15], the hardware manufacturer provides certificates for the provisioning certification keys (PCKs) accessible to the provisioning certification enclave (or the AMD Platform Security Processor (PSP) in the case of SEV-SNP) with the hardware manufacturer as the root authority and skips the provisioning step, rendering the provisioning certification key unable to prove machine ownership.

Runtime Protocol. When a TEE launches an enclave, it connects to the *quoting enclave*, an architectural enclave with access to the provisioned secret in the case of Intel EPID or the quoting key in the case of Intel ECDSA, to generate a proof of the hardware and initial enclave state that attests to its correctness.

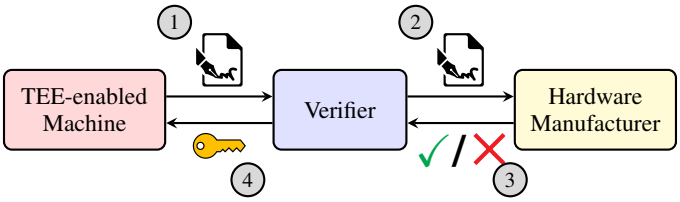

Fig. 1: Structure of interactive remote attestation: (1) The TEE-enabled machine sends the verifier a quote containing a proof of the initial user TEE state. (2) The verifier forwards the proof to the trusted hardware manufacturer, which (3) responds with the proof's validity. (4) If the proof is valid, the verifier provisions secrets into the TEE.

In the Intel EPID scheme, the proof consists of a measurement of the user enclave signed with a provisioned signing key, whereas in Intel ECDSA attestation, a quoting key that is signed by the provisioning certification key is used. Since the connection media between the quoting enclave and the user enclave is controlled by the untrusted OS, the connection is confidentiality, integrity and replay protected using an authenticated ECDH key exchange protocol documented by Intel. All enclave-to-enclave communication in Bellerophon use this same protocol ([35] and Appendix A). This protocol also convinces both enclaves of each other's hardware measurements in a trustworthy manner.

Figure 1 shows how the attestation procedure proceeds using this proof. First, the TEE provides the proof to a *verifier* running on behalf of the user, which then contacts the hardware provider to check its validity. If the proof is valid, the verifier can send secrets (e.g., decryption keys or access tokens) which the user code running in the enclave needs to perform its intended function.

This interactivity is useful for two reasons. First, it enables fresh (ephemeral) secrets on every enclave launch to ensure *forward security*, the guarantee that future compromise of long-term key material (e.g., the provisioned secret) cannot compromise past communications. Second, since the provisioned key is sealed using a key associated with the firmware version, interactive verification can detect rollback to old key and firmware versions. However, it also imposes significant slowdowns, as each message incurs a wide-area network delay.

Compromise Recovery. If the manufacturer learns of a flaw in the hardware/software TCB of a machine, it can issue TCB updates. In Intel EPID, machines that are patched re-run the provisioning protocol to obtain a new group signing key corresponding to the new firmware version. In Intel ECDSA, the hardware manufacturer issues certificates for the new PCKs corresponding to the updated firmware version. Likewise, the manufacturer can fail verification of any machine known to be compromised.

B. Hierarchical Identity-Based Encryption

Bellerophon builds on Hierarchical Identity-Based Encryption (HIBE), a public key encryption scheme where each party is assigned a hierarchical identity, any public key is derivable from global public key material and the target party's identity, and private keys are derivable down the hierarchy. More precisely, an identity id is a sequence of byte strings $[ID_1, \ldots, ID_n]$ acting as identifiers. Encrypting a message for party id requires only id and some global public key material. The *master* secret decryption key for the root identity $[\,]$ is generated randomly, and the $\mathrm{DeriveSKey}()$ API generates the decryption key for any of its descendants in the identity hierarchy. That is, given decryption key sk for identity $[ID_1, \ldots, ID_k]$, $\mathrm{DeriveSKey}(sk, [ID_{k+1}, \ldots, ID_n])$ will produce the decryption key for identity $[ID_1, \ldots, ID_n]$.

This feature enables a hierarchy of key servers. A top-level key server can delegate private key generation for a subset of the ID space. For example, a hardware manufacturer "AMD" could delegate key generation to a cloud provider "AWS" for key identifiers beginning ["AMD", "AWS"]. Clients only need the global public HIBE material (parameters for this specific instantiation of HIBE) and a machine's identity to encrypt a message for that machine.

III. Design Considerations

We have three primary goals for Bellerophon: (1) minimize the amount of trusted code required for remote attestation; (2) make enclave launch non-interactive, removing the performance overhead of network communication and allowing clients to be offline when enclaves launch; (3) retain the strong security guarantees of interactive remote attestation, We first describe the adversarial model for Bellerophon, and then specify our security and functionality goals more precisely.

A. Adversarial Model and Compromise Recovery

The Bellerophon threat model is largely based on the standard model from the TEE literature [36]. TEE-enabled machines are separated into the self-contained TEE and everything else—including the operating system, network facilities, and storage—referred to as the "untrusted OS." We assume that TEEs, once properly provisioned, will execute correctly and not leak anything—including cryptographic key material—to the untrusted OS not explicitly passed out of the TEE by the software. It is the job of hardware (e.g., SGX, TDX, or SEV) to prevent the untrusted OS from interfering with TEE execution, and Bellerophon does not change the hardware isolation mechanisms. TEEs isolate user code so they cannot access other TEEs or modify the untrusted OS directly. However, the user code can attempt to escape the TEE by utilizing software bugs.

TEE implementations may defend against additional threat models, such as physical attackers that can snoop the memory bus or read/write data in memory beyond the control of the processor. By building on top of existing TEE mechanisms, Bellerophon adopts the threat model of the TEE it builds on. For example, when building on a TEE without memory encryption (e.g., Keystone [37]) the system maybe vulnerable to memory bus snooping, while still defending against an untrusted OS. Notably, we do assume the hardware TPM on each machine is trusted and tamper-proof.

The Bellerophon system consists of three types of parties.

1) A hardware provider (e.g., Intel or AMD) who manufactures and distributes TEE hardware, provisions private key material, and publishes public keys. This hardware provider is fully trusted.
2) An infrastructure provider (cloud service) who controls the physical machines and their untrusted OSes. The infrastructure provider acts as an active adversary and can deviate from all Bellerophon protocols. In particular, the infrastructure provider is not trusted to trigger key rotations for forward security (Section III-B). Defending against denial of service is beyond the scope of Bellerophon
3) Clients who request jobs. Clients may attack each other and the infrastructure provider, but will not attack themselves.

We assume that the hardware provider and the infrastructure provider not collude to compromise user enclaves. Some infrastructure providers manufacture their own TEE hardware such as AWS Nitro and they are outside the scope of this work.

Compromise Recovery. In a practical context, some level of compromise is usually inevitable.

To that end, compromise of user code running inside an enclave or of provisioned secret keys for an individual cloud machine must be recoverable, with the damage limited only to the compromised job, or jobs running on the compromised machine near in time to when the compromise occurred, respectively. Compromise of burnt-in secrets within a CPU are unrecoverable for that CPU, but individual CPUs can be revoked, limiting the damage.

We do not consider the ramifications of compromises of any aspect of the hardware provider or of the infrastructure provider's long-term identity keys.

Cryptographic Assumptions. We assume that the HIBE scheme is atleast selectively identity secure against chosen plaintext attacks, that is IND-sID-CPA [38], [39]. This assumption means that without the secret key it is computationally hard to learn information from any ciphertexts corresponding to chosen plaintexts when the public key identifiers used for generating the ciphertexts is limited in number (corresponding to the number of hardware manufacturers in existence; the name of the hardware manufacturer forms the first identifier in all identities used in Bellerophon). In our implementation we use the BBG HIBE scheme [40], which is IND-sID-CPA secure. To encrypt and integrity protect long messages with HIBE, we assume a Key Encapsulation Mechanism (KEM) based scheme where a random element in the message space of the HIBE scheme is used to generate a symmetric key. We rely on a symmetric encryption scheme that provides authenticated encryption with associated data that is assumed to be both IND-CPA [41] and INT-CTXT [41], that is without the secret key, it is hard to forge a tag for an attacker generated message

given polynomially many chosen plaintext-ciphertext pairs. In our implementation we use AES-GCM.

B. Security and Functionality Goals

We categorise the objectives of Bellerophon into two classes: security goals and functionality (or deployability) goals.

Security goals.

- **S1:** Only enclaves initialized correctly and running on a genuine CPU can access secrets needed for their execution. This prevents the untrusted OS from accessing user secrets.
- **S2:** The user TEE code and data is protected from access by the untrusted system software. This is distinct from S1 that requires protecting secrets.
- **S3:** Enclaves can only access secrets if the hardware and platform software have the specified firmware version. Clients can ensure that TCBs with known vulnerabilities cannot access their secrets.
- **S4:** TEE binaries encrypted in the past are not decryptable by an adversary who compromises a newer decryption key (forward security).
- **S5:** Enclaves can only run in TEEs authorized by the client-specified infrastructure provider.

These security goals correspond to the security goals of current remote attestation mechanisms with interactive verification. However, non-interactivity brings with it a set of unique limitations. First, secrets are embedded inside the binaries, so secrets are long-lived compared to the ephemeral secrets provided by regular attestation that last only as long as the enclave runs. All the secrets that the enclave has access to are part of the encrypted TEE binary. Second, any policies that decide whether or where an enclave can be launched must be encoded in the encryption mechanism when the TEE is prepared, as there is no longer an interactive service that can enforce policy in real time.

Functionality Goals.

- **F1:** Remote machines can load and run enclaves using only local information, with no communication to external services. This reduces the launch latency and reduces the TCB by removing the code to run a highly available verifier service.
- **F2:** The infrastructure provider can run an enclave on a group of machines, not just one. This ensures TEE execution can be a scalable cloud service offering.
- **F3:** The public encryption key for an individual machine must be accessible in a trustworthy way without contacting the hardware provider. This requirement avoids the need for the hardware manufacturer to run a publicly accessible, high volume key server. Ideally, anyone requiring the public key should be able to compute it using public information it read once from the hardware manufacturer.
- **F4:** After a recoverable TCB compromise (see Section III-A) or patchable flaw discovery, it is possible to patch and reprovision a machine, allowing the machine to safely continue operation.

C. Intel EPID Deprecation

The Bellerophon provisioning design is based on Intel EPID [33], which has been deprecated [42] in favor of a certificate-based attestation. The change was made to remove the need to contact Intel for each verification operation, which hinders operation in (R1) client networks that are not connected to the Internet, (R2) entities that are risk-averse in outsourcing trust decisions to 3rd parties, (R3) distributed systems (such as P2P systems) where a single point of verification is suboptimal or a scalability limit, and (R4) environments that conflict with EPID's privacy properties [43]. While Bellerophon uses a similar provisioning protocol for key distribution to achieve forward security (see Section IV-C), Bellerophon avoids the concerns that led to EPID's deprecation: Bellerophon is non-interactive and verifier free (R1 and R2), it only requires one-time communication between clients and the hardware manufacturer, making is scalable (R3), and it follows certificate-based attestation in making machine identifiers public (R4).

IV. Bellerophon Design

Bellerophon provides non-interactivity by embedding user secrets in binaries and encrypting those binaries such that only a valid TEE is able to decrypt them. This approach eliminates interactive attestation and secret management, and even allows for storing binaries and executing them later when the client may be offline.

A. Preparing and Running a Binary

To deploy a computation in a TEE, a user packages their code and secrets into a single self-contained binary bin and prepares it for execution using the protocol in Figure 2. They encrypt bin into an *encrypted blob* using a freshly generated symmetric key κ. They prepend an *decryption stub* to this encrypted blob that, when loaded in a TEE correctly, decrypts the blob and starts executing the TEE binary with its secrets. The user generates a hash H of the encrypted blob and decryption stub to form an *authenticator* by optionally computing additional data ϕ of the same size as H and encrypting $\kappa \parallel H \parallel \phi$ with a public encryption key whose corresponding decryption key is known only to the remote machine. In our implementation on Intel SGX, we modify the Intel Protected Code Loader (PCL) [44] package to perform the encryption of each section of the binary (except the decryption stub), adding write permissions to enable in-place decryption and storing the original permissions in a table in the binary. We

Binary Preparation

On input $(bin, stub, pk, \phi)$*:*
$\kappa \leftarrow \mathrm{SymKGen}(1^\lambda)$
$blob \leftarrow \mathrm{SymEnc}(\kappa, bin)$
$H \leftarrow \mathrm{Hash}(stub \parallel blob)$
$auth \leftarrow \mathrm{PkEnc}(pk, \kappa \parallel H \parallel \phi)$
output $(stub, blob, auth)$

Fig. 2: Binary preparation protocol

Decryption Enclave

On input $(auth, \phi)$ *from enclave* $\mathcal{E}$*:*
$sk \leftarrow \text{LoadKey}()$
$\kappa \parallel H \parallel \phi' \leftarrow \text{PkDec}(sk, auth)$
$H' \leftarrow \text{verifyMeasurement}(\mathcal{E}, H)$
if $H = H'$ **and** $\phi = \phi'$ **output** κ **to** $\mathcal{E}$
else abort

Fig. 3: Decryption enclave protocol

modify the provided decryption stub to restore the original page permissions after decryption using the SGX Enclave Memory Manager [45]. See the Appendix (B and C) for how this is implemented in Intel TDX and AMD SEV-SNP.

The additional data ϕ allows the enclave code at runtime to specify the purpose of this decryption request, such as decrypting the initial enclave code or interpreting a specific encrypted request. By doing so, it supports chained verification. See Section VI for an example.

To ensure the CPU is genuine, the user must obtain the public key for the target machine from the hardware manufacturer in advance. The hash H authenticates the material prepared by the user, and the public key ensures that only a specific, trusted machine can decrypt and execute the code. The decryption stub, encrypted TEE binary with user secrets, additional data ϕ and authenticator comprise a *TEE package*.

The TEE platform on a remote machine ensures that only a correct TEE package can execute correctly. Inauthentic hardware or corrupted packages lead to decryption failures, so code and user secrets in the package are inaccessible. The platform provides an architectural *decryption enclave* that has access to the decryption key corresponding to the public key used by the user. As mentioned in Section II, the decryption enclave is architectural and does not have to be attested by the user. The untrusted OS on the remote machine loads the decryption stub and encrypted blob into an enclave and starts the enclave. After finalizing the initial memory state of the enclave, the untrusted OS executes the decryption stub with the authenticator as an argument. The stub connects to the decryption enclave, passing the authenticator and enclave-generated data ϕ as arguments. Note that all the communication between the decryption stub and the decryption enclave is encrypted with integrity and replay protection through the use of the authenticated ECDH key exchange protocol documented by Intel where the decryption stub can verify the measurement of the decryption enclave it is connecting to ([35] and Appendix A). This protocol also lets the decryption enclave access the measurement of the enclave connecting to it in a trustworthy manner. As shown in Figure 3, the decryption enclave loads its secret decryption key, decrypts the authenticator, and verifies that the caller's measurement (obtained from its attestation report during the channel setup) and the enclave-generated data match the respective values in the authenticator. Matching hashes ($H = H'$) indicates the expected decryption stub and encrypted blob were properly loaded into memory. Matching additional data ($\phi = \phi'$) indicates the running enclave code and the user agree on the purpose of this request. When both match, the decryption enclave returns the symmetric key from the authenticator to the decryption stub, which can then decrypt and run the encrypted blob containing TEE code and secrets. If any of these conditions do not hold or if the decryption of the authenticator fails, then the protocol is aborted.

This basic functionality is sufficient to accomplish four of our goals. Both S1 and S2 are satisfied since (a) the authenticator can only be decrypted by a trusted CPU, and (b) the full TEE code and data, including any secrets, remain encrypted until they are correctly loaded into an enclave in an isolated TEE environment. Hence, neither an untrusted machine nor the untrusted OS on a trusted CPU can access user code and secrets. We also ensure S3, because a machine with the wrong firmware version will not have the decryption key for the authenticator. Finally, because a TEE package contains its own authenticator, it can be stored and executed later, even if the client is offline or the machine is disconnected from the network, accomplishing goal F1.

B. Scalable Key Management

With a classic asymmetric cryptosystem like RSA, the keys for different machines are unrelated to each other. As a result, users must query the trusted hardware manufacturer every time they wish to run a job on a different machine, violating goal F3. Perhaps even worse, Intel SXG EPID [33] required users to contact the hardware provider for *each remote attestation*. Certificate-based schemes, like Intel ECDSA [34] and AMD SEV-SNP [15] allow infrastructure providers to cache certificates for specific machines, removing the need to contact the hardware provider directly. However, they offer no forward security—keys rotate only when the firmware version is updated.

Bellerophon avoids this trade-off using Hierarchical Identity-Based Encryption (HIBE), with the hardware manufacturer controlling the root master keys. Each machine receives a unique identity and associated HIBE private decryption key, making compromise recovery simpler and easier. The key material needed to encrypt an authenticator for machine (pk in Figure 2) is simply the global HIBE public key material and the identity of the target machine. Public key distribution is also extremely straightforward: the hardware manufacturer makes the global HIBE public parameters available. Clients use the same global HIBE public parameters for every machine, so they only ever need to retrieve them once to derive public keys for any machine. They do have to ask the cloud provider for the identity of a target machine.

An identity id consists of the hardware manufacturer's name, a firmware version, the unique hardware identifier for a CPU,[1] and a public signature verification key $vk_{\mathcal{P}}$ identifying the cloud

[1]Intel combines the firmware version and hardware identifier into one value that changes when the TCB updates. Bellerophon can support this behavior, but it limits clients' ability to noninteractively require up-to-date TCBs.

provider $\mathcal{P}$. A machine's id specifies its security attributes The cloud provider can lie about a machine's id to the client, however the provisioning step ensures that only machines satisfying the attributes in the id have the corresponding HIBE private key.

Each of these components serves a critical function for accomplishing our security goals. Including the firmware version directly in id accomplishes goal S3: a TEE with an out-of-date TCB will only be provisioned (see below) with the decryption key for some id' with the old version number, so it cannot decrypt authenticators encrypted using an id with the current version number. The CPU hardware identifier ensures that each physical machine operates with a different decryption key, limiting the damage from compromise and aiding goal F4 (see Section IV-C). Finally, as we will see below, including $vk_{\mathcal{P}}$ allows Bellerophon to require $\mathcal{P}$ to sign off on provisioning any machine with this identity. The provider can thus ensure that only machines it controls have identities that include its verification key, supporting goal S5.

Provisioning. The main goal of provisioning in Bellerophon is to ensure that only the decryption enclaves running in machines satisfying the attributes in its id get access to its corresponding private key. Recall that all the attributes in id except the machine ownership can be proved by proving the knowledge of the provisioning key. Similar to Intel EPID [33], we assume that the public key of the hardware manufacturer is hardcoded in the provisioning enclave. First, the provisioning enclave establishes a TLS channel with the provisioning service using the hardcoded public key. It then proves knowledge of the provisioning key corresponding to its claimed attributes using Salted Challenge Response Authentication Mechanism (SCRAM), a standard authentication protocol [46]. To verify that a cloud provider $\mathcal{P}$ authorizes provisioning of any machine with $vk_{\mathcal{P}}$ in its identity, the hardware manufacturer interactively requests a signature on a randomly generated nonce from $\mathcal{P}$ as part of the provisioning process. This achieves goal S5. Refer to Section IV-C for storage of the provisioned private key.

C. Forward Security

Forward security ensures that a compromise of a private key at a particular time does not allow an attacker to decrypt objects encrypted at an earlier time [47]. If a single long-term private key were used for a remote machine, compromise of that one key would allow decryption of all binaries ever encrypted for that machine. Bellerophon addresses this problem (S4) by dividing time into *epochs* and rotating keys and destroying old keys every epoch [38]. With key rotation, a key currently in use is unhelpful for decrypting older binaries, greatly limiting the damage from a compromise.

Bellerophon leverages HIBE for two types of time-based key rotations: major and minor. Major epochs define the granularity at which forward security is guaranteed in Bellerophon; a compromised key can, in the worse case, decrypt all binaries encrypted in the same major epoch, but no others. The hardware manufacturer inserts a major epoch counter after the unique hardware identifier for a CPU into the HIBE id for a machine. The manufacturer can efficiently generate new decryption keys by changing the major epoch number in an ID, but machines—and attackers in possession of compromised keys—cannot, as they only have a decryption key for the entire ID. The same provisioning protocol described above is used for major epoch rotations.

Each major epoch is further divided into *minor epochs*. Minor epochs further limit the damage that can be caused by an adversary other than the infrastructure provider within a major epoch; a compromise of a remote machine's key at a particular minor epoch does not allow the adversary to decrypt binaries that were encrypted before it, on average cutting in half the window of compromise. We discuss how major and minor epochs are triggered under the corresponding heading in Section IV-C. For *minor epochs*, the new key is derived from the existing key using HIBE by appending an epoch counter to the identifier sequence. This enables every machine to *independently* and without communication update its key and destroy its old key; the prefix property of HIBE ensures that the new decryption key cannot be used to derive the decryption key for an older epoch [38].

Since epochs are time-based, a user can compute major and minor epoch values independently.

The period of major epochs is configurable by the hardware provider and minor epochs by the cloud provider. If these periods are provided to users, they can independently calculate the current epoch from wall-clock time when preparing a binary for execution. We discuss how to estimate the major and minor epochs based on performance iformation in Section VIII.

Key rotations present several challenges and opportunities.

Transitions. An entire fleet of machines cannot simultaneously transition from one epoch to the next. Bellerophon solves this by storing two in each machine, one for the current epoch and one for the previous epoch. When it stores a new key, Bellerophon permanently deletes an old key.

Efficient representation of epochs As described, each minor epoch lengthens the HIBE identifer sequence. Bellerophon reduces this cost using the HIBE forward security scheme from Canetti et al. [38], that reduces the depth of the HIBE hierarchy to be logarithmic in the total number of minor epochs. Any forward secure HIBE scheme can be used here and we chose this scheme for ease of implementation.

Binary preparation. With epochs, users must include the epoch numbers in the id they use for key generation. Users should encrypt binaries with the last minor epoch when the binary can run. As long as that epoch has not passed, the decryption enclave can derive the associated decryption key from the current epoch's decryption key, allowing the binary to run any time up to the specified epoch.

Re-encrypting binaries. Encrypting with a future epoch number does not work across major epoch boundaries, because the corresponding key is not derivable on the remote machine. As a result, any binary stored for future execution will fail to decrypt after a major epoch rotation.

Bellerophon addresses this with an architectural *reencryption enclave* running on each machine. This enclave has access to the current epoch decryption key and the global public key material. When invoked by the untrusted OS, the reencryption enclave will decrypt the authenticator of a TEE package with the current decryption key and produce a new authenticator encrypted for the next major epoch. The untrusted OS can invoke this service during every major rotation. The system must update all stored binaries before they can run and before the next major epoch, when the old key will be deleted. The Bellerophon authenticator could be extended with a maximum major epoch to limit re-encryptions. In essence, the maximum major epoch is a limit on when the user should provide fresh binaries with updated secrets, and provides a tradeoff between security and functionality.

Provisioning: The hardware manufacturer provisions a new machine using the current major epoch and minor epoch of zero. This allows a machine to advance its minor epoch to the current time since the minor epoch can be computed from the current time. Major epochs will increase the load on the hardware manufacturer, as they now must re-provision machines on every major epoch rather than once at machine installation. This cost can be reduced by increasing the length of a major epoch. Alternatively, the HIBE hierarchy enables delegation of major epoch rotations to an intermediate node, such as the cloud provider, at the cost of trusting that intermediary.

Triggering rotations. A correctly behaving infrastructure provider will periodically trigger minor and major epochs. However, a faulty or malicious infrastructure provider is not guaranteed to enforce minor or major rotations on each machine. A TEE, if it has access to a trusted time source, can perform minor rotations autonomously. An external secure time source could also generate certificates with the current time, allowing the untrusted OS to prove to the TEE that it should rotate its minor epoch forward.

If a provider fails to perform major rotations, the machines will be unable to run new enclaves encrypted with the new major epoch, simply denying service. Users can calculate the major and minor epochs from the current wallclock time. Even if a machine does not advance its major or minor epoch , the users will always rotate to the correct epoch and generate new binaries with the keys for the correct epoch.

A malicious provider can also *freeze* a machine, taking it offline and stopping key rotations. This attack allows the provider more time to break the decryption key and decrypt any TEE packages with the same major epoch, but is not helpful across major epoch boundaries, as a frozen machine cannot be re-provisioned with the next major epoch key. The secrets obtained from the decrypted binaries could be, in the worst case, the same as that of a binary from a prior major epoch. We discuss this more in Section V.

This forward secrecy mechanism address goal S4, that breaking a current key does not allow decrypting binaries encrypted in the past beyond the current major epoch.

TPM-Based Key Storage. In some TEE systems, like Intel SGX, provisioned secrets are stored encrypted on untrusted storage. This approach poses a serious challenge for forward security. Untrusted software can store copies of old ciphertexts, allowing it to roll back the decryption key to an earlier epoch or and decrypt the old keys later if they compromise a core TEE key. Bellerophon addresses this concern by storing provisioned HIBE decryption keys, encrypted under a hardware-derived long term key, in erasable storage provided by a TPM, which we assume is trustworthy. By overwriting this TPM memory on key rotation, Bellerophon can ensure that old keys are not captured, even in encrypted form.

Notably, the communication between the provisioning and key rotation enclaves and the TPM passes through the untrusted system software. We therefore employ a standard forward-secret protocol for TPM communication with an active attacker [48]. The provisioning key is used as a password to access the index in the TPM's non-volatile memory where the provisioned HIBE private key is stored.

Compromise recovery. If an attack on the hardware/software TCB of the machine is known, the hardware manufacturer can issue TCB patches. To ensure old firmware versions are not in use (goal S3), the manufacturer and infrastructure provider must re-provision all patched CPUs and publish the new firmware version to users, who use it to encrypt TEE packages. Such updates cover all the firmware components, even if all of them do not have vulnerabilities. The secrets in binaries stored using a key with the old firmware version should be assumed to be compromised. For non-critical bugs, this re-provisioning can occur as part of the next major epoch rotation, limiting the expense. For critical bugs, however, it may need to happen "off-cycle," making recovery very expensive.

If the hardware manufacturer learns that a single machine is compromised, for example its provisioning secret leaks, the manufacturer can refuse to issue decryption keys to this machine during provisioning at the next major epoch. This prevents the compromised machine from decrypting TEE packages for future major epochs.

D. Load-Balancing

As described, Bellerophon requires users to prepare a binary for a specific machine. This is a poor match for a cloud environment, where a binary may be run on any of a cluster of machines, and the set of machines may change over time. We address this with a *load balancer*. This is an architectural enclave that accepts a TEE package encrypted for one machine and the identity of a second, and re-encrypts the authenticator using the public key of the second machine. Like the decryption enclave and re-encryption enclave, the load balancer is implemented as an enclave, and is authenticated using the same firmware versioning as the other two enclaves. Note that the implementation of the load-balancing enclave is almost identical to the re-encryption enclave. The cloud provider can invoke the load-balancing enclave for any machine satisfying goal F2.

Load balancing is possible within an architectural enclave because HIBE removes the need to know the public key for every machine; the load balancer can use stored global public HIBE key material to generate the public key for re-encryption. To ensure that the target machine is owned by the same provider and is not being redirected to on old, vulnerable machine, the load balancing enclave verifies that only the hardware identifier changes between ids, not the firmware version or provider public key.

Under this design, the user encrypts a TEE package for a specific load balancer instance. Bellerophon could support multiple load balancers transparently by issuing them all the same (virtual) machine ID so they share the same decryption key. While this slightly cuts against our compromise recovery goals, load balancers never run user-specified code, reducing the concern.

V. Security Analysis

We analyze Bellerophon's security to see how it satisfies the security goals in Section III-B with a focus on two primary concerns:

(i) Security with a malicious infrastructure provider.
(ii) Security with a malicious user.

Recall that the compromise of burnt-in CPU secrets is outside the scope of Bellerophon.

A. Malicious Infrastructure Providers

Recall from Section III-A that Bellerophon assumes the infrastructure provider will not deny service, but otherwise treats it as an active adversary, meaning that it might arbitrarily deviate from the Bellerophon protocols.

S1 and S2 (Confidentiality of User Code and Secrets). Without interactive verification, Bellerophon relies on the confidentiality of user code and data to ensure it only executes on valid TEEs, so the same mechanisms accomplish goals S1 and S2. Recall from Section IV-A that each user workload is encrypted under a fresh secret key κ included in the encrypted authenticator, so these goals reduce to keeping κ secure.

The channel between the user enclave and the decryption enclave is confidential and replay protected using the authenticated ECDH key exchange protocol documented by Intel ([35] and Appendix A) so any leak of κ must stem from the user enclave, a leaky load balancer, or leak of the decryption enclave's HIBE decryption key. A correct user enclave is assumed to not leak its own key or secrets. The decryption enclave verifies the initial measurement of the user enclave—the encrypted blob and decryption stub—against the hash provided in the authenticator, thus guaranteeing that κ is only released to correct user enclave code. The load balancer will only ever re-encrypt the authenticator under the identity of a different machine, so that simply reduces to compromising the HIBE key of the target machine.

The security of the provisioned HIBE encryption key stems directly from the security of the TEE hardware and provisioning protocol. The genuineness of the CPU and the firmware version is verified by proving knowledge of the provisioning key. The provisioning key a machine uses can only be derived by a provisioning enclave (i) if it runs on a CPU having a genuine burnt-in root provisioning key and (ii) if the provisioning enclave (with the claimed firmware version) is signed by the hardware provider. The provisioning enclave will not intentionally leak the provisioning key. Therefore the provisioned key accurately reflects the current firmware version.

Intel EPID sealed provisioned keys using a key associated with the firmware version, and interactive verification can detect rollback to previous key and firmware versions. Bellerophon stores the provisioned key in a TPM to avoid downgrade attacks without interactive verification.

The decryption enclave never leaks the output of an incorrect decryption of a HIBE authenticator to the cloud provider, thereby ruling out chosen ciphertext attacks. The security assumption of IND-sID-CPA on the HIBE scheme rules out chosen plaintext attacks for all the identifiers used in Bellerophon.

A malicious infrastructure provider cannot recover κ without violating one of Bellerophon's security assumptions. It therefore cannot decipher the encrypted blob provided by a user, and is unable to extract any user secrets or the code necessary to improperly run the job.

S3 (Correct Firmware Version). This goal stems directly from including the firmware version in the HIBE identity. The user encrypts their authenticator under an identity that includes a specific firmware version, and the decryption enclave only has access to the provisioned key corresponding to its current firmware version. If those versions do not match, the decryption enclave will be unable to decrypt the user's authenticator, preventing the job from running.

S4 (Forward Security). Forward security is accomplished through the key rotation and the epoch mechanism. As noted in Section IV-C, the infrastructure provider can *freeze* enclaves to prevent them from rotating keys every minor or major epoch. This gives the attacker time to launch attacks on the TEE hardware and firmware guarding the underlying cryptographic keys. If the HIBE decryption key is compromised, the infrastructure provider can decrypt any binaries encrypted for the major epoch in place when the freeze occurred. This violates goal S2 for any binaries encrypted for those minor epochs, and goals S1 and S3 for any remaining time in that major epoch. However, such a compromise of a HIBE key provides no benefit across *major* epoch boundaries. By refreshing application secrets once per *re-encryption limit*, users can force the breaking of a new machine every re-encryption limit many major epochs. The secrets obtained from the decrypted binaries at a major epoch, could be in the worst case, be the same as those of a binary from a major epoch, re-encryption limit many epochs ago.

Compromise of a TEE's current provisioning key enables the infrastructure provider to run the provisioning protocol outside the provisioning enclave, as well as access the stored HIBE private key, leading to compromise of the HIBE decryption keys for all major epochs from the initial point of compromise until

that provisioning key is revoked by the hardware manufacturer—which will occur upon detection—or there is a firmware version update. Such a compromise, while damaging, does not violate forward security, since all binaries encrypted for *prior* major epochs remain secure. Session establishment with the TPM uses the burnt-in TPM endorsement key for the session establishment with freshly generated session keys, ensuring forward security of the TPM communication and provisioning key based HIBE private key access protects against TPM de-soldering attacks [48].

S5 (Correct Infrastructure Provider). Similar to goal S3, this security stems from including the infrastructure provider's signature verification key in the HIBE identity, ensuring the binary only runs on hardware provisioned for that provider. Additionally, the hardware manufacturer verifies that key during provisioning (see Section IV-B), preventing one infrastructure provider (or a malicious third party) from impersonating another at provisioning.

B. Fully Malicious Users

A user will not intentionally leak their application secrets. The user can submit malicious code attempting to break free of the enclave and take control of the infrastructure provider's system software. However, this is subsumed by the fact that the infrastructure provider is an active adversary. The user can provide a malformed or invalid authenticator. If the hash of the binary or the symmetric decryption key is incorrect, the enclave will fail to decrypt, leading to a denial of service.

VI. Case Study: Reusable Serverless Enclaves

Bellerophon is flexible enough to support reusable enclaves [19], a state-of-the-art framework for confidential execution of serverless functions. Reusable enclaves reduce cold starts by running a "function enclave" containing a hardened interpreter in each enclave and reusing the enclave across multiple serverless function executions. The first execution requires standard enclave initialization, but each subsequent invocation resets the function enclave to its initial state before executing the user-supplied operation.This optimization reduces a 2–3 second cold start to around 25 ms for state reset, but the existing implementation still requires interactive enclave attestation with users at startup, incurring all problems of interactive attestation including WAN network latencies.

To integrate reusable enclaves with Bellerophon and eliminate the need for interactive attestation, we slightly modify the function enclave to execute the Bellerophon Decryption Enclave protocol and hash user workloads. To prepare a serverless workload, the user encrypts it as described in Section IV-A, but uses the hash of the encrypted workload as ϕ and the hash of the correct function enclave as H in the authenticator. On receiving an encrypted workload, the function enclave hashes it, and executes the Bellerophon Decryption Enclave protocol, sending the hash of the workload to the decryption enclave as ϕ. The decryption enclave decrypts the authenticator, verifies H and ϕ match, and sends back the key κ, which the function enclave can use to decrypt and run the workload.

If H in the authenticator matches the hash of the function enclave, then it must be running the correct code, and that code will correctly hash the encrypted workload and send it to the Bellerophon Decryption Enclave as ϕ. Therefore, if H and ϕ *both* match, both the function enclave and the encrypted workload must be unmodified from what the user intended, preventing improper release of κ. This chained verification can extend to multiple layers, with each hashing the previous layer and the decryption enclave ensuring the chain of hashes produces the expected value. We evaluate the performance of this approach in Section VII.

VII. Performance Evaluation

We evaluate the TCB size and the performance of Bellerophon along four criteria:

1) *Latency of verification:* how long does it take to load and verify a new enclave?
2) *Latency of load balancing:* as load balancing may be done before launch, its cost is on the critical path.
3) *Latency of minor epoch rotation: this is an overhead that reduces the ability to run enclaves.*
4) *End-to-end performance impact of Bellerophon when running serverless functions with the reusable enclaves framework.*

Implementation Notes. We implement a prototype of Bellerophon on top of Intel SGX enclaves. Our prototype prepares user binaries with a decryption stub and symmetric encryption of user code/data. At the remote machine, our prototype implements the full decryption protocol for attestation, the decryption enclave including support for minor epochs, the re-encryption enclave for major epochs, the load balancing enclave, and TPM storage of keys. We implement the provisioning protocol as a server application written in Rust and a provisioning enclave for each machine.

We use Intel SGX SDK v2.23 and the *hohibe* HIBE library [49], a Rust implementation of Boneh et. al.'s HIBE scheme with a constant size ciphertext [40]. We modified the library to run inside an SGX enclave by removing all dependencies on the standard library and replacing them with the enclave runtime components from the Apache Teaclave Project [50]. The system uses the authenticated ECDH protocol to protect communication from the decryption stub to architectural enclaves [51]. The Bellerophon enclaves access the TPM via the WolfTPM library [52]. We use the protocol described in the CPU to TPM Bus Protection Guide [48] to provide replay protection, confidentiality and integrity of messages sent to the TPM. We encrypt TEE binaries using AES-GCM with the PCL loader encryption tool that is a part of the Intel SGX SDK [44]. The tool encrypts each section after setting the write permission and inserts a table consisting of the offsets of all the encrypted sections including their respective IVs, tags and the original permissions. The decryption stub uses this table to decrypt all the encrypted sections in-place

Component	Lines of code
Encryption libraries	17,179
WolfTPM	14,424
Teaclave	1,034
Provisioning Server	1,107
Decryption enclave	1,257
Re-encryption and Load-balancing enclaves (same logic)	1,129
Provisioning enclave	1,555

TABLE 4: Implementation size

and restores the permissions using the SGX Enclave Memory Manager [45].

TCB Size. Table 4 shows the components of Bellerophon and their size. The enclave logic is dominated by HIBE encryption code. In contrast, Intel's SGX Software TCB including the verifier and the local architectural enclaves is 74,418 lines of code, not including any code for secret management or replication for high availability. Hashicorp's Vault secret manager [16], a reliable replicated service for managing user secrets, is over 450,000 lines of code, similar to what is needed to implement a verifier or secret provisioning service in the cloud.The addition of a hardware TPM into the TCB is in line with the usage of TPMs for VM attestation in prior academic work and real cloud deployments [30], [31], [32]

Testbeds. Our prototype is built on Intel SGX and we test criteria 1-3 (microbenchmarks) on an SGX enabled Intel NUC with an i7-10710U CPU running at 1.6 Ghz with 32 GB of RAM. The NUC is equipped with a TPM v2.0. Note that we are not using server CPUs. The NUC runs Linux 5.10 installed with the Intel SGX PSW and SDK v2.23.

To evaluate the end-to-end impact of Bellerophon on serverless functions, we use an Intel NUC (model NUC7PJYHN1) with an Intel Pentium Silver J5040 processor running at 2 Ghz and with 8 GB of RAM, since it has the Flexible Launch Control (DCAP) feature that is required by the reusable enclaves artifact. The i7-10710U outperforms the Pentium Silver J5040 by 1.91X on average [53]. This NUC runs Linux 6.8.0-52 with the Intel SGX PSW and SDK v2.23.

To simulate network delays between the user enclave and verifier, we use the `tc` tool to add delay to the loopback network device.

A. *Verification Latency*

We separately measure the time taken for enclave creation (load and perform measurement) and to complete the attestation process.

For Intel SGX remote attestation, we run the modified sample code from the Intel SGX SDK which implements the EPID attestation flow [54] with the verifier running as a separate process on the same machine accessed over a TCP/IP socket. Even though Intel EPID is deprecated, the primary outcome of the experiment depends on the number of network messages sent and the respective network latencies during the attestation process. The number of WAN network messages in Intel EPID is at most double that of ECDSA attestation based on Intel Trust Authority [24], resulting in the experiment estimating roughly twice the latency of ECDSA attestation in the best case. This implementation does not count the latency between the verifier and the hardware provider as they are implemented within the same process. To simulate the latency of a system with a low TCB where the users runs their own verifier outside the cloud datacenter, we use the *tc* tool to inject delays into the loopback network interface. The time to create an enclave is the running time of `sgx_create_enclave()` which includes time taken to hash the enclave content - this scales with the enclave size. Interactive remote attestation or the Bellerophon Decryption Protocol is executed after enclave creation.

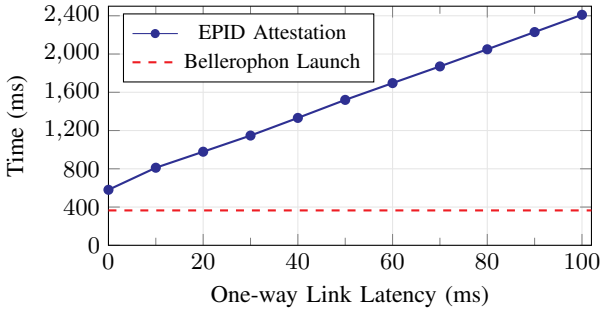

Fig. 5: Intel EPID remote attestation time increases linearly with link latency. Bellerophon's non-interactive nature means launch does not depend on the link latency. Even for 40 MB binaries, Bellerophon's entire launch process (which comprises Intel SGX enclave creation and the Bellerophon Decryption Protocol) is faster than EPID attestation (including the enclave creation time).

Figure 5 shows the total latency of completing the interactive attestation process (including the enclave creation time) for various simulated link latencies. All the data points were obtained by computing the average time over 100 runs of the remote attestation protocol. The protocol takes 580 ms (including the enclave creation time) with a latency of 0 ms, and for every 10ms increase in link latency, the protocol latency increases by approximately 160ms. This is because there are roughly 8 messages sent in each direction owing to the usage of TCP/IP. The EPID protocol running time is independent of enclave size, as it only sends fixed-size messages.

For Bellerophon, the decryption protocol replaces the interactive remote attestation protocol of SGX. Recall that Bellerophon does not require a trusted verifier for its decryption protocol, removing the TCB of the interactive verifier. To measure the latency of the Bellerophon decryption protocol, we use a single enclave and vary the number of user data pages in the TEE binary. Since the Bellerophon Decryption Enclave is using a pre-computed HIBE decryption key with [40], the decryption time of the symmetric key blob does not depend on the depth of the HIBE hierarchy. We also measure the time taken for SGX to create the enclave as a comparison. Note

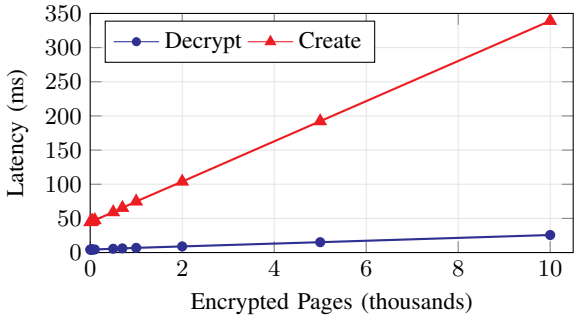

Fig. 6: Bellerophon Decryption Protocol and Intel SGX enclave creation latency. Note that the Bellerophon Decryption Protocol is executed after Intel SGX enclave creation.

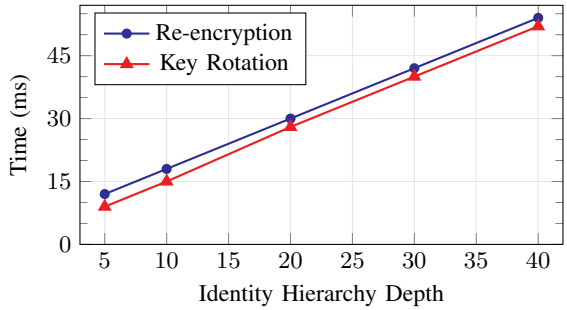

Fig. 7: Impact of identity hierarchy depth on authenticator reencryption and minor epoch key rotation.

Benchmark	Execution Time (ms)	Intel ECDSA (ms)	Bellerophon (ms)
`add`	38	132 + 8×WAN	27
`hash`	42	132 + 8×WAN	27
`prime`	10,716	132 + 8×WAN	27
`clock`	38	132 + 8×WAN	27

TABLE 8: Warm start performance of reusable serverless functions. The times for Intel ECDSA are in terms of the single-hop wide-area network latency, WAN.

that this creation time is the stock SGX enclave creation time dependent on the size of the enclave, and is present for both Bellerophon and SGX attested enclaves.

Figure 6 shows the latency of completing the Bellerophon decryption protocol along with the time taken to create the enclave. The decryption of the symmetric key blob takes a constant 4ms while the majority of the decryption time goes in the decryption of the encrypted binary using AES-GCM. An enclave with 10,000 encrypted pages (40 MB) takes 25 ms to decrypt completely. This is 216 ms faster than SGX with zero network latency, and 18× faster than with a link latency of 10 ms. Including the creation latency, a 40 MB enclaves takes only 365 ms to launch, still faster than SGX EPID excluding the creation time with a link latency of 0 ms.

B. Re-encryption and Rotation Latency

Bellerophon provides three architectural enclaves. The re-encryption enclave and load-balancing enclave decrypt an authenticator, derive a new HIBE key and then encrypt the authenticator under the new key. In both cases the data size, an authenticator, is fixed, and the performance variability comes from HIBE key derivation; a longer id sequence leads to longer key derivations. The decryption enclave performs minor epoch rotations where it derives a key for the next epoch and writes it to the TPM. For all three operations, the majority of the computation is spent in a single DeriveSKey operation. We vary the depth of the HIBE hierarchy (i.e., depth of minor epoch IDs) used for re-encryption and rotation with each identifier being a 64 byte string. The expected depth in depth in practice is not expected to exceed 30; at most 5 for the machine and infrastructure-related identifiers including the major epoch and 25 for the minor epoch. For each depth, we compute the average over 100 runs of the operation. As load balancing is almost identical to re-encryption, we do not report its performance.

Figure 7 shows mean re-encryption and minor epoch rotation computation time, which are nearly identical. For short hierarchies, re-encryption takes only 12 ms, and increases by 1–1.5 ms for each level. This time is minor compared to enclave decryption (Figure 3). Writing HIBE keys to the TPM takes on average 22 ms. Overall, the computational overhead of all three enclaves is minimal.

The provisioning protocol takes 0.97 seconds with no network latency (0ms) and 1.12 seconds with a network latency of 10ms.

C. End-to-End Performance

The reusable enclaves framework [19] runs WASM binaries inside an enclave with a WASM interpreter. We integrate Bellerophon into the execution path by encrypting the WASM binaries, generating authenticators as in Section VI and decrypting the binaries by the Bellerophon Decryption Protocol before loading into the WASM interpreter. We measure the time it takes to execute the Bellerophon Decryption Protcol and decrypt the WASM binary before execution. We use the same benchmark functions used by [19]:

- `add`: adds two input numbers
- `prime`: computes the 20,000,000th prime number
- `hash`: hashes a string of length 10, 100 times
- `clock`: measures the granularity of the interpreter's internal clock implementation

Recall from Section VI that the reusable enclaves framework requires that the function enclave be attested per invocation of the serverless function. To compare with Bellerophon, we measure the time it takes to perform ECDSA attestation (that is part of the reusable enclaves implementation) of the function enclave. The sizes of the WASM binaries are in the range of 34-36 4K pages.

Table 8 shows the mean function execution time in comparison with the time taken for remote attestation as well as for Bellerophon (standard deviation under 5ms for all measurements). For Intel ECDSA attestation, the time is

the same for each function since each uses the same function encalve. The measured time of 132ms includes no network delays, but ECDSA attestation incurs 8 wide-area network hops. By contrast, Bellerophon takes only 27ms to run in all of the functions, since the sizes of the WASM binaries are all similar. Bellerophon outperforms ECDSA attestation even with no network delays, and remains independent of the wide area latency. For long running functions, like `prime` which takes nearly 11 seconds, the performance gains of Bellerophon becomes less significant.

VIII. Discussion and Limitations

Real-time launch policies and session-based forward secrecy. Due to non-interactivity, Bellerophon does not support dynamic decision making of whether a particular TEE instance should be run or not, such as based on the time of day, as supported by Intel Trust Authority [55]. Static launch policies can be baked into the decryption stub with careful consideration of their dependency on the untrusted system software returning correct data. Dynamic changes to launch policies are not supported and a change to a policy requires the user to generate a new binary and invalidate old secrets. Further, non-interactivity does not support session-based ephemeral secrets in the TEE computation that provide forward *secrecy*.

Applications requiring these features can implement logic in the enclave code to perform interactive checks or request ephemeral secrets. One can always add interactivity to a non-interactive system.

Enforcing policy in architectural enclaves. The architectural enclaves in Bellerophon could be extended to enforce additional policies. For example, the load balancer described in Section IV-D will re-encrypt any authenticator with a new machine ID. This could be restricted by, for example, adding an *Allow Load Balance* bit in the authenticator which can be set by users and cleared by a load balancer when re-encrypting. This could prevent the output authenticator from being re-encrypted again for a new machine.

Customizing the Duration of Epochs Epoch lengths in Bellerophon can be estimated based on the performance of provisioning, minor rotations and re-encryptions.

According to our measurements, if the provisioning protocol takes 1 second, then 3600 machines can be reprovisioned in an hour on a single core, or 2.7 million/day on a 64-core machine. Likewise, if minor key rotation and re-encryption of an authenticator takes 30ms, then a single machine can rotate 30 stored authenticators/second/core. Then it takes 30 seconds for a single minor epoch rotation for a machine with 1000 binaries. Ideally major epochs would be frequent for maximum security, but they do impose a cost of rerunning the provisioning protocol. They also place a requirement on time synchronization, as clients use the current time to derive the current epoch, so epochs should not be too close in length to possible clock skew. We recommend major epochs last 1 day and minor epochs last 10 minutes as a resonable balance between the overhead, so a core can handle reprovisioning of a large cloud service, and a single machine can do re-encryption for 1000 TEEs with only 5% overhead.

IX. Related Work

Non-interactive attestation. PodArch [56] proposed an idea almost identical to that described in Section IV-A. However, it did not address scalable key management or forward security. Chancel [29] uses an attested program loader that can decrypt encrypted binaries. The user still has to maintain secrets in loader enclaves in different machines depending on where the user enclave is going to run in the future. Bellerophon eliminates the complexity associated with this secret management.

Other works have focused on non-interactive attestation in the context of IoT devices, but there is no notion of a user submitting a binary to an IoT device for outsourcing computation. zRA [26] achieves non-interactivity of attestation in IoT devices where the measurement of the trusted firmware is assumed to be known only to the trusted hardware provider. The device publishes a zero knowledge proof of its current firmware state on a permissionless blockchain, which is then verified against the commitment of the measurement at a later time. SCRAPS [27] achieves scalability of the attestation verifier by doing verification in a smart contract on the blockchain for Pub-Sub Iot networks. Similar work on scalable attestation [57], [58], [59], [60], [61], [62], [63], [64], [65], [66], [67], [68], [69] for Pub-Sub IoT networks provide a subset of the systems and security properties that are provided by SCRAPS. However, a trusted verifier is still required to verify the evidence of attestation generated by the IoT devices, making it unsuitable for achieving the goals that Bellerophon targets (goal F1).

X. Conclusion

Remote attestation is critical for the security of Trusted Execution Environments. However, existing remote attestation schemes rely on an interactive protocol with a verifier that either the user must manage, requiring long-latency communication paths, or is managed by the cloud provider, which greatly increase the trust required of the provider. Bellerophon provides non-interactive attestation by encrypting binaries in a key known only to secure hardware on a trusted machine. It removes all communication during enclave launch as well as trust in a cloud provided verifier. Our evaluation shows that Bellerophon provides comparable security to existing interactive attestation mechanisms at much lower latency.

References

[1] Microsoft, "Announcing: Microsoft moves $25 Billion in credit card transactions to Azure confidential computing," https://techcommunity.microsoft.com/blog/azureconfidentialcomputingblog/announcing-microsoft-moves-25-billion-in-credit-card-transactions-to-azure-confi/3981180, 2023.

[2] Fortanix, "Preventing Money Laundering in a Confidential Computing Way," https://www.fortanix.com/blog/preventing-money-laundering-in-a-confidential-computing-way, 2023.

[3] Intel, "Intel SGX Helps UCSF Propel Medical Device Innovations," https://www.intel.com/content/www/us/en/newsroom/news/ucsf-propel-medical-device-innovations.html#gs.i6buxf, 2020.

[4] Intel, "Maximum Security at the Processor Level: Intel SGX Protects Electronic Patient Record," https://www.intel.com/content/www/us/en/content-details/826053/maximum-security-at-the-processor-level-intel-sgx-protects-electronic-patient-record.html, 2020.

[5] intc.com, "Intel and Penn Medicine Announce Results of Largest Medical Federated Learning Study," https://www.intc.com/news-events/press-releases/detail/1593/intel-and-penn-medicine-announce-results-of-largest-medical, 2020.

[6] S. Network, "Secret network overview - private smart con- tracts on the blockchain." https://scrt.network/about/about-secret-network/, 2022.

[7] P. Network, "Phala Network Overview," https://docs.phala.network/overview/phala-network, 2022.

[8] Crust, "Crust Overview," https://wiki.crust.network/docs/en/crustOverview, 2022.

[9] Microsoft, "Azure Confidential Computing," https://azure.microsoft.com/en-us/solutions/confidential-compute, 2020.

[10] Intel, "Intel SGX," https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html, 2015.

[11] Intel, "Intel TDX White Paper," https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html, 2021.

[12] AMD, "AMD SEV," https://www.amd.com/en/developer/sev.html, 2021.

[13] Intel, "Intel SGX Remote Attestation," https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html, 2018, accessed April 2025.

[14] Intel, "Intel TDX white paper," https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf, 2022, accessed April 2025.

[15] AMD, "AMD SEV-SNP Remote Attestation," https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf, 2022.

[16] HashiCorp, "hashicorp/vault: A tool for secrets management, encryption as a service, and privileged access management," https://github.com/hashicorp/vault, 2015, accessed April 2025.

[17] W. Qiang, Z. Dong, and H. Jin, "Se-Lambda: Securing privacy-sensitive serverless applications using SGX enclave," in *SecureComm*, 2018.

[18] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza, "AccTEE: A WebAssembly-based Two-way Sandbox for Trusted Resource Accounting," in *Middleware*, 2019.

[19] S. Zhao, P. Xu, G. Chen, M. Zhang, Y. Zhang, and Z. Lin, "Reusable enclaves for confidential serverless computing," in *USENIX Security*, 2023.

[20] R. G. Brown, "Conclave Cloud Whitepaper," https://uploads-ssl.webflow.com/62e0881a72ba0c74c831c6f8/631a090677613438d726bd70_Conclave_Introductory_Whitepaper.pdf, Dec. 2021, accessed April 2025.

[21] AWS, "AWS Lambda Cold Start," https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-1/, 2022.

[22] Microsoft, "Microsoft Azure Attestation," https://learn.microsoft.com/en-us/azure/attestation/overview, 2024.

[23] AWS, "AWS Attestation," https://docs.aws.amazon.com/enclaves/latest/user/set-up-attestation.html, 2023.

[24] Intel, "Intel Trust Authority," https://docs.trustauthority.intel.com/main/articles/introduction.html, 2023.

[25] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner, "S-FaaS: Trustworthy and accountable function-as-a-service using Intel SGX," in *CCSW*, 2019.

[26] S. Ebrahimi and P. Hassanizadeh, "From interaction to independence: zkSNARKs for transparent and non-interactive remote attestation," in *NDSS*, 2024.

[27] L. Petzi, A. E. B. Yahya, A. Dmitrienko, G. Tsudik, T. Prantl, and S. Kounev, "SCRAPS: Scalable collective remote attestation for Pub-Sub IoT networks with untrusted proxy verifier," in *USENIX Security*, 2022.

[28] J. Neureither, A. Dmitrienko, D. Koisser, F. Brasser, and A.-R. Sadeghi, "LegIoT: Ledgered trust management platform for IoT," in *ESORICS*, 2020.

[29] A. Ahmad, J. Kim, J. Seo, I. Shin, P. Fonseca, and B. Lee, "CHANCEL: Efficient multi-client isolation under adversarial programs," in *NDSS*, 2021.

[30] V. Narayanan, C. Carvalho, A. Ruocco, G. Almasi, J. Bottomley, M. Ye, T. Feldman-Fitzthum, D. Buono, H. Franke, and A. Burtsev, "Remote attestation of confidential VMs using ephemeral vTPMs," in *Proceedings of the 39th Annual Computer Security Applications Conference*, ser. ACSAC '23. Association for Computing Machinery, 2023, p. 732743.

[31] Microsoft, "TPM attestation in Azure," https://learn.microsoft.com/en-us/azure/attestation/tpm-attestation-concepts.

[32] Trusted Computing Group, "TPM as an API for attestation in big, distributed environments," https://trustedcomputinggroup.org/tpm-as-an-api-for-attestation-in-big-distributed-environments/.

[33] J. Li and E. Brickell, "Enhanced privacy ID from bilinear pairing for hardware authentication and attestation," in *Social Computing / IEEE International Conference on Privacy, Security, Risk and Trust, 2010*, 2010.

[34] Intel, "Intel SGX Data Center Attestation Primitives (Intel SGX DCAP)," https://download.01.org/intel-sgx/latest/dcap-latest/linux/docs/DCAP_ECDSA_Orientation.pdf, 2022.

[35] ——, "SGX Local Attestation Flow," https://github.com/intel/linux-sgx/tree/main/SampleCode/LocalAttestation, 2025, accessed July 2025.

[36] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, 2016.

[37] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song, "Keystone: An Open Framework for Architecting Trusted Execution Environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys, 2020.

[38] R. Canetti, S. Halevi, and J. Katz, "A Forward-Secure Public-Key Encryption Scheme," *J. Cryptol.*, 2007.

[39] Canetti, Ran and Halevi, Shai and Katz, Jonathan, "Chosen-ciphertext security from identity-based encryption," in *EUROCRYPT*, 2004.

[40] D. Boneh, X. Boyen, and E.-J. Goh, "Hierarchical identity based encryption with constant size ciphertext," in *EUROCRYPT*, 2005.

[41] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *ASIACRYPT 2000*, 2000.

[42] Intel, "Intel EPID EOL Notice," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/sgx-ias-using-epid-eol-timeline.html, 2023.

[43] V. Scarlata, S. Johnson, J. Beaney, and P. mijewski, "Supporting third party attestation for intel® sgx with intel® data center attestation primitives," 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:221506554

[44] Intel, "Intel SGX protected code loader," https://github.com/intel/linux-sgx-pcl.

[45] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel software guard extensions (intel sgx) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, ser. HASP '16, 2016. [Online]. Available: https://doi.org/10.1145/2948618.2954331

[46] "Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms," https://datatracker.ietf.org/doc/html/rfc5802, 2010, accessed July 2025.

[47] R. Anderson, "Two remarks on public key cryptology," http://www.cl.cam.ac.uk/ftp/users/rja14/forwardsecure.pdf, 1997.

[48] Trusted Computing Group, "CPU to TPM bus protection guidance – active attack mitigations," https://trustedcomputinggroup.org/wp-content/uploads/TCG_-CPU_-TPM_Bus_Protection_Guidance_Active_Attack_Mitigations-V1-R30_PUB-1.pdf, 2023.

[49] D. Schadt, "hohibe.rs - Hierarchical Identity Based Encryption," https://crates.io/crates/hohibe, 2021, accessed April 2025.

[50] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin, "Towards Memory Safe Enclave Programming with Rust-SGX," in *CCS*, 2019.

[51] Intel, "Innovative Technology for CPU Based Attestation and Sealing," https://www.intel.com/content/www/us/en/developer/articles/technical/innovative-technology-for-cpu-based-attestation-and-sealing.html, 2013.

[52] wolftpm, "wolfTPM: Portable TPM 2.0 project designed for embedded use," https://github.com/wolfSSL/wolfTPM.

[53] Technical City, "Pentium Silver J5040 vs i7-10710U - Technical City," https://technical.city/en/cpu/Core-i7-10710U-vs-Pentium-Silver-J5040, 2019, accessed April 2025.

[54] Intel, "Code Sample: Intel Software Guard Extensions Remote Attestation End-to-End Example," https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/get-started.html, 2023.

[55] ——, "Intel Trust Authority," https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/linux-overview.html, 2023.

[56] Shweta Shinde and Shruti Tople and Deepak Kathayat and Prateek Saxena, "PodArch: Protecting Legacy Applications with a Purely Hardware TCB," School of Computing, National University of Singapore, Tech. Rep. NUS-SL-TR-15-01, February 2015.

[57] N. Asokan, F. Brasser, A. Ibrahim, A.-R. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann, "SEDA: Scalable Embedded Device Attestation," in *CCS*, 2015.
[58] X. Carpent, K. ElDefrawy, N. Rattanavipanon, and G. Tsudik, "Lightweight Swarm Attestation: A Tale of Two LISA-s," in *CCS*, 2017.
[59] A. Ibrahim, A.-R. Sadeghi, and S. Zeitouni, "SeED: secure non-interactive attestation for embedded devices," in *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '17, 2017.
[60] A. Ibrahim, A.-R. Sadeghi, G. Tsudik, and S. Zeitouni, "DARPA: Device Attestation Resilient to Physical Attacks," in *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2016.
[61] F. Kohnhauser, N. Buscher, S. Gabmeyer, and S. Katzenbeisser, "SCAPI: A Scalable Attestation Protocol to Detect Software and Physical attacks," in *WiSec*, 2017.
[62] E. Dushku, M. M. Rabbani, M. Conti, L. V. Mancini, and S. Ranise, "SARA: Secure Asynchronous Remote Attestation for IoT Systems," *TIFS*, 2020.
[63] F. Stumpf, A. Fuchs, S. Katzenbeisser, and C. Eckert, "Improving the Scalability of Platform Attestation," in *STC*, 2008.
[64] F. Kohnhäuser, N. Büscher, and S. Katzenbeisser, "SALAD: Secure and Lightweight Attestation of Highly Dynamic and Disruptive Networks," in *ASIACCS*, 2018.
[65] M. M. Rabbani, J. Vliegen, J. Winderickx, M. Conti, and N. Mentens, "SHeLA: Scalable Heterogeneous Layered Attestation," *IEEE Internet of Things Journal*, 2019.
[66] M. Bampatsikos, C. Ntantogian, C. Xenakis, and S. C. A. Thomopoulos, "BARRETT BlockchAin Regulated REmote aTTestation," in *WI Companion*, 2019.
[67] M. Ambrosin, M. Conti, A. Ibrahim, G. Neven, A.-R. Sadeghi, and M. Schunter, "SANA: Secure and Scalable Aggregate Network Attestation," in *CCS*, 2016.
[68] B. Kuang, A. Fu, S. Yu, G. Yang, M. Su, and Y. Zhang, "ESDRA: An efficient and secure distributed remote attestation scheme for IoT swarms," *IEEE Internet of Things Journal*, 2019.
[69] F. Kohnhauser, N. Buscher, S. Gabmeyer, and S. Katzenbeisser, "A practical attestation protocol for autonomous embedded systems," in *EuroS&P*, 2019.
[70] AMD, "AMD SEV-SNP Attestation," https://www.amd.com/content/dam/amd/en/documents/developer/lss-snp-attestation.pdf, 2022.
[71] C. Fruhwirth, "LUKS1 On-Disk Format Specification Version 1.2.3," https://gitlab.com/cryptsetup/cryptsetup/-/wikis/LUKS-standard/on-disk-format.pdf, 2018, accessed July 2025.
[72] AMD, "SEV Secure Nested Paging Firmware ABI Specification," https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56860.pdf, 2025, accessed July 2025.
[73] Chris Lott, "Which Intel Platforms Can Be Used to Play Ultra HD Blue-Ray Discs?" https://www.intel.com/content/www/us/en/support/articles/000089271/intel-nuc.html, 2022.
[74] Intel, "Which Intel Platforms Can Be Used to Play Ultra HD Blue-Ray Discs?" https://hackaday.com/2022/01/18/sgx-deprecation-prevents-pc-playback-of-4k-blu-ray-discs/, 2022.
[75] Google, "Confidential VM Attestation," https://learn.microsoft.com/en-us/azure/attestation/overview, 2024.
[76] AWS, "AWS Nitro Enclaves," https://docs.aws.amazon.com/enclaves/latest/user/nitro-enclave.html, 2020, accessed April 2025.
[77] Microsoft, "TPM Key Attestation - Microsoft," https://learn.microsoft.com/en-us/windows-server/identity/ad-ds/manage/component-updates/tpm-key-attestation, May 2023, accessed April 2025.
[78] J. Ménétrey, C. Göttel, A. Khurshid, M. Pasin, P. Felber, V. Schiavoni, and S. Raza, "Attestation Mechanisms for Trusted Execution Environments Demystified," in *DAIS*, 2022.
[79] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "SecTEE: A software-based approach to secure enclave architecture using TEE," in *CCS*, 2019.
[80] W. Li, H. Li, H. Chen, and Y. Xia, "AdAttester: Secure online mobile advertisement attestation using TrustZone," in *MobiSys*, 2015.
[81] Z. Ling, H. Yan, X. Shao, J. Luo, Y. Xu, B. Pearson, and X. Fu, "Secure boot, trusted boot and remote attestation for ARM TrustZone-based IoT nodes," *Journal of Systems Architecture*, 2021.
[82] C. Shepherd, K. Markantonakis, and G.-A. Jaloyan, "LIRA-V: Lightweight Remote Attestation for Constrained RISC-V Devices," in *SPW*, 2021.
[83] G. Chen, Y. Zhang, and T.-H. Lai, "OPERA: Open remote attestation for Intel's secure enclaves," in *CCS*, 2019.

APPENDIX

A. Session Establishment Between Enclaves

We describe the Intel SGX ECDH based secure channel establishment protocol [35] between two enclaves A (decryption stub) and B (decryption enclave) over a channel controlled by the untrusted OS for completeness. The key idea here is that the Diffie-Hellman key exchange messages are authenticated using the attestation reports that can only be generated in the intended enclaves with their corresponding measurements, preventing man-in-the-middle attacks. Also, note that the hash identity of the decryption enclave is hardcoded in the decryption stub. The exact details of the report generation and verification varies between different TEE implementation:

1) **Session Initialization**: Enclave A sends a session initialization message to enclave B
2) **Message 1 (B to A)**: B generates an ECDSA key pair (g^B_{pub}, g^B_{priv}) and requests its own attestation report $r_B([])$ without any additional data (indicated by []). It then sends g^B_{pub} and $r_B([])$ to A.
3) **Message 2 (A to B)**: On receiving message 1 from enclave B, A verifies the identity of B using the hash in $r_B([])$ and the hardcoded identity of B. It then generates an ECDSA key pair (g^A_{pub}, g^A_{priv}). It then computes the shared secret ($g^{A,B}$)) from g^A_{priv} and g^B_{pub}. A then generates an attestation report with $SHA256(g^B_{pub}||g^A_{pub})$ as the additional data ($r_A([SHA256(g^B_{pub}||g^A_{pub})])$. Finally, A generates message 2 as a struct consisting of g^A_{pub}, $r_A([SHA256(g^B_{pub}||g^A_{pub})])$ and a CMAC computed over the message using the shared key $g^{A,B}$). It then sends message 2 to B.
4) **Message 3 (B to A)**: On receiving message 2 from A, B verifies the report and verifies the report data by computing $SHA256(g^B_{pub}||g^A_{pub})$. It then computes the shared secret ($g^{A,B}$)) from g^B_{priv} and g^A_{pub}. The CMAC on the message is verified using the shared secret $g^{A,B}$). B then computes message 3 by computing the report $r_B([SHA256(g^A_{pub}||g^B_{pub})])$ along with its CMAC using the shared secret $g^{A,B}$). Message 3 is then sent to A
5) **Process Message 3 (A)**: On receipt of message 3, A verifies the report, its sender hash, the additional data and its CMAC using the shared secret $g^{A,B}$). Using the shared secret $g^{A,B}$) both A and B can compute a shared session key which can be used for encrypted and replay-protected communication. One such way (as in TLS) is to use AES-GCM to encrypt each message and have the sequence number of each message as part of the message header, which is passed in as additional data to generate a tag.

Note that message 2 and 3 cannot be This protocol can be adapted to work with Intel TDX and AMD SEV-SNP. All that is

required is a way to generate attestation reports with additional data that can be verified at each of the communicating enclaves.

In Intel TDX, the platform also supports SGX enclaves. The decryption enclave is an SGX enclave, and the user enclave is a TD VM. Attestation reports for TD VMs can be requested from the TDX module running in SEAM mode. The TDX module issues SEAMREPORT instruction to get a hardware generated attestation report, which can be verified in an SGX enclave using the EVERIFYREPORT2 instruction. However, there is no instruction to verify the report generated by an SGX enclave inside a TD VM. Therefore, the SGX enclave must generate a quote using the quoting enclave and then send it to the TD for verification using Intel ECDSA attestation [14]. This does not require external communication, as the certificates needed for verification can be cached by the cloud provider.

In AMD SEV-SNP, the attestation report is requested from the AMD Platform Security Processor (PSP) firmware. The AMD SEV-SNP VM sends and receives commands from the firmware using a confidentiality, integrity, and replay protected channel using a shared secret that is shared using a private page mapping with the VM. The decryption enclave, as well as the user enclav,e is an SEV-SNP VM. They both verify each other's attestation reports using AMD's certificate-based attestation [70], and this does not require external communication, as the certificates needed for verification can be cached by the cloud provider.

B. Bellerophon on Intel TDX

Porting Bellerophon to another platform requires support for (i) enclaves with session establishment capabilities described above (ii) a provisioning enclave with access to a provisioning key whose knowledge proves the firmware version, CPU manufacturer, and individual machine ID of the CPU, (iii) to support Bellerophon architectural enclaves, a hardware key derivation mechanism that implicitly takes the enclave author's public key as input and an enclave launch mechanism that verifies the signature on the enclave's metadata block (that also includes its expected measurement), (iv) enclave binaries that can be encrypted and execute the Bellerophon decryption protocol.

Since all Intel TDX platforms also support Intel SGX, all steps for running Bellerophon are the same except for the session establishment (discussed in the previous subsection) and the binary preparation. The VM image consists of an OS image along with a filesystem that contains the userspace applications. The entire filesystem containing the user secrets can be encrypted with a symmetric key, such as using LUKS [71] with the decryption stub being a userspace process which executes the Bellerophon decryption protocol to obtain the key needed to decrypt and mount the LUKS filesystem.

C. Bellerophon on AMD SEV-SNP

AMD SEV-SNP provides support for VM based enclaves with session establishment capabilities. The provisioning enclave in this case is a VM with access to the provisioning key derived based on the public key of the hardware manufacturer, which was used to sign the identity block of the VM [72]. All the architectural enclaves in Bellerophon are VMs in SEV-SNP since each VM has support for a key derivation mechanism (provided by the AMD Platform Security Processor (PSP) firmware) that implicitly takes the enclave author's public key as input and an enclave launch mechanism that verifies the signature on the enclave's metadata block (that also includes its expected measurement). Similar to Intel TDX, an encrypted filesystem containing user secrets and applications can be part of the initial image which can be decrypted and mounted follwoing the execution of the Bellerophon decryption protocol.

D. Other Use-Cases of TEEs

Intel SGX has been used to enforce DRM for copyrighted media such as Blu-Ray discs by decrypting media inside an SGX enclave [73]. Bellerophon does not support this use case since since the DRM protected media would need to be encrypted for each individual Blu-Ray player; without group decryption keys, each player has a unique public key. Even with the same decryption key provisioned into the Blu-Ray players, it is impossible to enforce a particular firmware version without interactivity. However, Intel has phased out SGX for consumer CPUs (including PCs and Blu-Ray Players) and SGX is currently only available on server-class Xeon processors [74].

E. Remote attestation in cloud environments.

Remote attestation in cloud environments is performed based on infrastructure built on top of the attestation schemes supported by the hardware manufacturers of the CPUs running in the cloud datacenter. At the time of writing, Microsoft provides a unified attestation service for all the TEE-based services [22] that it provides, namely AMD SEV-SNP VMs and containers, Intel SGX enclaves, Intel TDX VMs and TPM based systems. The service follows an interactive protocol with a user-facing secret manager that is in charge of verifying the generated attestation result followed by provisioning the secrets into the TEE. Although the attestation service runs inside a TEE, the source code is available only to government customers. Google Cloud provides similar services [75]. AWS provides a custom hypervisor-based TEE solution called Nitro Enclaves [76]. The root of trust is the hypervisor which provides measurements similar to the registers in a TPM, which are then endorsed using a signing key along with a certificate chain provided by AWS [23]. All of these solutions require interactivity and a large trusted verifier and hence do not satisfy all the functionality goals of Bellerophon.

F. Remote attestation in commodity hardware.

Intel SGX supports attestation using the EPID protocol, which preserves the privacy of the CPU being attested, as well as attestation based on ECDSA certificates. With ECDSA certificates, each CPU has a unique asymmetric key pair called the Provisioning Certification Key (PCK) that is derivable by the Provisioning Certification Enclave (PCE), an architectural enclave (i.e. signed by Intel). Intel publishes the certificate for

each unique PCK public key and the PCK private key is in turn used to sign a fresh attestation key generated by the Quoting Enclave, thereby completing a certificate chain. The certificate chain can be cached by the infrastructure provider with the user verifying the chain. AMD SEV [12], Intel TDX [11] and TPM based systems [77], [75] have a similar scheme for attestation where a certificate chain is established from the hardware manufacturer to a device specific attestation key. ARM has not published a remote attestation protocol for its TrustZone based architectures [78], however previous works [79], [80], [81] have proposed several interactive remote attestation schemes for the same. Similar to ARM, the RISC-V specification does not include a remote attestation scheme. LIRA-V [82] proposes an interactive remote attestation protocol for low-powered RISC-V devices and does not satisfy all the functionality goals of Bellerophon.

G. Delegated Attestation

A line of work has explored replacing the hardware manufacturer in the interactive attestation flow with a trusted set of enclaves. OPERA [83] replaces the Intel Attestation and Provisioning services by running the same protocols as Intel in a set of trusted SGX enclaves. Trust in these delegated enclaves is established by periodically running the standard EPID attestation protocol against the Intel Attestation Service. The servers are provisioned and the user enclaves are attested using these delegated enclaves, thereby preventing Intel from learning about the identity of the enclave being attested as well as the machine on which the enclave is running. This work is orthogonal to Bellerophon in that it does not solve the problem of verifier scalability and interactivity.

H. Key Compromises

The provisioning key and the provisioned HIBE decryption key are two recoverable keys critical for the security of Bellerophon. Recall that the provisioning key is used in the provisioning protocol to prove the current firmware version of the hardware/software TCB on the machine and is the shared secret between a genuine CPU and the hardware provider. A compromise of the provisioning key enables the adversary to run the provisioning protocol outside the provisioning enclave and gain access to the decryption key for the current firmware version and major epoch. With a decryption key at minor epoch 0, all the binaries encrypted for the current major epoch for the particular machine can be decrypted. With a load balancer, any binary encrypted for the load balancer could be re-encrypted for the compromised machine and then decrypted.

A compromise of a decryption key at a specific minor epoch renders all the binaries encrypted for the current and future minor epochs up to the next major epoch vulnerable. Note that compromising the decryption key in a major epoch does not necessarily mean that the decryption key of the next major epoch is compromised since the key for the next major epoch cannot be derived from a decryption key of the current major epoch.

A compromise of a machine's hardware/software TCB, required to compromise the provisioning key or the provisioned decryption key may persist until it is disclosed to the manufacturer and patched. Similar to Intel SGX and other commercial TEEs, it is difficult for the user to ascertain the exact time (major epoch) when the attack vector was discovered, and hence the user must assume that all the binaries encrypted with a lower firmware version have been compromised.

Compromise of burnt-in provisioning key within a CPU is unrecoverable for that CPU, but individual CPUs can be revoked by the hardware provider. The hardware provider can keep track of the machines that are unrecoverable using the hardware identifier in the machine identity, and refuse to provision those machines with new private keys during provisioning.

A compromise of any aspect of the infrastructure provider's identity is beyond the scope of this work.

I. Ownership changes or stolen machines.

When a machine is sold, the new owner has to re-provision keys using their public key fingerprint. If a machine is stolen, the adversary can run stored binaries encrypted for the infrastructure provider. However, the hardware manufacturer will not re-provision it at the next major epoch, limiting which TEE packages it can run. Moreover, the machine loses power at least once as it is stolen, the decryption enclave could be augmented to require a signature check similar to that in the provisioning protocol, at start-up to verify its current operator.