Attestation-based Authorization for Stronger Security in the Cloud

Yan Zhai[1] *Qiang Cao*[2] *Jeff Chase*[2] *Michael Swift*[1]
[1]University of Wisconsin Madison [2]Duke University
type: research

Abstract

Cloud platforms provide authorization systems that govern how tenants and their applications interact with one another and share data on the cloud. We consider how a cloud platform can enable richer access control when requests originate from within the cloud, e.g., from a running software instance controlled by another tenant. It is increasingly useful for these policy checks to consider information about the requesting program, including the software that it runs and its configuration, in order to create a stronger foundation for secure sharing of data in future clouds.

This paper describes Latte, a cloud attestation system that provides a richer basis for authorization. It can authorize operations based on requester's code identity, which includes source code, build environment and runtime configuration, as well as third-party endorsements of trustworthiness. Latte supports the layered environments common in cloud computing, such as Docker containers running within virtual machines, and distributed services such as the Spark data-analytics platform.

We integrated Latte with OpenStack, Docker and Spark to demonstrate how Latte can be used to improve security and enable new usage scenarios, such as allowing untrusted parties to compute over private data. Adopting Latte requires few changes to application platforms. The overhead of Latte in most cases is negligible.

1 Introduction

The security of any application service depends on the security properties of its code. Our work seeks to incorporate software identity, such as which programs are requesting access, as first-class entities in the protection system of modern cloud environments. This provides much tighter control over how data can is used. Software programs run as *instances*, e.g., a hosted VM or container launched from an image. Instances interact through a network controlled by the cloud provider, and may act as clients or servers for other instances or external entities.

Our premise is that a cloud platform can act as a trusted broker to authenticate instances and expose metadata about them for access control and policy compliance, including attesting to software identity and configuration. For example, a user might grant read access to secret data to an instance of a program if the user's policy says that the program is trusted to be free of data leaks.

Modern cloud platforms have useful metadata about instances that provide visibility into its code identity and configuration. For example, code run directly on **cloud infrastructure** must be launched and configured through cloud platform APIs, which ensures the cloud provider knows the disk image used to boot a service and its network configuration.

Three intersecting technology trends suggest that the time is right to explore how to expose such metadata and reason about it in a comprehensive way. First, cloud environments have **mature facilities for sharing access to sensitive data**. They are audited for many compliance requirements for storage of sensitive data [51, 46]. In addition, they provide many features to support sharing across tenants, such as shared storage and authentication services. As a result, tenants can share sensitive data within a cloud.

Second, cloud-hosted applications are increasingly deployed as **immutable instances** launched from declarative specifications [9]; the configuration is frozen at launch and any management changes are applied by restarting the instance with a new configuration. These and other DevOps practices ensure that each instance has a known identity that is sealed at launch and not compromised by management operations after launch.

Third, with the increasing criticality of cyberinfrastructure, there is a rich and growing ecosystem of **verification tools** to check or endorse the trustworthiness of code. Verification systems such as IronClad [30], IronFleet [29], Verdi [62], Sel4 [35], and VCC [20] are increasingly practical. Related tools like IBM Vulnerability Advisor [32] and Clair [47] and verified software such as s2n [52] are being deployed.

Building on these trends, we present a framework called *Latte* that realizes this vision of pervasive attestation and flexible authorization in a cloud environment. It identifies the code and configuration of a running instance, and provides authorization mechanisms to reason about its trustworthiness. For example, a data owner can specify an access control policy that allows access from instances running qualified software stacks with proper configuration (e.g., locked down).

Latte defines a basic vocabulary and set of tools for services to issue authenticated assertions (*attestation statements*) about various objects—e.g., hosted instances, program objects (identified by hash), and configuration templates—together with logical policy rules to derive checkable security predicates from chains of related assertions. Cloud systems can use these tools to expose metadata that is useful for trust and visibility. These platforms can attest to properties of instances (i.e., about running code), while software build and verification tools can endorse program security properties (i.e.,

about source code repositories or binary images).

With Latte, we show how cloud applications and services can define and invoke *logical guards* that validate assertions about instances requesting access, and reason about them to infer high-level security predicates needed for compliance with a logical policy. Similarly, a client can examine a service's attestation statements and endorsements and decide whether to trust the service by validating that it was launched by a trusted platform and its code was endorsed by a trusted entity. These features enable *attestation-based authorization*, in which a service's access policy can consider attestations of code identity—together with endorsements of that code—for the instances requesting access.

Latte's use of logic allows us to address critical obstacles to the practical use of attestation in cloud systems. First, modern cloud services are often built atop multiple layers of virtualization, e.g., a Spark JVM process in a Docker container in a virtual machine. Latte's logical structure naturally supports *chain attestations* to validate a full stack of software. Second, scalable cloud-hosted services are distributed systems with multiple instances working together. Latte's logic enables a *grouping* mechanism that allows authorizers to verify that all members of a distributed service meet trust requirements. Finally, trust is usually granted to source code, which can be inspected for correctness, as opposed to binary images. Latte enables transitive trust in an image from a *trusted build service* that builds a source repository into a binary image, and issues an authenticated endorsement binding the image back to its source.

We note that attestation does not improve security in the presence of unknown bugs. However, like a firewall it provides basis for avoiding known problems, such as ensuring software is patched and locked down.

Latte introduces source-based attestation into cloud environments, and provides layered attestation and groups to check policies on a full software stack, including distributed components. Its metadata service is a centralized repository for instance metadata, enabling a wide variety of policies. These policies are expressed in a flexible logic-based policy language that can detail which programs, not just which users, should be trusted with access to data.

Our software prototype for Latte is based on a small set of extensions to OpenStack, an open-source cloud infrastructure-as-a-service platform (IaaS). Latte adds a metadata service to store statements and uses the trusted network environment of IaaS clouds for remote authentication. We implement example platforms on Latte to illustrate rich trust scenarios: platform services based on containers (Docker) and JVMs (Spark). Experiments show that Latte adds only minimal overhead.

2 Overview

As a motivating application scenario, consider the problem of *joint data mining* or *cooperative analytics*. Suppose that two tenants each have a private dataset and wants to cooperate by running analytics program P over both datasets (A and B) together. They trust that P produces output that does not expose confidential details of A or B. When running on the same cloud platform, they want to leverage cloud infrastructure and common platform frameworks—an open analytics stack such as Spark in Docker containers—to deploy P and grant it access to A and B.

How can the data owners ensure that their datasets are accessible by a requester only if it runs the correct program P? We propose that each party installs an access policy for its dataset that permits data access only from instances running P—an example of *attestation-based access control*. The cloud storage service that stores A and B examines metadata for the requesting instances to verify that each requester complies the data owner's policy: e.g., grant access to requests originating from a trusted program P running in a secured environment.

Latte defines a software infrastructure and tools to realize such scenarios for secure, flexible data sharing. Latte extends an IaaS platform with a secure *metadata service* that stores authenticated *statements* about instances. These statements are metadata assertions in a declarative logic language. They are issued (spoken) and published to the metadata store by the IaaS service itself or by another service (e.g., a hosted PaaS) layered above it. Latte defines client libraries: an *attestation library* that issues statements from predefined templates, and a *guard library* to check retrieved metadata for compliance with a policy, which is specified as a set of logical rules.

Figure 1 depicts the phases and principal roles for a typical program running as an instance in a Latte-enabled cloud. We refer to the issuers of statements about instances as *attesters* and to the compliance checker as an *authorizer*. The attesters are platforms that launch instances: the IaaS provider itself or tenant-managed third-party cloud services layered above the IaaS platform (§4.2). Various parties may act as authorizers: services of the IaaS provider, tenant-managed services hosted as tenant instances, or external entities that trust metadata fetched from the IaaS provider over a secure connection. An authorizer may conduct access checks at request time or may store metadata for later auditing.

In the joint data mining example above, the cloud storage service acts as the authorizer to check that the origin (*requester*) of each data access request complies with the data owner's specified policy. This example scenario requires extensions to the cloud platforms that launch the instances (to issue statements about them as attesters) and also to the cloud storage service (to check those

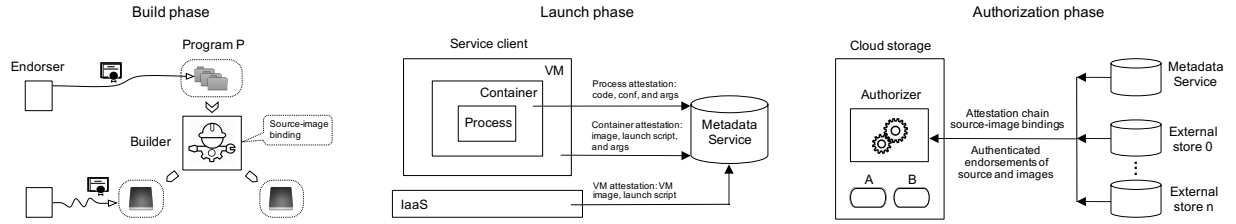


Figure 1: Principal roles in three phases of Latte operation for a typical program running as an attested instance in the cloud. In the build phase, a build service (*builder*) prepares a program image and certifies the build from an identified source repository. Other parties (*endorsers*) may issue endorsements of the source code (top) or built image (bottom). In the launch phase, a Latte-enabled hosting cloud platform (an *attester*) launches an instance from the image and certifies its image identity and configuration. In the authorization phase, an *authorizer* introspects on the instance metadata (certifications and endorsements) to check compliance with a policy. For example, if the instance requests access from a Latte-enabled service (it acts as a *requester*), the service may perform an attestation-based access control check (acts as an *authorizer*).

statements as an authorizer). There are no changes to the data analysis programs.

How does a policy in an ACL infer trust in a program P? These *guard* policies may consider logical statements that *endorse* a program object P, identified by a secure hash of its source code or binary image. The issuer of an endorsement is an *endorser*. An endorser asserts a property of P as a named predicate with optional values (§3.1); for example, the endorser might be a program analyzer (e.g., Clair [47]) or software assurance platform (e.g, SWAMP [5]) hosted by a third party. A guard policy specifies the endorsers it trusts to assert a given named program property. The Latte guard library can also import external endorsements authenticated by signature.

If an endorser is an instance within the cloud, a Latte authorizer may inspect its instance metadata in considering whether to trust its endorsement. For example, our prototype includes a trustworthy build service (§4.4) that builds a source repository into a VM or Docker container image. It runs within a VM instance and endorses the image to indicate how it was built and from what source. An authorizer may validate that the endorsement is issued from a trusted build service (a *builder*) by inspecting the builder's metadata. Trusted builds enhance the power of attestation by allowing endorsements of software properties at the source code level, e.g., by human inspectors or source analysis tools.

2.1 Trust Assumptions

Latte makes three key trust assumptions:

Trusted IaaS. Latte takes the security and isolation properties of the cloud IaaS as a given. While this is currently assumed by many (even most) cloud tenants, techniques from previous work can ground the trust chain in hardware roots of trust, and/or instantiate security-critical code modules with a hardware-attested minimal trusted computing base (TCB), as in Flicker [40], Haven [14], and SCONE [12].

Secure internal network. The IaaS platform controls the internal network that interconnects its tenants, so that network addresses cannot be spoofed or forged [10, 65]. Latte uses instance IaaS addresses as authenticated principal identifiers. This choice distinguishes Latte from many previous systems that use public key infrastructure to authenticate attested instances. Relative to these approaches, Latte is compatible with applications that do not use cryptography, reduces communication overhead, and does not require secure key management, which itself is a challenging problem [18]. We assume without loss of generality an unconstrained flat public IP address is used for space for all instances, e.g., IPv6 [11].

Sealed instances. Attested properties are only valid if instances are *sealed* to block any tampering that might undermine the properties: their configurations are fixed and they cannot be changed by management APIs. Sealing requires that the application code itself incorporates its own management interface, which is subject to inspection at endorsement time. For example, a privacy-preserving survey program might provide APIs to close a survey and output aggregated data, but not to read individual responses.

2.2 Logical Trust in Latte

Latte uses a declarative logic data model to express secure metadata including attestation statements and endorsements of program objects. We define an exemplary vocabulary of predicates for Latte using ordinary safe Datalog [17], a simple and tractable logic language. Policies are expressed as packages of Datalog rules. A guard library incorporates an embedded Datalog inference engine to check compliance with logical policies.

Logic serves as the foundation of Latte's attestation architecture. We show that logic rules capture precisely how to validate chained attestations and combine them with endorsements and other assertions for rigorous and verifiable security checks. The vocabulary is easily extensible for a wide range of attestations, endorsements, and user-defined compound policies, decoupled from the Latte implementation. Extensions to the vocabulary require no change to the Latte framework itself: only the

Statement	Description
`runs(PID, ImgHash)`	Instance `PID` is launched from the image with hash `ImgHash`.
`config(PID, ConfKey, ConfValue)`	Instance `PID` was launched with configuration property `(ConfKey, ConfValue)`.
`bindToID(PID, NetAddr)`	Instance `PID` is bound to network address `NetAddr`.

Table 1: Simple attestation statements in Latte. Each statement asserts a fact—a logical predicate with constant parameters—attributed to an authenticated issuer.

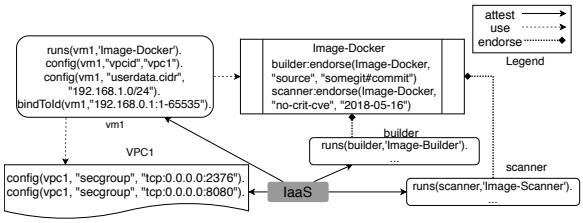


Figure 2: Metadata assertions in Latte. Logical statements include attestations of newly launched guest instances, source certifications for images, and endorsements of properties of code objects. The metadata service indexes each statement according to its subject (e.g., an instance or code object). Each statement is attributed to a speaker—a principal that issued the statement—which may also be an instance that is the subject of other published metadata. Statements are linked to both speaker and subject. For example, "vm1" is attached to "VPC1" and uses "Image-Docker".

logic templates and matching policy rules must change.

The logical approach allows Latte to address several challenges for cloud attestation. Importantly, logical rules can integrate statements published by multiple principals, enabling an authorizer to inspect the full stack of a distributed cloud application. We show how the logical structure supports querying instance properties and configurations (§4.1), layered cloud platforms (§4.2), authentication by network addresses (§4.3), source-based attestation (§4.4), and composition for horizontally scaled cluster services (§4.5). Generally, an authorizer can use Latte rules to specify who it is willing to listen to, what it trusts them to say, and what it needs to hear to approve compliance with its policy.

As we will show, logic enables a rich space of policies that combine metadata from multiple sources in a unified way. Evaluation cost scales with the complexity of the policy and the length of the compliance proof (§7). While complex logical policies take longer to evaluate than simple role-based ACLs, our results show that the relative cost of logical access checking can be negligible in practical scenarios. Moreover, caching intermediate results can reduce this cost [41].

To our knowledge Latte is the first use of Datalog for attestation and the first use of logic for cloud attestation. Previous uses of Datalog as a trust language include Binder [22], SD3 [34], RT [36] and SeNDLOG [7]. Nexus [56, 50] introduces logical attestation based on a more expressive logic; we contend that Datalog offers sufficient power and is fast enough for practical use. (See §8 for related work.)

3 Attestation in Latte

Latte defines an architecture and data model to expose attestation statements about instances and their configurations, maintain them in a cloud metadata store, query the store to retrieve groups of related statements, and process these statements to evaluate policy compliance.

3.1 Attestations and Endorsements

When a Latte-enabled cloud platform launches an attested instance it publishes one or more attestation statements (Table 1) about the instance's code identity and configuration parameters (key-value pairs). Latte's metadata service generates a unique principal identifier (PID) for each instance.

Instances or other principals may issue endorsements of code objects (source or binary) identified by a unique hash. Latte endorsers represent these properties as key-value pairs whose meanings are user-defined. Listing 1 shows example endorsements coded as logic assertions. The first is from Clair [47], a container image analyzer, that determines that a container has no known critical level vulnerabilities in the CVE database as of a given date. The second is made by an auditor that the image is sealed against remote shell access (*ssh*) once launched. The third endorsement binds a VM image to its source code on a GitHub repository.

Latte authenticates the issuing principal (speaker) of each statement and attributes the statement to its speaker. Any statement spoken by an instance within the cloud is attributed to the speaker's instance PID. A statement issued by an external principal outside the secure cloud network is attributed to the hash of the external principal's public key.

3.2 Validation Logic

Listing 2 presents a sample set of logic rules that an authorizer uses to reason from attestations and endorsements to infer properties of a running instance. We first illustrate simple inference rules for attestation, then refine and extend them to meet various goals (§4).

The syntax is standard Datalog extended with a **says** ("`:`") operator. Each predicate (or each atomic statement) has an optional **says** prefix that identifies the speaker; if the prefix is omitted then the predicate represents a belief of the authorizer (policy owner) itself. The capitalized parameters represent logical variables.

Facts F0-F1 and Rule R0 configure trust anchors for the policy. F0 states that the authorizer trusts the IaaS as

Guard Predicate	Description
`hasConfig(I, Name, Value)`	check if instance I has a configuration
`attester(I)`	check if instance I has attester property (§4.2)
`builder(I)`	check if instance I has builder property (§4.4)
`hasProperty(I, P, V)`	check if instance I has a customized property P of value V

Table 2: Some predefined guard predicates in the Latte guard library.

Listing 1: Sample endorsements.

```
endorse(img, "no-crit-cve", 2018-5-17).
endorse(img, "no-ssh", true).
endorse(img, "source", https://github.com/
boot2docker/boot2docker.git#2bb74c92).
```

Listing 2: Logic rules to infer instance properties from attestations and endorsements.

```
(F0) trustedCloudProvider("[IaaS-ID]").
(F1) endorser("[endorser-keyhash]").
(R0) attester(H) :- trustedCloudProvider(H).

(R1) runs(Instance, Image) :-
        Host: runs(Instance, Image),
        attester(Host).
(R2) hasProperty(Image, Property, Value) :-
        E: endorse(Image, Property, Value),
        endorser(E).
(R3) hasProperty(Instance, Property, Value) :-
        runs(Instance, Image),
        hasProperty(Image, Property, Value).
```

a cloud provider; F0 implies (via R0) attestations issued by the IaaS layer are trusted. F1 asserts that statements signed with a specified key come from an endorser.

Rules R1-R3 capture what it means to validate a simple attestation. Rule R1 implies that if a trusted attester—such as the IaaS cloud provider—asserts that some instance was launched from a specific image, then the authorizer believes it. Rule R2 implies that it trusts endorsers, so if any endorser asserts that an image has some specific property, then the authorizer believes it. Rule R3 implies that an instance launched from an image with a property also has that property (i.e., if an program is secure, then a process running the program is secure).

Suppose an authorizer runs a guard policy that requires the requester to have some specific property. The authorizer loads the logic in Listing 2 together with pertinent attestation and endorsement statements, and tests a goal statement `hasProperty` for the desired property. If a matching attestation (`runs`) and a matching endorsement (`endorse`) are present, then the inference engine in the guard library concludes that a requesting instance has the required property.

Listing 3: Logic rules using `builder` predicate and constrained list of endorsements.

```
(R4) builder(Instance) :-
        hasProperty(Instance, "builder", true).
(R5) hasProperty(Image, "builder", true) :-
        E: endorse(Image, "builder", true),
        trustedEndorserOn(E, "builder").
```

3.3 Retrieving Metadata

Each basic statement applies to a single subject. The subject of an attestation is an instance. The subject of an endorsement is a code object (source repository or image). The Latte metadata service (MDS) indexes the statements it stores by their subjects (Figure 2). It also stores a mapping from cloud network addresses to instance identifiers (PIDs) (§4.3).

An authorizer queries the metadata service for published metadata about an instance identified by network address. The metadata service returns attestations for the instance and any stored endorsements for its code. We show how this indexing can be extended to chained attestations and endorsements in §4. The Latte guard library also allows an authorizer to import additional logic content from external services such as a designated Web service or a certificate store (e.g., queried by image hash), as shown in authorization phase of Figure 1.

4 Logical Attestation in a Cloud

Latte supports flexible and general cloud attestation building on the logical foundation in §3, including extensions for common software deployment patterns.

4.1 Logical Guards

Guard policies can incorporate rules that require arbitrary conjunctions or disjunctions of basic properties to be satisfied. In essence, a guard is an extended access control list that identifies sets of instance properties that are compliant with the policy, e.g., what access is granted to a requester.

Beyond attestations about the running code, guard rules may consider instance configuration properties. Latte-enabled cloud platforms expose configuration metadata as attested key-value pairs (Table 2, §6). An authorizer can check for the presence of a specific configuration value with a guard rule requiring its presence. For example, Listing 4 shows rule R6 verifying that a

Listing 4: Logic rules to infer instance configurations.

```
(R6) isolatedContainer(Instance) :-
        H: hasConfig(Instance, "volume", ""),
        attester(H).
(R7) sparkMasterCmd(Instance) :-
        H: hasConfig(Instance, "cmd0",
           "start-master.sh"),
        attester(H).
```

Listing 5: Policy rule for inference of an attester.

```
(R8) attester(Instance) :-
        hasProperty(Instance, "attester", true).
```

Docker container mounts no volumes, so it cannot store any data persistently after it terminates. Rule R7 verifies that a Spark instance starts with command line "cmd0" indicating it a cluster master. We describe how each platform attests configuration in §5.

Guard policies are structured as sets of facts (e.g., trusted principals or other statements the authorizer believes) with guard predicates that help derive high-level properties of an instance.. For user convenience, Latte defines a library of useful guard predicates listed in Table 2. Authorizers can use these guard predicates as building blocks for more advanced policies. For example, rule R4 in Listing 3 infers a guard predicate called `builder` from an instance property; §4.4 and Listing 6 show how a rule for source-based attestation uses `builder`.

R5 in Listing 3 also illustrates how guard policies use `trustedEndorserOn` to constrain their delegations of trust to endorsers. The policy in Listing 2 trusts a named endorser to assert *any* property of the code object. R5 limits this trust: it requires that the authorizer trusts the endorser specifically to assert the `builder` property. This restriction on endorsement is applicable to inference of other principal properties, such as `attester` (§4.2).

4.2 Layered Platforms

We next show how to extend logical cloud attestation for layered environments, such as IaaS virtual machines running Docker containers running Spark programs. The premise of layered attestation is that any platform-as-a-service (PaaS) server may itself run as an attested instance, or even as a tenant of an attested instance. A PaaS server loads and runs code images in a PaaS-specific format (e.g., Docker container images, Spark .jar files), generating a new instance with its own network address. For attestation to be useful, we require that PaaS instances are isolated from one another and from external tampering, and that they have unique network addresses (discussed below). PaaS software must be extended with callouts to publish attestation statements for the instances that it launches. An example of a layered PaaS server is the TapCon secure container server outlined in §5.1.

An endorser asserts the `attester` property for an image that implements a layered execution service, after verifying that it meets these requirements. If an authorizer trusts the endorser, and a trusted cloud provider attests the PaaS instance running the image, then it can infer an `attester` property of the instance. Likewise, execution platforms launched within a PaaS platform can also be attesters. Rule R8 in in Listing 5 codifies this recursive definition of an attester.

Rule R1 in Listing 2 together with R8 can be applied recursively to validate the attestations issued by a layered attester. In this way, Latte's use of logic naturally validates *chains* of endorsements and attestations describing the entire software stack of an instance. To support fast verification of attestation chains, the Latte MDS maintains the lineage between image endorsements and attestation of an instance launched from that image, and the lineage between an instance attestation and its hosting platform attestation. Using this linkage, an authorizer can retrieve a complete attestation chain without unnecessary statements.

4.3 Network Authentication

Latte delegates network addresses hierarchically to ensure that each instance has exclusive control of a unique network address. If an instance is an attester, it controls a block of addresses and invokes the MDS to delegate addresses to its child instances at instance creation time. The MDS verifies that the attester controls the delegated addresses, and updates its map of addresses to instances. The attester must ensure that the instances it creates can transmit or receive only on their assigned addresses. This ensures that network addresses are accountable: each packet uniquely identifies the instance that sent it. The Latte prototype implements address delegation in OpenStack, Docker, and Spark (§5).

4.4 Source-based attestation

Many previous systems support attestations for binary program objects. However, software trust is often based on inspection or analysis of source code. For example, FindSecBugs [13] is a source-level analyzer that checks Java source code for common vulnerabilities, e.g., to ensure that the code sanitizes user inputs properly. A key basis for trust in open-source software is open inspection of the source code by a community. Furthermore, attesting source is critical for sharing data across tenants: how can a data owner they trust a binary program without knowing how and from where it was built?

Attestation is more valuable if an authorizer can apply

Listing 6: Policy rule to apply a source endorsement to an instance that is attested to run an image built from this source by a certified builder.

```
(R9) hasProperty(Image, P, V) :-
        B: endorse(Image, "source", Repo),
        builder(B),
        hasProperty(Repo, P, V).
```

safety properties of source code to binary objects derived from it. In general, that is possible only if the build chain is also trusted [59].

We extend Latte's logic with rules to reason about builds and other program transformations. Latte defines two exemplary endorsement properties: `builder` and `source` for this purpose. The `builder` property represents a belief that an image implements a *trustworthy build service*. A trusted build service runs as an instance that is attested as launched from an endorsed `builder` image via the rules in Listing 3 combined with rules R1 and R3 in Listing 2.

A builder issues an endorsement after each successful build, using the `source` property to assert that the image derives from a source repository fingerprinted by a secure hash. Listing 6 gives a logic rule to apply properties of source to a derived binary. Rule R9 says that if a certified builder B endorses $Image$ as derived from a source repository version $Repo$, and trusted endorser E says $Repo$ has property P with value V, then the derived image also has property P with value V.

With this extension, Latte provides a powerful mechanism for an authorizer to acquire trust in a service: if it trusts how an image was built from a source repository, and endorsements of safety of the source code, then it has a strong basis to trust an instance executing the image. AWS provides a hosted build service called CodePipeline [8]. The Latte prototype includes a hosted build service (§5.2). A Latte authorizer may examine the metadata of the build service instance recursively, as described above.

4.5 Grouping for Distributed Systems

Cloud-hosted services often comprise many server instances grouped in a cluster for horizontal scaling. These instances may share data or have other internal relationships and dependencies. It follows that trust in the integrity of a service as a whole requires some degree of trust in the integrity of all of its instances.

Latte implements a simple grouping mechanism to support clustered services. Our approach presumes that a cluster service consists of a group of *worker* instances led by a *master* instance. Each worker contacts the master to join the service group as a worker. The master controls membership in the group. Specifically, Latte grouping requires that the master acts as an authorizer to verify the metadata of each worker and validate its code identity and configuration.

After validating each worker, the master issues a statement to the metadata service granting the worker membership in a named group. This statement is stored with the worker instance metadata (example of Spark cluster in Figure 3). An authorizer that queries for the worker receives the metadata of the master as well for validation. An authorizer checks a worker by verifying (i) it is a member of a duly constituted service group and (ii) the master complies with its policy. If the master's membership requirements (i.e., code identity and configuration) are configurable, an authorizer can verify that the master's configuration meets its requirements.

The authorizer may optionally validate the worker's metadata. However, in our prototype the authorizer does not validate that the worker meets the master's requirements for membership in the group. Rather, it validates its trust in the master and then accepts that the trusted master has validated all of its workers. This optimization reduces validation costs substantially.

4.6 Other Extensions

Policies may incorporate supporting assertions (or rules) from any authenticated source, e.g., external endorsements or delegations of trust to other asserting principals. In particular, the logic system enables the guard library to derive trust in endorsers based on logical delegations issued (transitively) by other designated trust anchors, which may also be external and authenticated by keypairs. For example, a policy might allow an endorser to assert particular properties based on statements about the endorser by a more trusted party, e.g., the owner's employer, or an open-source consortium. An authority may also define groups of properties that the endorsers are trusted to assert. A policy might also delegate to another authority control over which cloud sites are accepted as attesters (e.g., a cloud federation root).

Latte supports end-to-end authorization in which each user can invoke the guard library to validate compliance with its policy for itself. The logical approach also makes it easy to designate an intermediary to act as an authorization service by checking policy and issuing assertions of compliance. Intermediaries are useful when metadata is proprietary or sensitive, and they can also reduce the cost of compliance checking by representing the result of a complex check with a simple assertion.

5 Exemplary Principals

We built several components that act in various Latte principal roles to demonstrate its capabilities and exercise our Latte prototype. This section describes services that run as attested instances and themselves act as endorsers, attesters, and builders. We show how these com-

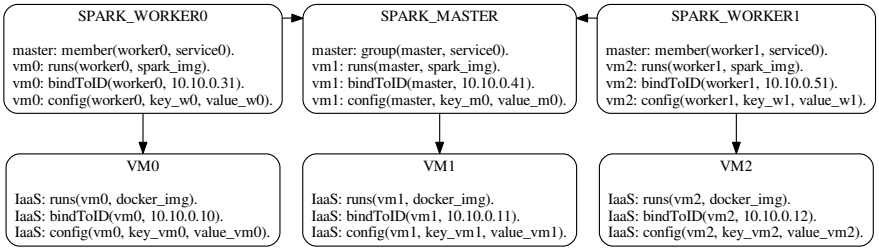


Figure 3: Chained attestations in an exemplary service–a Spark analytics platform with layering and composition. Each box has the attestations about an instance, labeled with their subject. A Spark worker instance at the top of the software stack is attested by a chain of attestations from lower platform layers, and statements from an attested cluster master admitting it to a cluster service group as a worker.

ponents work together in a complete example: an attestesd Spark data-analytics cluster in which Spark program identity can be used as a basis to grant access to sensitive data in an HDFS storage service. In this example, Spark jobs run on a cluster group of worker nodes controlled by a master, all of which run in attested Docker containers layered on attested OpenStack virtual machines.

5.1 Attesters

The core job of attesters (i.e., execution platforms) is to enforce isolation, delegate network addresses, and issue attestations.

OpenStack. We extend our previous work CQSTR [65] to issue statements to the Latte MDS. Before launching a VM, OpenStack attests the image used to launch a VM and IP addresses assigned to the VM. In addition, we use CQSTR's sealing mechanisms to lock down management APIs. Latte OpenStack attests VM's configuration from configuration files, which it parses into key/value pairs and publishes to the metadata server. Latte OpenStack only attests new VMs when requested by a tenant.

Docker. We extended Docker from our previous work TapCon [64]. Each container receives a unique IP address from an IP pool reserved by its hosting Docker service running in an attested VM. The Docker daemon attests containers to their images and configurations, i.e., container options, environment variables and launch command. As many deployments provide passwords via environment variables, we implement a mechanism to filter out some configuration keys. We reuse the sealing mechanisms from TapCon and disable privileged containers and certain administrative APIs that can break established attestations, e.g. launching of arbitrary shell commands in a container. To support delegation of individual ports, we extended Linux with a new system call to allow parent processes to restrict their children to a specified range of ports.

Spark. We deploy Spark in Docker containers using standalone mode, with a single master and multiple workers, all attested by Latte Docker instances. When workers contact the master, in checks and adds qualified workers to a Latte group. This naturally follows the bootstrapping steps of a Spark cluster. In addition, workers attest per-job task executors as a process instances.

5.2 Builder

We implement a *trustworthy build service* that runs a standard tool chain to build certain container and virtual machine images from a git source repository, and endorse the binary with a `source` assertion. The build service runs a pre-defined environment with standard build tools and takes as input the location of a code repository, including a revision hash and build script. The build script can download standard packages from a list of trusted binary repositories or compile them from source. The build service issues a `source` endorsement to bind the hash of a generated image to the source repository and revision.

A challenge is to bind the source of the build service itself to the running `builder` instance—it cannot build itself. To permit authorizers to trust the build service image, we use a *reproducible build* approach for the VM image running the build service [4]. This allows anybody to build the service image from source and verify it is correct. Our reproducible build is based on boot2docker [1].

5.3 Endorsers

We an attested endorser by adapting the image scanning tool Clair [47] to issue endorsements for Docker images. Tools such as Qualys for scanning VM images [38] could be similarly adapted. Clair runs in a container instance and checks whether a client-provided image has any known vulnerabilities (CVEs) above a given severity level. If so, it issues an endorsement of the image `"no-crit-cve", 2018-5-17` to indicate no critical level CVEs were found in the CVE database of up to the specified date.

5.4 Grouping

Our adaptation of Spark requires two overlapping groups. The Spark master creates the first group and authorizes all workers before adding them to the group (referred to as *SparkGroup*). Second, the Spark *driver* for a job creates a group (*DriverGroup*) of *executors*, i.e., JVM processes that get launched and attested by worker containers as members. The job submission process, when launching the driver, attests the user-supplied jar file as the image. This means an authorizer can verify what program the driver runs. When assigned executors by the master, the driver checks that the worker is a member of the master's SparkGroup. If this is successful, it adds the executor to its DriverGroup. As we describe below, this allows authorizers to verify which user program generated a request.

5.5 Guard policies

We describe a set of guard policies for our Spark cluster when accessing data in HDFS. An application is deployed as attested instance, either in a container hosted by Latte Docker, or on Spark's worker node.

The sample guard targets a data owner who wants to limit access to an instance (i) running a known Spark distribution (ii) without known vulnerabilities (iii) configured in a closed network (iv) running an analytics program endorsed as not leaking secrets. We assume that the data owner *Alice* trusts that the code in the Clair source repository correctly checks for vulnerabilities in container images. Alice also trusts an external endorser *Bob* to only endorse analytics programs (Spark jar files) with privacy-preserving analyses and implementations that do not leak data.

Listing 7 lists a complete guard policy Alice can put on her data to safely grant access to Spark executors. F2 states Alice trusts Bob to endorse a custom property `"no-leak"` indicting an image will not leak secrets. R10 extends the predicate "trustedEndoserOn" to trust an instance running the Clair source on the `"no-crit-cve"` property.

Rule R11-R13 captures the safety requirements needed to grant an executor access to data. The `safeNode` predicate requires a worker node to be admitted by a Spark master running Latte Spark, and runs locked down configuration. The Spark master is trusted to check the worker runs correct code and configuration. The `safeJob` predicate checks a driver has admitted the executor into its group . It further examines this driver runs on a worker node subject to safeNode rule, and the driver is attested by the worker to a qualified image (a .jar file endorsed by Bob). Finally, `grantAccess` combines the two rules to ensure that the requesting executor runs on a qualified worker node, and driven by a qualified application driver.

Listing 7: Sample access policies to protect grant data access only to certain analytic jobs.

```
(F2) trustedEndorserOn(Bob, "no-leak").

(R10) trustedEndorserOn(Instance, "no-crit-cve") :-
        hasProperty(Instance, "source",
                ClairSource).

(R11) safeNode(Node) :-
        Master: member(Node, MasterGroup),
        hasProperty(Master, "no-crit-cve",
                2018-5-16),
        hasProperty(Master, "source",
                LatteSpark),
        hasConfig(Master, "volume", "").

(R12) safeJob(Executor) :-
        Driver: member(Executor, DriverGroup),
        Worker: runs(Driver, AppJar),
        hasProperty(AppJar, "no-leak", true),
        safeNode(Worker).

(R13) grantAccess(Executor) :-
        Worker: runs(Executor, SomeImage),
        safeNode(Worker),
        safeJob(Executor).
```

5.6 Authorizers

For the Spark example above, we integrated authorization checks into HDFS. We allow users to install a per file policy, which will in turn be translated into a list of data block IDs. The policy is then distributed to each data node, so that each time a HDFS client requests for datablock, it will check the guard policy if the operation is approved. We also integrated attestation-based authorization into other applications.

Attestation protected credentials. We integrate attestation into OpenStack's authentication service Keystone, so that one can create a role with a guard policy specifying who can use the role. Keystone will check the guard at authentication time, and only issue an access token if the client instance passes the guard. This allows credentials to be bound to a specific software stack or even source repository, so that only instances using images built from target repository can authenticate with the role. With such capability, even if *the credential is lost*, anyone trying to use the credential must run in the exact same software stack.

Database protection. We implement connection-based protection in mysql-router proxy for MySQL [44]. The proxy loads a guard and verifies that all incoming connections are from approved instances. This can be used to harden the database port so it can only be accessed from certain instances.

6 Implementation of Latte

The Latte framework consists of the metadata service and the client library.

6.1 Metadata Service

The Latte metadata service (MDS) stores statements as objects in a key-value store indexed by the subject of the statement: instances (attestations) or code objects (endorsements). To accelerate fetching all the statements needed for authorization, the MDS automatically links instances to related statements, such as their launching instance and image. The Latte library defines an interface for authorizers to fetch the transitive closure of all statements pertaining to an instance.

The MDS is structured as a *front-end* that implements the client API and metadata management, and a scalable *back-end* storage service. The front-end is largely stateless and can be replicated for scalability. For the back end we use Riak [58], a distributed key-value store that is fault tolerant and scalable.

Network authentication and control. The MDS internally uses PIDs to refer to instances, and stores a map from network address to PID. All access to the MDS is authenticated by address using this map. To allow address delegation, an instance may be created with a range of addresses, which are passed to the MDS in CIDR format [24]. When an instance delegates addresses using the `bindToID` statement, the MDS verifies that the issuer controls the addresses (i.e., it is bound to a range including the addresses). The MDS uses a hash map and interval tree to cache the mapping of IP and port ranges to instance PIDs.

Caching and consistency. The front end of the MDS caches recently accessed statements. Most statements are immutable, and only need to be evicted when an instance is deleted. Similarly, the back end can use eventual consistency for most data: if a statement is missing it may temporarily cause a guard to fail, but retrying authorization will eventually fetch any missing statements. The one piece of data that requires strong consistency is the map of network addresses to PIDs: if an address is reused and an MDS is unaware, it may fetch statements for the previous instance using the address. We require strong consistency for `bindToID` statements. In addition, the MDS assigns a time-to-live, so that cached values will be re-fetched periodically.

Garbage collection. Statements are garbage collected automatically when their subject is defunct. An instance is defunct when it has terminated *and* the subjects of any statements it has issued are also defunct. An image is defunct when it has been deleted from the MDS *and* any instances launched from that image are defunct. Our prototype does not permit statements about a live subject to be withdrawn.

6.2 Latte Library

The Latte Library comprises two parts: an API for issuing attestations and endorsement statements and an API for authorization. Statements are marshaled as JSON requests and sent to the MDS using HTTP. The authorization functionality runs in a separate container, which which the library communicates via local RPC.

We implement authorization using SAFE [16], which uses the Styla Datalog engine [57] to check guard policy. The library takes as input a requester's network address, and contacts the MDS to fetch statements about the requester. Authorizers can also pass additional endorsements to consider, but the authorizer must verify the endorsements' authenticity. The implementation caches statements using the same rules described above.

6.3 Total Effort

Latte: Attestation and guard libraries comprise 4322 lines of C++ and the metadata service took 1761 lines of Go and 959 lines of python. We link against SAFE.

Attesters: We reused code from CQSTR and TapCon comprising 6000 lines of Python and Go, and 852 lines in the Linux kernel. Spark changes required 268 lines for adding attestations and authorizations, and a 133 lines wrapper in C to enforce port usage using our added system calls.

Authorizers: HDFS changes are 480 lines, the MySQL router took 278 lines, and we added 20 lines in CQSTR's version of Keystone. There are in total 710 lines for all guards.

Endorsers: The build service is about 500 lines, and the Clair scanner is 172, both written in Go.

7 Evaluation

The costs of Latte come from (i) issuing attestations during instances startup and (ii) evaluating guards for authorization. We evaluate these costs separately, as well as the overall performance overhead on applications

7.1 Evaluation Setup

We evaluate Latte on a 6-node cluster on CloudLab [2]. Each node has 20 Intel E5-2660 cores, 160GB memory, and two 10GbE NICs. The cluster runs OpenStack and Docker with modifications for Latte. We use four compute nodes for OpenStack, one for a network gateway, and one for the cloud controller. The metadata service's frontend runs on one compute node, and backend storage runs on the other three compute nodes. The 4 compute nodes are also used for applications such as Spark. We configure VMs with 4 VCPUs and 16GB memory.

7.2 Attestation Cost

The metadata service introduces the dominant overhead of attestation, as it must be contacted to issue and delete statements when instances start and stop.

Methodology We simulate parallel instance startup by running driver programs on multiple machines that issue the attestations needed to start VM, container, and process instances. We evaluate the cost for deployments using 1 (VM only) , 2 (VM + container), and 3 (VM + container + process) layers. The table below shows the number of instances at each level for each configuration.

Layers	VMs	Containers	Processes	Total
1-layer	204,800	0	0	204,800
2-layer	4096	50	0	204,800
3-layer	1024	50	4	204,800

The VM posts 10 configuration statements for each container. We run this experiment using 32, 128, and 256 threads.

We measure the total time to post all statements and report on the throughput and latency in Figure 4. We separately measure the latency and throughput of fetching all the statements for each instance.

Figure 4(e) shows the throughput for instance creation and fetch for a single MDS with 3 backend Riak servers, which store data on a SSD. With 32 threads, there is not enough parallelism to saturate the MDS, so throughput is lower. With 256 threads, throughput reaches its peak at 2500 create operations/sec and 3600 fetches/sec. Figure 4(a) shows a CDF for fetch request latency, and shows that latencies are generally below 50ms. With fewer threads, there is less queuing in the MDS and hence lower latency. Figure 4(f) shows the impact of layering on throughput. Layering increases latency slightly, as it requires more linking operations and fetches must return more statements. Overall, these results indicate that a single MDS is able to handle thousands of instance creations per second, which is suitable for a large network. Latency for issuing creation statements is generally much lower than the time to start a VM or container, although they may slowdown launch of very small processes.

The above results look only at cost of accessing the MDS. On an authorizer, caching can substantially reduce costs. Figure 4(c) shows the impact of caching statements and network address-to-instance mappings. Overall, performance without caching statements is 100x slower (not shown), and without caching network addresses is 5x slower.

Figure 4(b) shows the impact of group membership. We configure a master to create groups of 30 instances and measure the latency to fetch statements for the group master and members. Being a member has little impact on fetch latency, as it only incrementally adds to the number of statements. Fetching statements for a master, though, returns all the member statements and hence leads to larger tail latencies: 20% of fetch requests for a group master take longer 40ms.

Space usage is very low: each instance or image takes less than 1KB, so even for large networks all data can be kept in memory.

7.3 Authorization Cost

We measured the latency to evaluate guard predicates described in previous sections on containers (2 levels) and Spark executors (3 levels). For `builder` we check the building service instance. For `attester` we check a container. For the `cve` we check a single guard (adding more did not change the results). The isolation guard add more isolation conditions to the one in Listing 4. And the cluster guard refers to Listing 7.

Figure 4(d) shows the average latency and standard deviation for each of these guards when cache hits. Cache miss effect is consistent with Figure 4(c). The number of rules is shown after the guard's name. Overall, guards took between 5-10ms except for the cluster guard, which took 13ms. The time for logical inference alone is 1ms for most guards, and 8ms for the cluster guard due to its complexity (isolation guard has longer rules but fewer statements involved and used). The time above inferences comes from unoptimized guard parsing. To put these time in perspective comparison, AWS S3 access latencies are generally higher than 10ms. In the case of coarse-grained access, such as connecting to a database, the authorization check is only needed once. For storage services, the results of authorization can be cached and used for any object with the same guard (not implemented).

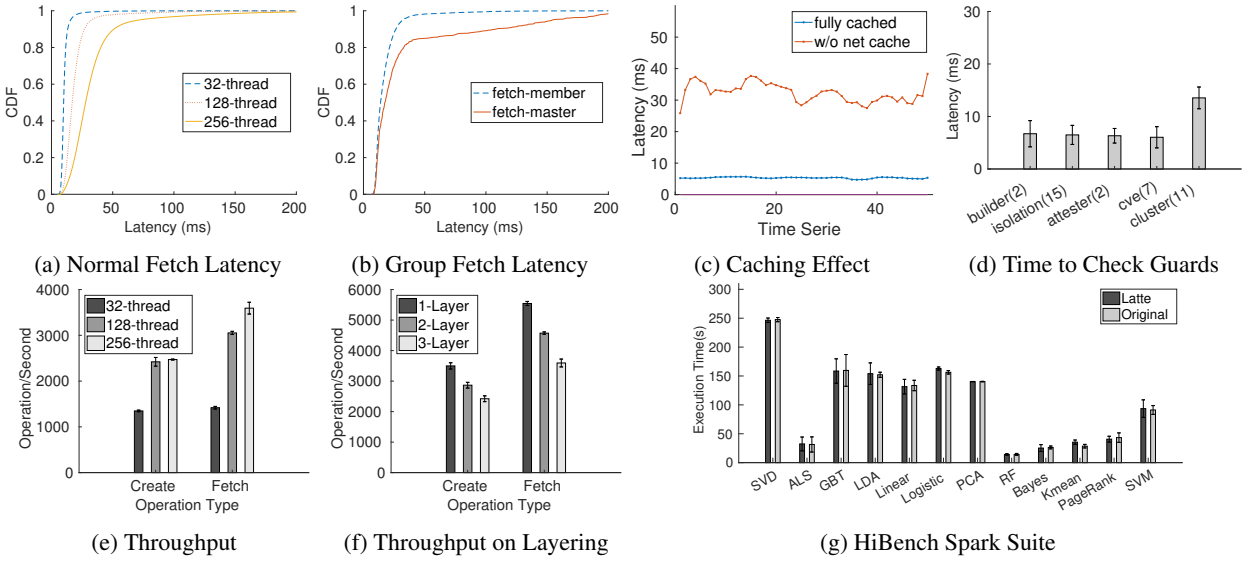
7.4 Applications

Finally, we evaluate application performance. For Spark, we use Intel's HiBench [33] bigdata benchmark, and compare Latte-extended Spark and HDFS performing authorization against their native counterparts. The execution time of "Large" dataset is shown in Figure 4(g). Overall, Latte performed identically to the native system. This is explained by the nature of analytics jobs, which are not storage bound.

For Keystone, we measure the latency to obtain the credential and use it with an unmodified OpenStack Swift storage service. Latte added 40ms, which is a 6% slowdown on small object access and less for large objects. Similarly, we compare the result of an OLTP benchmark dbt2 [45] with modified MYSQL proxy and no proxy case. The transactions per minute reduced by 2% with Latte, within the standard deviation of 3%.

8 Related Work

Code attestation. Several hardware-based attestation approaches have been proposed. Terra [26] and BIND [54]



(a) Normal Fetch Latency (b) Group Fetch Latency (c) Caching Effect (d) Time to Check Guards

(e) Throughput (f) Throughput on Layering (g) HiBench Spark Suite

Figure 4: Evaluation results. Unless noted, experiments use 3 layers and 32 threads. For figure(d), the number in parenthesis is the number of rules in guard.

use hardware TPM to make secure statements about what code is running in a VM. Similarly, Azure provides shielded VMs for stronger protection against rogue operators [42]. Haven [14], SCONE [12], and Ryoan [31] attest code running in hardware-protected SGX enclave which is tamper-proof after launch. These approaches are complementary and could be used as an alternate root of trust in Latte. Like Latte, Asylo provides a framework to integrate enclave attestations into application policies, e.g. a storage ACL [6]. However, these systems attest to binary hash of identify for a single instance. Our work enables reasoning about combinations of attestations and endorsements, extends software identity to source code, and supports third-party authorizers, making it suitable to protect inter-tenant interactions in the cloud.

Attestations provide a basis for access control to services and data. Singularity [55] and Nexus [56] explore attestation-based access control in an operating system of a single host. Latte's use of logical trust is similar to logical attestation in Nexus [56], but extends logical attestation to a cloud setting with distributed applications, and is based on standard Datalog logic. Like Latte, OpenStack Congress [3] allows verifying policy compliance using Datalog rules. However, Congress targets compliance checks for system policies using system-generated metadata. Latte extends this idea to multiple sources of metadata and policy and multiple authorizers, and incorporates software identity.

Software bases for trust. Property-based attestation [19, 49] advocates attesting to security properties of a given platform, instead of the binary identity. These properties are similar to Latte endorsements, and are attainable through formal verification and sandboxing. Formally verified systems [30, 29, 62, 35, 20, 39, 43, 52] may provide a strong basis for software trust and automated endorsers in Latte.

As containers are a popular technique to package, build, and deploy applications in the cloud, a number of tools and frameworks exist to secure containers: container scanning with Atomic [48], Clair [47], and DockerScan [21]; credential management with HashiCorp [28]; and certification of images with Docker Notary [23]. These can also be integrated into Latte as endorsers. Sandboxing [63, 25, 37] is another basis for software trust and is used in Nexus and Ryoan.

Secure computation. Cryptographically verifiable computation proves the correctness of remote computations [60, 53, 15, 61], and fully homomorphic computation [27] provides an alternate basis for safe computation on private data. However, to date they are substantially slower than native execution or are limited to restricted application domains. In contrast, Latte applies to a wide variety of existing software.

9 Conclusions

Many computing settings require high assurance in the code being run, which cannot be provided by current cloud computing systems. We propose that code attestation is a suitable primitive for establishing this trust. We show how it can be applied for flexible authorization policies. While Latte roots its trust in the IaaS platform, the same architecture can be used with trusted hardware such as SGX, by seeding the metadata service with statements that the processor is trusted.

References

[1] Boot2docker. https://github.com/boot2docker/boot2docker.

[2] Cloudlab. https://www.cloudlab.us.

[3] OpenStack Congress. https://wiki.openstack.org/wiki/Congress.

[4] Reproducible build project. https://reproducible-builds.org/.

[5] the software assurance marketplace. https://continuousassurance.org/.

[6] An Open and Flexible Framework for Enclave Applications. https://asylo.dev, 2018.

[7] M. Abadi and B. T. Loo. Towards a declarative language and system for secure networking. In *NetDB*, 2007.

[8] Amazon Web Service. AWS Code Pipeline. https://aws.amazon.com/codepipeline.

[9] Amazon Web Service. Life Without SSH: Immutable Infrastructure in Production. https://www.slideshare.net/AmazonWebServices/aws-reinvent-2016-life-without-ssh-immutable-infrastructure-in-production-sac318.

[10] Amazon Web Services. Vpc security capabilities. https://aws.amazon.com/answers/networking/vpc-security-capabilities/.

[11] Amazon Web Services, Inc. Ipv6 support for ec2 instances. https://aws.amazon.com/blogs/aws/new-ipv6-support-for-ec2-instances-in-virtual-private-clouds/.

[12] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. OKeeffe, M. L. Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[13] P. Arteau. Find security bugs. https://find-sec-bugs.github.io/.

[14] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, pages 267–283, 2014.

[15] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 341–357, 2013.

[16] Q. Cao, V. Thummala, J. S. Chase, Y. Yao, and B. Xie. Certificate Linking and Caching for Logical Trust. http://arxiv.org/abs/1701.06562, 2016. Duke University Technical Report.

[17] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166, 1989.

[18] R. Chandramouli, M. Iorga, and S. Chokhani. Cryptographic key management issues and challenges in cloud services. In *Secure Cloud Computing*, pages 1–30. Springer, 2014.

[19] L. Chen, R. Landfermann, H. Löhr, M. Rohe, A.-R. Sadeghi, and C. Stüble. A protocol for property-based attestation. In *Proceedings of the first ACM workshop on Scalable trusted computing*, pages 7–16, 2006.

[20] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *International Conference on Theorem Proving in Higher Order Logics*, pages 23–42, 2009.

[21] Daniel Garcia, Roberto Munoz. Dockerscan: A Docker Analysis and Hacking Tools. http://github.com/cr0hn/dockerscan.

[22] J. DeTreville. Binder, a logic-based security language. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 105–113, 2002.

[23] Docker. Docker Notary. https://docs.docker.com/notary/.

[24] J. Ellingwood. Understanding IP Addresses, Subnets, and CIDR Notation for Networking. https://www.digitalocean.com/community/tutorials/understanding-ip-addresses-subnets-and-cidr-notation-for-networking, Mar. 2014.

[25] B. Ford and R. Cox. Vx32: Lightweight User-Level Sandboxing on the x86. In *USENIX Annual Technical Conference*, pages 293–306, 2008.

[26] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-based Platform for Trusted Computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[27] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, 2009.

[28] HashiCorp. HashiCorp Vault: A Tool for Managing Secrets. https://www.vaultproject.io/.

[29] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th ACM Sympo-*

sium on Operating Systems Principles, pages 1–17, 2015.

[30] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, pages 165–181, 2014.

[31] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the 12th USENIX Symposium Operating Systems Design and Implementation*, 2016.

[32] IBM inc. Is your docker container secure? ask vulnerability advisor. https://www.ibm.com/blogs/bluemix/2015/07/vulnerability-advisor/.

[33] Intel Corp. Intel hibench suite. https://github.com/intel-hadoop/HiBench.

[34] T. Jim. Sd3: A trust management system with certified evaluation. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 106–115, 2001.

[35] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[36] N. Li and J. C. Mitchell. Datalog with Constraints: A Foundation for Trust Management Languages. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL '03, 2003.

[37] Y. Li, J. M. McCune, and J. Newsome. MiniBox: A Two-Way Sandbox for x86 Native Code. In *Proceedings of the Usenix Annual Technical Conference*, 2014.

[38] J. Lute. Qualys virtual scanner appliance. https://community.qualys.com/docs/DOC-3452-reference-qualys-virtual-scanner-appliance).

[39] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying security invariants in expressos. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 293–304, 2013.

[40] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, 2008.

[41] Microsoft Corp. Caching access checks. https://msdn.microsoft.com/en-us/library/windows/desktop/ff394767(v=vs.85).aspx.

[42] Microsoft Corp. Guarded Fabric and Shielded VMs on Azure. https://docs.microsoft.com/en-us/windows-server/virtualization/guarded-fabric-shielded-vm/guarded-fabric-and-shielded-vms-top-node.

[43] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 252–269, 2017.

[44] Oracle Corporation. Mysql-router. https://dev.mysql.com/doc/mysql-router/2.1/en/.

[45] Oracle Inc. Mysql benchmark tools. https://dev.mysql.com/downloads/benchmarks.html.

[46] PCI Security Standards Council. Official PCI Security Standards. https://www.pcisecuritystandards.org/document_library.

[47] Red Hat, Inc. https://coreos.com/clair.

[48] Red Hat, Inc. Project Atomic. https://www.projectatomic.io/.

[49] A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *Proceedings of the 2004 workshop on New security paradigms*, pages 67–77, 2004.

[50] F. B. Schneider, K. Walsh, and E. G. Sirer. Nexus authorization logic (nal): Design rationale and applications. *ACM Transactions on Information and System Security (TISSEC)*, 14(1):8, 2011.

[51] A. W. Service. AWS Artifact. https://aws.amazon.com/artifact/.

[52] A. W. Service. Automated Reasoning and Amazon s2n. https://aws.amazon.com/blogs/security/automated-reasoning-and-amazon-s2n/, 2016.

[53] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, 2011.

[54] E. Shi, A. Perrig, and L. Van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *Security and Privacy, 2005 IEEE Symposium on*, pages 154–168, 2005.

[55] D. R. Simon, A. Yumerefendi, T. Wobber, M. Abadi, and A. Birrell. Authorizing applications in singularity. In *Proceedings of the 2007 Eurosys*

Conference, March 2007.

[56] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical Attestation: an Authorization Architecture for Trustworthy Co mputing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 249–264, 2011.

[57] P. Tarau. Styla - a prolog in scala,. `https://code.google.com/archive/p/styla/`, 2012.

[58] B. Technologies. Riak is a Distributed, Decentralized Data Storage System. `https://github.com/basho/riak`.

[59] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[60] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 223–237, 2013.

[61] R. S. Wahby, Y. Ji, A. J. Blumberg, A. Shelat, J. Thaler, M. Walfish, and T. Wies. Full accounting for verifiable outsourcing. In *CCS*, 2017.

[62] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation,*, pages 357–368, 2015.

[63] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009.

[64] Y. Zhai, Q. Cao, J. Chase, and M. Swift. Tapcon: Practical third-party attestation for the cloud. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, 2017.

[65] Y. Zhai, L. Yin, J. Chase, T. Ristenpart, and M. Swift. CQSTR: Securing Cross-Tenant Applications with Cloud Containers. In *Proceedings of the 7th ACM Symposium on Cloud Computing*, pages 223–236, 2016.