Practical Third-Party Attestation for the Cloud

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

Submission Type: Research

Abstract

Central to establishing trust in a service is knowing what code is running. On a single host, this can be ensured by locally downloading and compiling code, including the operating system image. However, this is not possible for programs that are run by another party: even if the source code is known and available to a client, she cannot verify that a service is actually running that trusted code.

This paper describes a layered attestation architecture Latte in the multi-tenant cloud that provides source-based attestation and network-based authentication. Clients can verify that cloud services deployed in Latte are running code built from specified source repository. Latte's attestation enables new use cases such as joint data mining, where two data owners can agree on the code to use for data analysis, and then guarantee that only the specified code can access their data. We implemented Latte in Openstack, Docker and Spark to demonstrate how Latte can be used to improve cross tenant trust with attestation. We also show that our implementation can be extended to support more general software and platforms. The overhead of using Latte in most case is negligible.

1 Introduction

Cloud-based applications are increasingly structured as layered compositions, with components managed by different tenants. For example, a tenant might deploy and manage an elastic cloud-based micro-service as a platform for other tenant applications. Layering and composition create new trust relationships among tenants and new security challenges. This paper proposes an architecture for pervasive *software attestation* in the cloud, as a foundation to address these challenges.

Consider this *service composition* example: suppose tenant Alice launches a commercial website using Bob's storage service and Cindy's machine learning service. How can Alice verify that Bob and Cindy's services are properly secured and patched against known vulnerabilities, and will safeguard her data? Existing cloud infrastructure provides a partial solution, in that services can authenticate to their clients by name (e.g., with IAM [17]). However, authentication by name offers only "social trust": Alice must rely on the reputation of the service providers (Bob and Cindy). Service layering adds another dimension to the problem. For example, Bob's software may run over a Platform-as-a-Service (PaaS) environment provided by Dean, which further requires that Alice trust Dean's service as well.

One possibility is to rely on cryptographic mechanisms for secure computation, such as homomorphic encryption [29, 42], which can assure privacy and integrity for software on untrusted platforms [36]. This approach removes the need to trust hardware vendors (e.g., Intel) and infrastructure providers (e.g., Amazon), but its performance cost is prohibitive for most applications.

An appealing alternative is to use *software attestation*, in which parties produce and consume statements about running code objects as a basis to verify their security properties. Many new systems use attestation enabled by hardware support such as Intel's SGX [19]. For example, several systems allow a cloud tenant to verify that its code is deployed correctly and is safe from tampering by an untrusted cloud platform [22, 21]—an example of *first-party* attestation, in which the attestation statement is issued to the code owner, who has inherent trust in its own code and build chain, and knows the hash of the binary code object. Our goal is to expand support for *third-party* attestation, in which the statement is consumed by a client of the attested service to check compliance with the client's security policy. Ryoan [32] is an early example of third-party attestation enabled by SGX.

The goal of our work is to enable practical and general third-party attestation that is compatible with existing software stacks in cloud environments. A key challenge is to enable reasoning about service compositions with multiple layers and elements. For example, it is easy to build trust in compact applications written with powerful programming formalisms at the top of the cloud stack, but this requires trust in the supporting infrastructure, which may be larger and more complex but also more widely used and inspected, e.g., by an open-source community. An effective architecture must address sev-

eral key safety challenges: (i) link binary artifacts to inspectable source code via a secure build chain, (ii) authenticate instances to bind attestations to instances securely; (iii) combine multiple trust assertions to infer properties of compound deployments in a way that is provably sound; (iv) use these properties within policies to control access to protected data (*attestation-based access control*).

We leverage several observations about cloud environments in our design:

- Cloud providers tightly control their infrastructure and are (generally) trusted by tenants; hence they can serve as a root of trust. Our architecture is compatible with a hardware root of trust (e.g., SGX), but does not require its cost or complexity when the cloud provider itself is trusted by the tenants.
- IaaS providers control their networks and prevent spoofing and sniffing of packets by tenants, so we can use network as secure identifiers of service instances.
- Much of the code running in clouds originates from public code repositories, which can provide trust through a large developer and user base.
- There is a rich and growing ecosystem of code analysis tools that can automatically identify common security bugs and verify software safety properties [33, 24, 31, 30]. Attestation enables us to apply these sources of code trust to deployed instances.

Based on these observations, we propose the *Latte* architecture[1] for attestation in layered platform stacks anchored in a trusted IaaS cloud. A manager at each level of the stack makes *statements* about the code it launches, including the identify of the code, the configuration of the instance, and the network endpoints controlled by the instance (i.e., IP address and port ranges). Other parties may combine these statements with external statements endorsing properties of the software, and check them against security policies expressed as logical rules. Latte authenticates instances by their network addresses: this approach is compatible with existing software and reduces the need for expensive asymmetric cryptography and the burdens of managing keypairs. Finally, attested software objects support a limited management interface to prevent tampering of instances after launch.

Latte enables cloud tenants to use a logic engine to combine trust chains from multiple roots of trust (anchors)—principals the client trusts *a priori* to establish trust in other entities. Trust anchors in Latte include: (i) IaaS cloud providers, who are trusted to launch VM instances from images, to attest those instances, and maintain a secure network; (ii) secure build services, which link a binary artifact to a source code repository; and (iii) endorsers, who assert security properties of source codes based on inspection or automated analysis. Any of these anchor principals may itself be attested as part of a larger composition; in principle, the entire edifice may be grounded in hardware roots of trust such as SGX.

We implement Latte on OpenStack with CQSTR [43] to manage virtual machines. To demonstrate its layering capabilities, we built an extended Docker platform called TapCon that runs in an attested VM and itself attests the containers that it manages. We also extended the Spark analytics platform to support attestation of Spark programs and controlled access to data by trusted programs, and deployed Spark in attested TapCon containers. We show how layered attestation for Spark and PredictionIO [12] can support trusted programs that process private data sets owned by multiple distrusting parties (see §5).

We first motivate the use of attestation and discuss our trust model (§2). We follow with description of the Latte architecture(§3) and implementation (§4). Finally, we describe application use cases (§5) and evaluate performance (§6).

2 Background and Trust Model

Motivating example. Consider the problem of *joint data mining*: suppose that two groups each have a private dataset and want 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 to one another. They choose to leverage cloud infrastructure and common platform frameworks—an open analytics stack in Docker containers—to deploy P and grant it access to A and B. How can they ensure that their datasets are accessible by a VM instance only if it runs the correct P?

Attestation-based access control. We propose *attestation based access control* as a solution to this problem. Each group installs an access policy that permits data access from an instance running P. The cloud storage service that stores A and B verifies that requesters comply with a policy that the data owner provides or approves: the request originates from a suitably trusted application (P) running in a secured environment. A and B might grant access based on the identity of P or on endorsements of its general safety properties by trusted parties.

Similar scenarios are increasingly common in cloud computing as service granularity becomes finer with micro-services and lambda functions [18]. Layered attestation makes it possible to base dynamic trust on properties of the code running in the interacting instances, rather than on the owners of those instances.

[1] Layered Architecture for a*TTE*station.

IaaS provider as a root of trust. Our implementation of Latte grounds all attestation chains in trusted infrastructure-as-a-service (IaaS) providers. These providers launch virtual disk images as virtual machine instances, and fully control the platform infrastructure, hypervisors, hardware, and network. This model follows our earlier work on CQSTR [43], which extends OpenStack to support basic attestation of VM instances.

While not absolute, this trust is well founded: cloud providers often have ample resources to devote to security, and market forces push them to improve their security practices continuously. Trust in the IaaS provider could be social (reputation), or based on auditing and endorsement by a trusted third party, or (in principle) on hardware-based attestation of the IaaS hypervisors and cloud control system.

Specifically, we rely on cloud providers to (i) attest that virtual machine instances are faithfully launched from named virtual disk images, and (ii) secure their networks against spoofing or sniffing attacks, to protect and authenticate network traffic without cryptography at the application layer. Major commercial providers do assure these properties, e.g., Amazon AWS, Google Cloud, and Microsoft Azure.

Secure code provenance and source analysis. A key principle of Latte and third-party layered attestation is that it should be possible to bind attested code securely to a hashed source code repository that produces the attested binary artifact. Secure code provenance can harness new software analysis capabilities that operate at the source level. For example, a vulnerabilty scanning service (e.g., Swamp [16]) might certify that a program *P* passes a check, or that it *P*'s configuration has been patched for known vulnerabilities such as Heartbleed [7].

But even if we can verify and certify safety properties at the source code level, how can we be sure that a deployed binary was produced from exactly that source code without tampering? A secure binding to source code requires two properties: (i) verifying that the code compiled to create a binary comes from a known version of a code repository, and (ii) using a trusted environment and trusted tools to compile the code into a binary [40]. Property (i) can be achieved using source-code control tools such as Git [41] that provide a hash over the code comprising a version; selecting a source repository and version hash is sufficient to specify the specific source code to be used. Property (ii) can be achieved using a certified build platform and tool chain.

But this introduces a circular dependency: how can a certified build platform be created to build code from source, as the platform must exist as a binary? Latte addresses this problem with support for *reproducible builds*—the ability to build source code at a later time and on a different platform and produce a matching binary. This capability allows a client to trust an existing binary hash by building it locally from source and then verifying the local build matches the binary used as a compilation platform.

For example, Latte relies on a standard virtual machine image to compile container and application code. This virtual machine image can be verified by any client of Latte by downloading the code for the image, compiling it locally, and then comparing the resulting binary against the standard image. It is not necessary for every user to perform this check: the potential for any party to validate it grounds our trust in the binary artifact broadly in the community.

As we discuss in Section 4, we can support reproducible builds only within certain limits. This makes it difficult to use for all programs. Instead, we show how to use reproducible builds for the base build service (as part of the TapCon platform), which provides a trustworthy foundation for certified builds of other source programs to run in TapCon containers.

Ownership. The owner of a deployed cloud instance usually has complete control over the instance and can change it in arbitrary way through management interfaces. To make an instance trustworthy, it is critical to block any tampering of the instance after launch in any way that could undermine the attested properties. We presume that any instance owner—e.g., of a virtual machine, container, or process—is a potential attacker.

To eliminate this threat, we must first secure any back doors in the application source code itself. We view this aspect as an element of the safety endorsement based on the source code: any management APIs in the program must be subject to inspection at endorsement time, and considered by any analysis. For example, a trusted program such as a privacy-preserving survey might provide management APIs to close a survey and output aggregated data, but not to read individual responses. Second, we must close any dangerous management channels in the underlying platform. Our approach restricts dangerous management channels at the cloud API for attested instances.

3 Attestation Architecture for the Cloud

The Latte architecture is a practical layered attestation architecture for the cloud. A host instance at each layer (e.g., IaaS, PaaS, application) issues attestation statements for any guest instance it launches. Figure 1 illustrates how attestations are chained through the layers and rooted in the IaaS layer. Furthermore, the binary for an instance can be linked securely to a specified source version by an attested build service that secure assertions about what was compiled. The union of all of these statements can prove exactly what code is running in a ser-

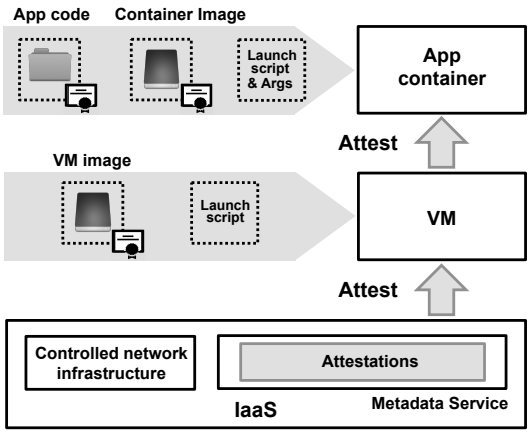

Figure 1: **Layered attestation example: a trusted IaaS service attests a sealed Latte VM, which launches and attests application containers. The shaded areas represent attested programs and their attestations stored in an IaaS metadata service. Latte attests to the container's image, application code, and the launch script and parameters.**

vice.

Latte supports a novel combination of features for practical cloud attestation. Attestations available to end user is property based: whether a remote party has a certain property. These property based attestations are issued in logic sets and linked to form chains that connect a running service back to a set of trusted roots, the service's source and the IaaS platform. Latte leverages the trusted network of IaaS platforms to authenticate instances at each layer and use source-based attestation to reason program properties. With these, the architecture naturally extends across multiple layers.

For example, in the joint data mining scenario, the instance running P can be in a container managed by a platform-as-a-service (PaaS) layer such as Docker Swarm [27], which itself is on a virtual machine launched by the IaaS. In this environment, a user wanting to ensure data is used correctly can validate that P complies with the policies of the owners of datasets A and B, that the stack upon which P runs is trustworthy, and that the code running P represents a faithful compilation of the trusted source for P.

Attestation elements include Latte are *instance principals* and *images*. A principal represents a running body of code that can be launched independently at some layer of the system, such as an OS, a container, or an application process. In addition, some principals may server as *managers* in that their primary function includes acting as hosts to launch guest instances at the layer above. For example, Docker is a manager at the PaaS layer that launches containers. An image is a body of code and data that can be executed to form an instance principal, much as an executing program creates a process. An image may be a VM image, container image, or application. Each instance principal associates to a UUID provided by its parent/host instance.

Latte stores attestations and assertions about principals and images in a *metadata service*, which is an IaaS-provided storage service. Principal and image assertions are present in the form of *logical statements*, which can precisely capture attestations and property assertions in various layers of the system§ 3.2 and allow efficient reasoning on a set of relevant statements. An assertion can be made by any principal in the system and it includes the issuer principal ID that provided the statement. For example, a container instance principal may have a property, provided by the OS that launched the container, specifying what image the container runs.

Managers publish attestations to the metadata service when they launch new guest instances. For example, an IaaS service manager publishes properties about the VM instances it launches, and a container manager (e.g. TapCon Docker) publishes properties about the containers it launches. The metadata service stores the authenticated identity of the issuer for each statement.

There are two special kinds of assertions. First, statements made by a third party about an image can be useful to incorporate external information and make the system open. This allows, for example, statements that certain image is trusted a priori. A *builder* property on an image specifies the principal that compiled the image and a URL specifying the source code for the image (e.g., a Git URL). Similarly, principals have an associated *image* property providing the UUID of the image used to launch the principal.

With these properties, the client of a service can verify a complete set of attestations about the service: its code, its host manager code, and so on down the chain. Furthermore, it can verify not just the code, but what instance compiled the code for the service (and other layers), and by following the image properties for that principal, recursively how the build instance itself was built. Figure 1 shows an example of layered attestation.

Reproducible base image. These chains must eventually lead back to an inital binary used to build the other images. We rely on a *reproducible build* to verify the base image. This allows anyone to verify the binding of the base image to source by building a copy locally and comparing its hash against the base image.

Most build processes are not reproducible, as they introduce differences through time stamps, deliberately introduced entropy, build counters, or non-determinism (e.g., random ordering of parallel build processes). For example, Debian has been working on this topic for years, but still has thousands of packages that are not reproducible [3]. TapCon works around this by requring only that the base image is reproducible. A TapCon instance running the base image then bootstraps trust by building the remaining images and issuing certified state-

ments about them.

Source-based program properties. Knowing the source code for a service can help provide assurance it is correct, but is not sufficient due to bugs and other vulnerabilties. TapCon provides facilties for high assurance by allowing external principals (e.g., vulnerability scanning services or auditors) to publish statements about source code to the metadata service, authenticated by public key.

3.1 Network-based Authentication

Latte uses network endpoints to authenticate principals. The source IP address and port can be linked to the principal controlling that address and port. The binding of a network address range and principal is stored in the metadata service by managers when they launch a new instance. Thus, the IaaS service initially publishes a property for the principal of a new VM stating it controls all the ports at the VM's IP address. When the container manager in the VM launches a container, it publishes a property on the container with its restricted port range.

Port management. Within a VM instance, all instances share the same IP address but use different ports. We further extend network control to the software stack on an end host by adding support for *safe port management* for processes. The objective is to allow a parent instance to safely delegate a range of its local ports to a child instance it spawns. When launching a new process, the parent process can specify a subset of its port range to delegate to the child process. This has two effects: the parent can no longer use that port range, and the child can only use that port range. Furthermore, processes are prevented from later expanding their port ranges. This process also ensures that the port ranges for separate instances do not overlap. We describe the details of this mechanism in §4.

Managers of each layer needs to post network address bindings of its children to the metadata service, too. Also, any IP addresses outside the cloud can not be authenticated by this means, so they cannot create new principals or change existing network address bindings.

3.2 Logical Trust for Latte

Logic offers a natural formalism to represent layered attestations, program endorsements, network address delegations, trust anchors, access policies, and validation rules. Our logical inference engine is called *checking service*, which is independent of the other components. It fetches attestation statements from the metadata service and composes them as DAGs. Then it follows inference rules to determine whether a principal is linked to its source, or more specifically, whether a principal has a property that is associated with a source repository.

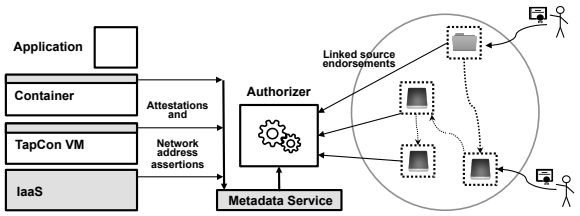

Figure 2: **An application principal that wishes to perform attestation-based access control checks is an *authorizer*. It runs an off-the-shelf logic engine in a library or process to evaluate logical guard conditions according to its local policy rules. The logical rules evaluate code attestations published at each layer, and endorsements of the attested code.**

Layered attestation chains. An attestation chain is a linked DAG of trust logic statements that establishes the code identity and network address(es) of the instance at the top of a cloud stack (Figure 2). For example, it may have statements about the source repository for its code, the build chain that produced an image from the source, and the layered platforms below it. Any relying party or "authorizer" may use an off-the-shelf logic engine to check compliance by evaluating specified conditions—e.g., guard conditions for access control—against sets of these authenticated logic assertions and policy rules governing the authority of speakers to make those assertions. Figure 3 shows statements across layers are linked to form compact attestation chains anchored at the IaaS.

Listing 1: Policy rules to validate a layered attestation chain.

```
(R1) runs(Instance, Image) :-
       runsInstance(H, Instance, Image),
       attester(H).

(R2) runsInstance(H, Instance, Image) :-
       AuthNID: attest(Instance, Image),
       bindToID(H, AuthNID).

(R3) attester(Instance) :-
       runs(Instance, Image),
       E: endorseAttester(Image),
       attesterImageEndorser(E).

(R4) attester(H) :- trustedCloudProvider(H).
```

Validation rules. Listing 1 shows simplified rules R1-R4 that an authorizer uses to validate an attestation chain. Each rule has a *head* on the left, which represents a belief that is implied by ("`:-`") a list of subgoals in a *body* on the right: the head is true if all subgoals in the body are true, under some assignment of string values to variables (capitalized terms).

R1 allows an authorizer's logic engine to verify that an instance *I* runs a program image *P*, if some valid host instance *H* attests that *I* runs *P*, and *H* is trusted to issue such attestations. The value of *P* is a hash over the code for *P* and and its configuration, as shown in Figure 1. **R2** concludes that *H* attested *I* if the attestation was spoken from a network address that is bound securely to the host

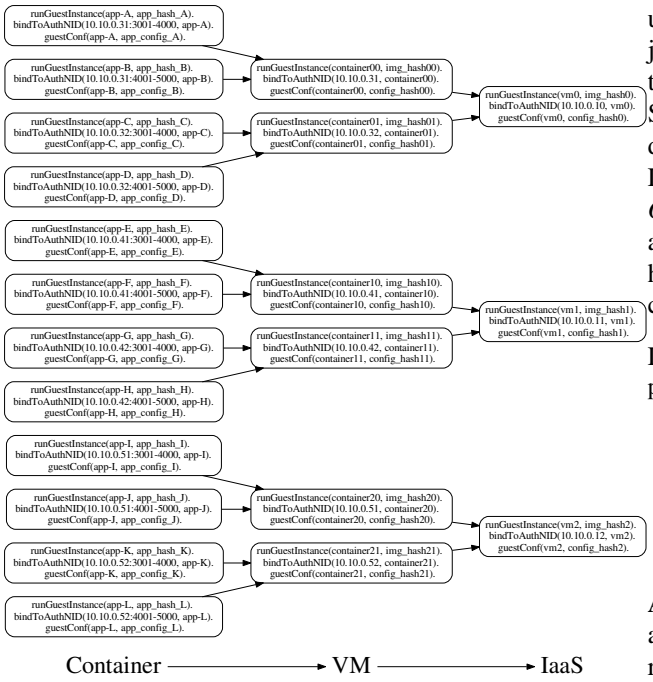

Figure 3: **Linked attestations in Latte: attestation to an app is linked to the attestation of the host container; attestation to a container is linked to the attestation of the host vm; an attestation chain is anchored at the IaaS.**

instance *H* by other rules for *bindToID* (not shown). It could use other authentication methods (e.g., keypairs) to establish the binding; this is a form of reconfigurable authentication [34]. The recursion in **R3** enables these rules to check layered attestation chains of any depth. **R4** is a basis for the recursion: it trusts attestations from a trusted IaaS provider.

Validating an attester. **R1** requires that the attesting host *H* at each step is accepted as a valid *attester*: as we have explained before, this property captures the belief that *H* is faithful in launching guest instances, binding them to secure network addresses, and attesting to the guest programs. This condition is satisfied if, for example, *H* is a trusted IaaS cloud provider (**R4**), or if *H* itself attested by its own host as running a secure program, and some endorser *E* endorses that program for the `attester` security property (**R3**). Trust in the endorser *E*—and in the IaaS provider—is also derived from authenticated (e.g., signed) logic statements and/or local policies (not shown). An authorizer could, for example, accept endorsers that are approved by its enterprise or by an open-source consortium.

Authorizing data access. In our joint data mining scenario, a policy decision to grant access to dataset *A* or *B* can be based on the identity of the code running in the requesting instance, and the data owner's beliefs about the security properties of that code. As shown in Figure 2, the data owner attaches an access policy to an object. The data owner trusts the authorizer—in this case, the data storage service—to apply its policy faithfully. Similarly, the data owner places trust in third-party endorsers according to its other rules, as discussed above. Listing 2 shows an exemplary rule that verifies (i) that the *Owner* says that *Endorser* is trusted to endorse *Property* about the program, (ii) that *Endorser* says that *Program* has *Property*, and (iii) *Owner* says the object can be accessed by a program with *Property*.

Listing 2: Policy rule that grants access based on accepted properties of an attested program that the requester is running.

```
hasAccessPrivilege(Program, ObjID, Owner) :-
  Owner: trustEndorserOn(Endorser, Property),
  Endorser: hasProperty(Program, Property),
  Owner: accessPrivilegeByProgramProperty(
                                Property, ObjID).
```

Authorizing access from workers. Applications such as Spark form clusters across containers. Data access requests are often sent from individual workers. To authorize access from a worker to data objects on a store, an authorizer checks i) the worker is running code with accepted program property; ii) a master has endorsed the worker as a member of the cluster; iii) its master is also running right code. Listing 3 shows logic rules we use to authorize a worker's data access requests.

Listing 3: Policy rules used to grant access to a worker running in a cluster.

```
ApproveAccessFromWorker(Worker, ObjD, Owner) :-
  approveAccess(Worker, ObjID, Owner),
  Master: clusterMember(Worker),
  approveAccess(Master, ObjID, Owner).

approveAccess(Instance, ObjID, Owner) :-
   runs(Instance, Image),
   hasAccessPrivilege(Image, ObjID, Owner).
```

3.3 Latte service APIs

We defined a set of APIs on the metadata service through which cloud instances can send requests to post statements for new guest instances or new images (Table 1). In attestation-based access control, we defined a set guards (Table 2) to perform authorization on a running instance from an authenticated network address.

3.4 Security Analysis

The main goal of Latte is to guarantee properties of a running service to third parties, either clients of the services or other services invoked by the attested service. Here we discuss the threats that Latte addresses as well as threats it does not.

Service operators. Normally services are completely vulnerable to operators who can remotely log in, access all data in use by the service, and control or change the

API	Description
`launchGuest (GuestID, ImageID, IP, PortRange, Config)`	Post attestation statements for a guest instance using the parameter values.
`createImage (ImageID, SourceURL, Config)`	Post statements for a new image.

Table 1: **APIs implemented in the metadata service.**

Guards	Description
`attest (IP, Port)`	Attest what is running in an instance at a network address.
`attestProp (IP, Port, Property)`	Check if an app running at a network address has an accepted property.
`accessObject (IP, Port, DataObject)`	Determine if an app running at a network address has access to a data object based on accepted program properties.

Table 2: **APIs implemented in the trust script of an authorizer using attestation-based access control.**

service's behavior. Furthermore, operators may be able to use management APIs at the IaaS or Paas level to control services. With Latte, security against this kind of attack depends on the secured service and all layers of code below it not allowing operations that would change the attested guarantees of code. For example, the operating system cannot be secured if it allows remote logins an the ability to load kernel code dynamically.

If, however, management APIs are restricted to prevent such remote acces, then Latte can secure client data against access by the operator: it can ensure the system was booted from a secure configuration that prohibits remote changes.

Buggy code. Latte does little to secure code containing vulnerabilities that can be exploited remotely. However, if these bugs are found, Latte allows a client to verify that a service is using patched code by checking the statements about which repository and code version were used to build the service. In addition, Latte allows clients to verify that security scanning tools were run as part of the build service and verify that they did not find any bugs. Despite these efforts, some bugs may persist. Thus, Latte is best deployed with a defense-in-depth strategy that also relies on restrictive access controls and network protections such as firewalls to make it more difficult to exploit vulnerabilities.

Fake attestations. The security of Latte depends on the security of each layer: a compromised IaaS or PaaS manager could publish false statements about services they launch. Similarly, a third party could publish false claims about a principal, such as claiming a repository should be trusted. In this case, client's can detect that these claims are not linked to a trusted root (i.e., a trusted repository or IaaS provider), and will therefore ignore the claims when verifying attestations.

4 Latte Implementation

We implemented Latte in a layered system with OpenStack as the IaaS layer and Docker as a PaaS layer. In §5 we extend the stack upwards to include a Spark analytics service that is hosted on the PaaS and attests Spark applications. This section describes our extensions to both OpenStack and Docker, and services for certified builds and sharing of attestations. In total, our implementation comprises about 9000 lines code based on CQSTR [43] and SAFE [25].

4.1 OpenStack Extensions

We based our implementation on CQSTR, which is itself based on OpenStack Kilo [11]. CQSTR provides a metadata service to publish attestations and other security assertions. CQSTR also supports "sealed" VM instances that restrict the use of IaaS management APIs that an attacker can use to subvert a running instance, such as backing up the VM or setting up tunnels that bypass firewall rules. To prevent changes to an instance from outside, Latte disables access to port 22, blocking remote ssh, so that only sealed services with self-contained management APIs may be used.

CQSTR provides general mechanisms to restrict VMs to boot from a select set of images. Our Latte prototype initially limits VM launches to a single *Root VM Image*, described below, which implements a platform named TapCon for secure Docker containers. We implemented a policy that also allows launch from any VM image that is *built by* the root image using the certified build service in TapCon (see below).

Root VM Image. The *Root VM Image* is the base OS installation for all containers running on Latte. We use a customized version of *boot2docker* [1], which is a lightweight distribution designed to run Docker containers, and the only service it runs is our TapCon extension of Docker. The image uses Linux kernel version 4.4.39 with corresponding AUFS patches from Docker. The image itself requires a standard Docker daemon to build.

The Root VM image is the only image in Latte that *cannot* be certified by Latte. As described previously, we instead rely on reproducible builds to allow any client to verify the image by building a replica from source for comparison. Table 4.1 shows the steps needed to allow boot2docker to be built reproducibly, which are based on published instructions [14]. The largest causes of binary mismatch are timestamps embedded in libraries.

So far, we can reproducibly build everything in the *boot2docker* image except the Docker daemon itself, and there is no publicly available build of TapCon Docker.

1. Remove unnecessary software.
2. Add all source explicitly downloaded with *curl* during build to Git source repository.
3. Use *faketime* tool [5] to generate identical build timestamps.
4. For packages installed by a package manager (e.g., *apt* or *dpkg*), specify the exact version and required hash, which is verified during build.
5. Instruct GCC to omit build IDs.

Table 3: Steps for reproducible image build.

We work around this with a two-step process that instead uses a certified build for just Docker. When first launching Latte, we use an initial Root VM image that specifies which version of the official Docker source to use from the Docker repository (relying on Git to reliably copy and verify the hash of the source code). During the first launch of this image, the VM applies our patches (below) to the Docker source and compiles Docker binaries. To avoid such bootstrapping on every VM launch, the initial VM generates a new VM image including TapCon Docker during this process and registers its source and build properties into the IaaS metadata service. Thus this image can be used for subsequent secure VM launches.

4.2 Docker Extensions

Our extended Docker creates and deletes container principals and publishes statements about its containers in the metadata service. It also delegates port ranges to specific containers, so that the source of network traffic can be identified from the port and IP address.

Docker modifications. We made changes to Docker to (i) post statements about newly launched containers, (ii) limit what images can be launched, and (iii) limit administrative access to Docker. TapCon adds a container monitor that detects container start and stop. On detecting a change of state, the monitor calls the metadata service to create and delete principals and images. We implemented this as a separate service to minimize changes to Docker itself, but it could be integrated.

Like our modified OpenStack, we modified Docker to limit the allowable images to launch. Currently, it launches only images built on the Docker host; it allows no external Docker images except for a small number of white-listed images that contain the base OS distribution (e.g., Debian or Ubuntu). We rely on only a few existing reproducible packages like modified gcc [15] tools, and we compile the initial Docker images from scratch.

For each container built and launched, TapCon posts the source information and the base container image to the metadata service. The metadata is trustworthy because TapCon runs a verifiable image that is known to launch only allowed images and to post their metadata faithfully. Then through the base container image properties, one can obtain a full chain of container build scripts grounded in the initial white list packages.

We limit the functionality of standard Docker to ensure security properties are not violated through management APIs. To do so, we first use an authz plugin [4] to put restrictions on API usage. TapCon's Docker cannot execute command shells in the container, directly copy to or from a container, or launch privileged containers (i.e., with root or other elevated access to the OS). Mapping of host path as volume is also prevented; instead docker assigns a random path.

Port Management. Latte uses network endpoints for authentication, so TapCon ensures containers use only ports assigned to them. Docker already handles assigning server ports to containers. We extend this functionality by also limiting what client ports a container may use when initiating a connection.

We extended the Linux kernel to support *allowable port ranges* on clients and processes. The kernel drops any packets from ports outside this range. We added 5 system calls to manage port ranges. The container manager sets the port range for containers it creates and specifies the range via environment variables. Similarly, a process running in a container can refine the allowable port range for its child processes, which is useful for application services such as Spark that also launch new code.

4.3 Metadata Service and Guards

We implement the prototype metadata service using SAFE [25], which adds scripted linking to connect related statements in DAGs that match the delegation structure, and publishes linked trust statements in an indexed put/get metadata service. SAFE scripts provide an API for a principal to issue a certificate of statements according to a pre-defined logic template. To enable efficient retrieval of relevant attestations, SAFE allows a certificate template to include programmable *link* statements to refer to dependency certificates.

For example, the scripts for Latte provide a generic API for a host layer to attest a guest instance. The template includes statements about the image and configuration of the guest instance, as well as a link pointing to the host's attestation. This parameterized primitive can be used to attest a TapCon VM, a Spark container, or a Spark application. As a result, a valid attestation chain in Latte is linked for efficient retrieval.

The cloud metadata service stores only authenticated statements. It authenticates each client to its network address, and includes the issuer's id into each statement.

We further develop guard scripts to perform compliance checks for access control. We implement guards as primitives in another SAFE trust script. They instruct

SAFE to retrieve and cache attestations and to perform logic inference against a closure of linked logic sets. A guard specifies a target query, references to local policy rules, and a reference to an attestation DAG used as the inference context. Exemplary guards include those that check the identity of a program running in an instance and the properties of the program. Compliance checks for attestation based access control are end to end: the guard checks program properties against access policies (e.g., ACLs) attached to data objects for attestation-based access control.

4.4 Certified Build Service

The Root VM image contains basic OS services but no applications. TapCon builds application container images from source repositories with its *secure build service*. It allows a user to build new VM images and container images in a known and trusted environment. It takes a Git URL with a Dockerfile (a manifest desecribing how to build the container), then uses the Dockerfile to generate a Docker image. Once complete, the build service publishes the image to IaaS storage (such as a virtual block device or blob storage), and posts statements about the image to the metadata service.

5 Application Use Cases

We describe how Latte is used to strengthen security in five use cases: a package and container building service, data storage, joint data analytics, machine learning, and a multi-tier web service.

5.1 Building Service

We extended the certified build service to also build packages from a `makefile`. The service publishes a binary it builds and provides public access to it via a specified path. At the same time, it posts statements about what was built to the metadata service.

We also added extensions to automatically apply static analysis tools during the build process to scan for bugs and vulnerabilities. The building service uploads the source package to SWAMP [16], which runs the clang static analyzer and gcc checker [6]. It then issues statements about the number of common weakness enumeration bugs (CWEs) found in the package. These statements are linked to attestations of the instances that run the code and can be retrieved by a client to verify if a service meets its security requirements.

5.2 Data Storage

We extended two existing data stores to add attestation-based access control to their existing authentication and access control methods. This addition allows data owners to ensure that data can only be accessed from an approved software stack. We rely on the existing authentication mechanisms of the underlying data store to identify users and apply normal access controls; Latte's access control is an additional layer of control.

MySQL. We implemented attestation-based access control at the granularity of database connections using MySQL-router [10]. This service acts as a transparent proxy for a backend database. We extended MySQL-router to store aaccessor IDs and corresponding ACLs that specify what software versions are allowed to connect. On each connection, the router invokes a guard to authorize based on the attestation statements about the client. Only if the attestations prove that the client's software stack is permitted does the router proxy connect to the MySQL backend. This design re-uses MySQL's existing password-based user authentication mechanism.

HDFS. We extend HDFS to associate each file path with a local policy file. This policy file contains a list of ACLs, specifying required code properties, e.g., version of source code, that the client program must have to gain the access. On creation of a file, HDFS posts its code property-based ACLs to the metadata service. When a client requests to access using the file path (e.g., traversal, read write), HDFS invokes a guard to check if the client code, authenticated by network address, is permitted to have access. This allows different software to access different sets of files. This design demonstrates how to add attestation-based access control to HDFS.

5.3 Data analytics

We extend Apache Spark [20] to post attestations and sandbox applications. A cluster of modified Spark runs as a platform that provides data analytics service to multiple clients. Spark provides isolation among clients' apps: tasks of each app run on a separate set of executors. We extend Spark to attest to the code an application runs, allowing data access to be granted to a single computation. This is useful for solving the data sharing problem in joint data mining: clients can store data in HDFS or MySQL protected ACLs that limit access only to desired analysis programs.

Cluster attestation. Both Spark master and workers run in Docker containers. When a worker joins a cluster, the Spark master posts attestation statements about the cluster membership of this worker with a link referring to the master's attestation. A worker's attestation by its host TapCon VM further links to the worker's membership attestation. This allows a storage service to verify that a client is part of a valid Spark cluster, and that all members of the cluster run the correct code: the guard verifies the worker is correct, and checks for membership statements from the master, and then also checks that the master is correct.

Web frontend. The Spark master also attests the web front end, through which one can upload code, submit

jobs, and upload and download data sets. The frontend authenticates clients and harvests each client identity as a public key hash. It then binds the ID to job-related requests so that Spark workers and HDFS name nodes can use the client identity for access and authorization checks.

Program attestation. Each Spark worker publishes attestations about the programs it runs. These programs are submitted as jar archives. Before job submission, the builder of the jar, e.g., the building service, issues statements about the program, using the hash of the jar as ID, and its source information as properties. When a job is submitted to Spark, it launches an application, issues attestations about the new instance principal and the ID of the program running on it. Thus, the source information is naturally linked to the attestation of a running application.

We deploy Spark within containers, where each Spark container has code for master, worker and web frontend service, and can launch the service by different commands. We run the containers within VMs, and use port restrictions to specify which ports are used by each running Spark program. This allows storage services to identify which Spark program generated a request from the IP address and port of the source.

Application sandboxing. Spark provides a rich programming interface for applications to operate on data sets, e.g., RDD operators to incorporate user-defined functions. This has potential to violate the security and privacy of client data. Thus, with our Spark extension a worker sandboxes application code by rejecting tasks containing sensitive data operations, e.g., transmitting records over a network connection created by the app. The Spark master further attests to the membership of each worker. An authorizer verifies if each cluster node runs a correct Spark version to assure that applications are sandboxed across the entire cluster before releasing data.

5.4 Machine learning

We further extend Spark to use it as a component of a machine learning system, Prediction IO (PIO) [12]. This use case of Latte demonstrates how attestation can be used in multi-component applications to restrict data flow and provide assurance in how data is used. Prediction IO contains several pieces: a front end that accepts users input, a data storage for training data and model, and a learning engine that compiles and submits Spark applications to a Spark cluster. The storage can be either MySQL or HBASE; in our experiments we use MySQL.

The system architecture of the trusted Prediction IO service in Latte is shown in Figure 4. A front end container includes all components of Prediction IO to receive user's data and command. On launch of such container, an learning engine is created with a secret access key, which is shared among clients of this container to send private data in. These clients can also invoke training process, or send queries to the container for prediction result. To train a model, the container talks to a SafeSpark cluster to submit submit analytic jobs with database credential, which will then access Prediction IO controlled secured MySQL database. The trained model is written back to MySQL, and will be used by Prediction IO to serve future requests.

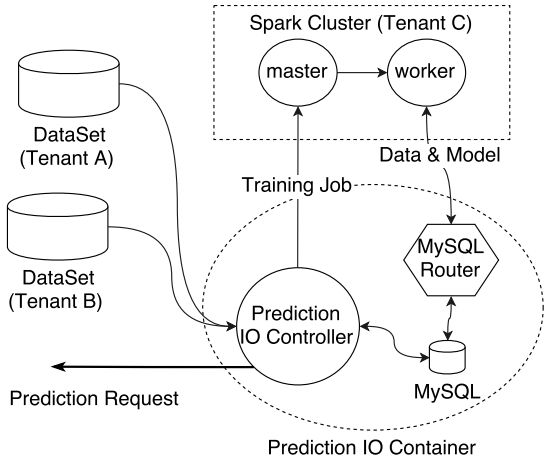

Figure 4: **Prediction IO data flow. A Prediction IO container, which can belong to any tenant, is running as a controller. Tenant A and Tenant B can feed in training dataset, which will be stored in a container local MySQL database. The Prediction IO submit training job to Spark cluster and will access these data from Spark worker. After training is finished, a model is write back and future prediction can be served.**

To trust this setup in a join data mining scenario, a client needs to verify a few conditions before pushing in training data: (i) a correct version of the Prediction IO container that implements only the approved command interface, (ii) the Prediction IO container verifies a correct Spark cluster is running, and (iii) Prediction IO container restricts database access so that only a container and a Spark job that runs trusted learning engines can directly access the training data. To do this, clients query the metadata service about the Prediction IO's code identity (condition (i)), from which he/she can infer that Prediction IO will enforce condition (ii) and (iii).

5.5 Web Service

As a final example, we set up a Docker container based multi-tier web service. For each container, they will post the environmental variables configured by Docker, and use these variables to start the actual components. All such variables are posted into the metadata service by starting script of the container. These self-claims could be trusted as long as the underlying source code identity is correctly verified, since we can inspect the startup script and make sure it does so. We aim to use these properties for high assurance in deployment.

The web site administrator can enforce correct installations by ensuring each component only talks to correct implementations of the other components. In particular, the web front end can verify that the database is correctly installed and configured. For example, the client of a web service could verify that the webserver is patched against known vulnerabilities such as Heartbleed.

We set up Mediawiki [9] as an attestable web service. There are three tiers architecture: frontend load balancer built with Nginx TCP load balancer module [35], application server using Mediawiki container, and backend attested version of MySQL database, guarded by MySQL-router.

This installation is bootstrapped by first launching MySQL as a separate container, with a trust script only allowing access to Mediawiki's image. When Mediawiki launches, it posts the IP address and port of the MySQL database it's configured to use. Finally, the load balancer launches and posts the IP addresses and ports of all the web servers to balances load across.

A client of the web site sees only the load-balancer's address, and can verify through attestations it is correct. It can then enumerate the web servers linked to the load balancer and verify they are all correctly instantiated, and finally, enumerate databases connected to the web servers and similarly verify their correctness. We note that most clients would not bother doing such verification, but administrative users may, for example to verify correctness before publishing data.

6 Evaluation

We measured the overhead of our Latte prototype for launching containers and the performance of attestation-based access control. As most attestation checking only happens when establishing a new connection, overhead should be generally low for long-running applications. We evaluate on a 7-node cluster on CloudLab [2]. Each node has 16 Intel E5-2630 cores, 128GB memory, and two 10GbE NICs. The cluster runs OpenStack and Docker with modifications for Latte. For OpenStack setup, we use 4 compute nodes, one for storage, one for network gateway, and one for cloud controller.

6.1 Micro Benchmarks

Metadata service and checking service. To benchmark the metadata service, we post attestations at different layers, using the uniform post primitive but passing in suitable values to attest a particular guest. With the metadata service, 95% attestation post operations can be finished within 8 ms. We also synthesize workload to comprehensively evaluate our checking service. We observed that 95% attestation-based access authorization queries can be resolved within 5ms.

Network performance. network performance is not affected by kernel changes at connection time. We use qperf [13] to measure the latency and bandwidth of changed kernel and unmodified kernel from container network to host network. Both bandwidth is stable at 9.6Gbps under multiple measurement. Latency wise, the modified kernel runs 255us (+-22us), while the unmodified runs 237us (+-27us), tail latency shows similar trend. This result comes mainly from extra iptable rules to enforcing container network usage and additional connect time checking.

Principal Boot time. there will be a slight overhead added to application launching. We specifically measured the container boot overhead with principal creation for a simple process image, there is a 10ms overhead in average, and is 2% for the whole container to setup.

Similar results can be confirmed with Prediction IO container, which takes 3 second to boot and configure, within which 40ms is spent on configuring attestation based access control and principal.

6.2 Application Workloads

We set up ive application usage scenarios from Section 5 to evaluate Latte.

Storages. We query Mysql booted in a Docker container with a trivial select statement, and measure the latency changes. The container runs in a VM with 8 vcpu and 32 GB memory. For HDFS we evaluate simply HDFS read performance We run HDFS in 4 VMs with same configurations. There are 4 datanodes and 1 namenode. We mesure the read performance on a 128MB file repeatively, and compare the case for unmodified HDFS, and modified HDFS w/wo access policy set on the target file. In fact, w/wo policy incurs only 1% difference so we show only the case w/ policy.

Spark. We use a five virtual-machine (VM) cluster, with each VM having 8 virtual CPUs and 32GB memory. One VM runs the Spark master and the remainder run workers, with one container per VM. We also run the same HDFS configuration described above. The client workload is Pagerank from the Intel HiBench suite [8], with input scale "large". The client uploads data into HDFS. For modified HDFS, client will then specify access policy to restrict code identity. When data is prepared, Spark jobs are launched to compute the ranks. In this process, the client verifies the code correctnes of both HDFS and the Spark cluster, and HDFS verifies the correctness of Spark. We compare completion time of SafeSpark with modified HDFS against unmodified Spark and HDFS.

Prediction IO (PIO). We test Prediction IO with a one-million song dataset [23]. For training we extracts 20000 data points with 90 features. The total size of the

Workload	Latte	baseline
Mysql	20.6ms (+-1.2ms)	9.6ms (+-0.3ms)
HDFS	2.3sec (+-0.1sec)	2.2sec (+-0.1sec)
Page Rank	65.8sec (+-1.0sec)	64.4sec (+-1.5sec)
PIO Training	38.5sec (+- 0.4sec)	38.3sec (+-0.4sec)
PIO Prediction	392.2sec (+-2.2sec)	391.9sec (+-1.3sec)
Web Checking	122.6ms (+-1.6ms)	N/A
Web Accessing	22.3sec (+-0.4sec)	18.4sec (+-0.2sec)

Table 4: Workload execution time. Web Checking means integrity check, and fetching is completion time of web pages access.

features is about 16MB. We then send 100,000 prediction queries using the generated model.

Web service. We launch a cluster of containers with one MySQL database, two nginx load balancers, and five Mediawiki servers. We measure the time to verify attestations for the entire cluster. We also measured the completion time to fetch 1000 wiki pages with 10 parallel clients with hot cache.

The completion times for all workloads are in Table 4. The difference is that there will be attestation during Spark worker tries for data request. Mysql latency is almost doubled in trivial test, but this does not affect Prediction IO performance. On the other hand, the web fetching case has 17% difference is because the wiki server is initiating frequent Mysql connection, which incurs quite a bit attestation query to the checking service. It could be improved if long connection is used.

7 Related Work

Secure containers in TapCon are complementary to SCONE [21], which runs container code in SGX enclaves (following Haven [22]). The TapCon alternative avoids the performance costs and limitations of enclaves. More generally, Latte shows how to support third-party attestations with logical trust rules that extend checks to multiple layers. Latte could use SCONE to ground attestation chains in secure hardware enclaves, although we have not yet explored this possibility. However, enclaves still impose a considerable performance cost, as the SCONE paper shows, and this cost is avoidable when the cloud provider is trusted, as is common in deployments today.

Our use of logical trust to reason about attested software is similar to *logical attestation* in Nexus [39], which provides a rich framework for logical attestation within a single host [39]. Latte extends logical attestation to a cloud setting with distributed applications, and it uses Datalog as a trust language, which is simple, standard, and fast. In contrast, Nexus introduces a powerful authorization language that is intractable in the general case.

Terra [28] supports layered attestation rooted in hardware where each software layer (hypervisor, operating system, etc.) attests to the layer above and endorses an encryption key for remote communication. Latte applies layered attestation in a trusted cloud setting, using network addresses as secure identifiers. This approach is interoperable with existing applications and layering architectures, with minimal changes to invoke the new APIs for attestation and access control—no new protocols or key management.

The CQSTR secure cloud [43] attests the boot image and configuration for one or more virtual machines running in a cluster. Compared to CQSTR, Latte provides layered attestations allowing multi-tenancy within a virtual machine, container, or even an application, all with different access rights.

Other attestation solutions. BIND [38] offers a fine grained attestation service which attests to binary code region. It uses sandbox mechanism to execute attested code and bind the output with it. Chen et al [26], Sadeghi et al [37] both implemented property-based attestation, which cares more about properties of a given platform, instead of the binary identity. All these works focus on binary code objects; Latte extends these ideas with secure linkages to source code, which facilitates practical *third-party* attestation.

Secure joint data mining. Our motivating examples can plausibly be addressed by cryptographic solutions for verifiable and secret computing. For example, secure multiparty computation can provide guarantees about secrecy and correctness for a computation that reads inputs from multiple parties [36]. Also, using fully homomorphic encryption [29, 42], the data can be safely outsourced to a remote computation without disclosing the secret. These approaches are attractive but impose substantial costs and burdens for cryptography and key management.

8 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, and show how it can be applied to a hierarchy of service managers using Latte. This architecture enables the client of a service to verify that the all the code comprising an application are from a trusted repository built on a trusted platform, so that what is executing exactly matches the desired program.

References

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

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

[3] "debian reproducible build project".

https://wiki.debian.org/ReproducibleBuilds.

[4] Docker authz plugin. https://github.com/twistlock/authz.

[5] Faketime tool. https://launchpad.net/~sweptlaser/+archive/ubuntu/faketime.

[6] Gnu c compiler. https://gcc.gnu.org/.

[7] Heart Bleed Bug. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160.

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

[9] Mediawiki. https://www.mediawiki.org.

[10] Mysql-router. https://dev.mysql.com/doc/mysql-router/2.1/en/.

[11] Openstack. http://www.OpenStack.org/.

[12] Prediction io. http://prediction.io.

[13] Qperf. https://www.openfabrics.org/downloads/qperf/.

[14] "reproducible build project". https://reproducible-builds.org/.

[15] Reproducible tools. https://wiki.debian.org/ReproducibleBuilds/ExperimentalToolchain.

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

[17] I. Amazon Web Services. Aws identity and access management (iam). https://aws.amazon.com/iam/.

[18] I. Amazon Web Services. Aws lambda. https://aws.amazon.com/lambda/.

[19] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative Technology for CPU-based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13, 2013.

[20] Apache Foundation. Spark. https://spark.apache.org/.

[21] 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*. USENIX Association, 2016.

[22] 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. USENIX Association, 2014.

[23] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[24] K. Bhargavan, M. Kohlweiss, A. Pironti, P.-Y. Strub, S. Zanella-Beguelin, and C. Fournet. Proving the TLS Handshake Secure (As It Is). In *Advances in Cryptology – CRYPTO 2014*, pages 235–255, July 2014.

[25] 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.

[26] 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. ACM, 2006.

[27] Docker Inc. Swarm mode overview. https://docs.docker.com/engine/swarm.

[28] 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.

[29] C. Gentry et al. Fully homomorphic encryption using ideal lattices. 2009.

[30] 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 Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.

[31] 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.

[32] 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.

[33] 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.

[34] W. R. Marczak, D. Zook, W. Zhou, M. Aref, and B. T. Loo. Declarative Reconfigurable Trust Management. *Computing Research Repository*, Sept. 2009.

[35] Nginx Software. Nginx. https://www.nginx.com/.

[36] T. Ristenpart and S. Yilek. The power of

proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In *Advances in Cryptology-EUROCRYPT 2007*, pages 228–245. Springer Berlin Heidelberg, 2007.

[37] 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. ACM, 2004.

[38] 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. IEEE, 2005.

[39] 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.

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

[41] L. Torvalds. Git Version Control. `https://git-scm.com/`.

[42] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 24–43. Springer, 2010.

[43] 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. ACM, 2016.