Verifying Cloud Services: Present and Future

Sara Bouchenak
University of Grenoble – LIG
Grenoble, France
sara.bouchenak@imag.fr

Gregory Chockler
IBM Research and
Royal Holloway, University of London
Gregory.Chockler@rhul.ac.uk

Hana Chockler
IBM Research
Haifa, Israel
hanac@il.ibm.com

Gabriela Gheorghe
SnT
University of Luxembourg
gabriela.gheorghe@uni.lu

Nuno Santos
MPI-SWS
Germany
nsantos@mpi-sws.org

Alexander Shraer
Google
Mountain View, CA, USA
shralex@google.com

ABSTRACT

As cloud-based services gain popularity in both private and enterprise domains, cloud consumers are still lacking in tools to verify that these services work as expected. Such tools should consider properties such as functional correctness, service availability, reliability, performance and security guarantees. In this paper we survey existing work in these areas and identify gaps in existing cloud technology in terms of the verification tools provided to users. We also discuss challenges and new research directions that can help bridge these gaps.

1. INTRODUCTION

With trendsetters like Amazon, Microsoft, Google, or Apple, cloud technologies have turned mainstream. Tools and services such as Dropbox, Google Docs and iCloud are widely used by home users. As cloud technology matures, public cloud services are becoming more attractive to enterprise users as well. Interest in the cloud has been shown by players such as critical infrastructure providers, including medical and banking industries, power grid operators, and more. Indeed, the benefits of cloud services, such as flexible and rapid service deployment, cost reduction, and little (if any) administrative overhead, are widely accepted. Yet, very limited tools are currently available for clients to monitor and evaluate the provided cloud services.

It is only reasonable that consumers who pay for a service expect it to be (among other features) available, reliable, secure and careful with their data. But examples abound that this is not always the case: in terms of availability, Amazon Elastic Cloud faced an outage in 2011 when it crashed after starting to create too many new backups of its storage volumes [33]. Many large customers of Amazon such as Reddit and Quora were down for more than one day. Intuit experienced repeated similar outages [48] in 2010 and 2011. No explanation had been provided to customers, who for long could not access their financial data. Authentication and authorization problems are also common: in 2011, DropBox admitted that a bug in their authentication mechanisms had disabled password authentication [37]; hence, for four hours, the accounts of Dropbox's 25 million users could be accessed with any password. Similar authorization issues have affected other cloud service providers, including Google [35]. Recent data loss incidents include the Sidekick disaster [25], involving T-Mobile and Microsoft. A server failure is said to have caused the loss of the data of one million users, who had to wait for two weeks to have their two-week-old data restored. In numerous incidents user data was compromised by hackers. For example, Sony Playstation was compromised in 2011 [78]. Forensic analysis took several days to complete and the breach caused the data of 77 million users to be stolen.

Users could benefit from knowing to what extent a cloud provider delivers the promised service. More concretely, the contract between the cloud and its users should be verifiable (at least to some extent) and the ability to detect failures, without relying solely on the cloud provider's report, can be useful to the users. For example, it may be important to promptly find out that a service does not respect its functional specification; or that it generously shares personal data with the world; or that it is down, underperforms, or if its basic security controls seem to be failing. This information can be especially helpful for critical applications such as medicine or banking and facilitate their process of adopting cloud technologies. In addition, verification tools to check these aspects can help consumers pick and choose a particular cloud provider.

But, other than the claim that a service achieves a certain goal (stores or serves data, computes a function, etc.), what delivery parameters may be of interest to consumers? To answer this question, we identify several areas of concern:

(C1) Trusted software and server identity. Is the service running the right software over the correct set of servers?

(C2) Functional correctness[1]**.** Once the service is running, is it doing what it is supposed to?

(C3) Performance and dependability. How efficient is the service? Is it reliable and available?

(C4) Security. Does the service comply with security policies, if any?

State of the art research has started to tackle these issues individually. However, we feel that cloud users may benefit from understanding in breadth, rather than only in depth, whether they could verify service provisions in the cloud.

[1]Functional properties are specified as properties of individual executions of the system (an execution is an alternating sequence of global system configurations and events, specified by the protocol), and similarly verified over executions. They can be both safety and liveness properties.

Answering concerns such as C1-C4 can raise awareness and lead the way to more tools that empower cloud users.

In this paper, we attempt to identify existing gaps in today's cloud technologies with respect to concerns C1-C4. We discuss recent research advances, and propose directions for future research to help and bridge those gaps. Our survey is related to several others in the area [23, 66]. However, in this case we go beyond a few random examples and provide a first attempt at better systematizing potential client concerns and related solutions.

We consider two main types of cloud customers – *service providers* deploying their software for execution in the cloud (with Platform-as-a-Service or Infrastructure-as-a-Service), and *cloud users*, who use a software or storage service executing in the cloud, be it provided by a third party software provider or by the cloud provider. Next, we briefly introduce the different research areas covered in the remainder of the paper.

Verification of Strong Service Identities.

Today, service providers have no guarantees that the services being delivered to their users match the implementation deployed to the cloud. The risk of cloud mismanagement stemming from cloud administration mistakes or from abuse by other cloud tenants could result in corruption or misconfiguration of the service implementation. Consequently, the service could deviate from the behavior originally intended by the service provider. For example, previous work [74] manipulated the identities of virtual machine images to demonstrate an attack on the consumers of Amazon EC2. In Section 2, we discuss a possible path towards enabling service providers to attest the deployed services and check for compliance with their original service implementation. The idea is to bind a strong *service identity* to the service instances on the cloud such that this unique association is preserved throughout the entire service lifecycle, from deployment to decommissioning. We focus on a promising implementation of this idea based on *Trusted Computing*. Cloud nodes run special software stacks – *trusted software systems* – that can host the service instances in special environments, isolated from both the administrator and other tenants. Cloud nodes are also equipped with commodity trusted computing hardware, which validates the integrity of the software stack upon boot and enables service providers to verify that the nodes are running a trusted software system; if this is the case, service identity is preserved. In Section 2, we introduce this general approach, discuss existing related work, and highlight the main challenges in realizing this vision.

Verifying Functional Properties of Cloud Services.

Users can benefit from gaining assurance that the behavior of a cloud service complies with its advertised functional specification. In Section 3, we propose a new approach allowing the users to verify service integrity in a scalable fashion without relying on either a centralized certification authority or access to the actual implementation code.

Our approach is based on *decomposition* of the verification process into three phases: *test suite generation*, *test suite execution*, and *validation of the results*, where each phase can be performed at a different location to maximize performance and exhaustiveness of the verification process.

Our proposal for implementing the test suite generation is based on *black-box testing* techniques that generate test suites covering all interesting behaviors described by the specification. Since in our framework, the specification is described as a state machine, a test suite would produce inputs to generate all possible traversals of the state machine. Test suite execution is done in the cloud, and the resulting traces are stored in the cloud for future compliance testing. The latter is done on the end-user infrastructure using *sampling* techniques, such as *property testing*. Sampling improves efficiency and scalability of our approach, while guaranteeing specification compliance with high probability.

Verification of Cloud Storage Services.

While generic verification methods such as those we propose in Sections 2 and 3 may, in the future, allow verifying functional properties of cloud services, they have not yet matured. Multiple recent works have tackled specific concerns that arise in the context of cloud storage, and promising techniques have emerged. In Section 4 we survey such desirable storage properties and state of the art verification techniques.

Performance and Dependability Non-Functional Properties Verification.

Verifying non-functional properties like performance, dependability, energy consumption and economical costs of clouds is challenging today due to ad-hoc management in terms of quality-of-service (QoS) and service level agreement (SLA). We believe that a differentiating element between cloud computing environments will be the QoS and the SLA provided by the cloud. In Section 5, we call for the definition of a new cloud model that integrates service levels and SLA into the cloud in a systematic way. The proposed approach aims to combine and guarantee multiple cloud service level objectives in a consistent and flexible way. It also allows to provide better than best-effort cloud QoS through a control-theoretic approach for modeling and controlling SLA-oriented cloud services. We also discuss how to help system designers build SLA-oriented clouds that are controllable by construction, and how to assess cloud service QoS guarantees.

Security-Oriented Non-Functional Properties Verification.

Service providers may request that the deployment of their service in the cloud adheres to certain security constraints. For example, a service provider might ask that their deployed service should only reply to authorized requests coming from the US, between 2 and 6 pm, or that it should never divulge sensitive data to a set of end users, or that it should destroy or backup data at periodic intervals and in a certain way. These behavioral constraints are often independent of the application that is being provided. It is difficult to guarantee adherence to such constraints, because of the dynamic and multi-tenant nature of the cloud environment. For both users and service providers, it can be beneficial to have tools that monitor the high-level system behavior and raise 'alarms' when security policies of this type are violated. Such monitoring tools have not yet matured. Section 6 explains the connected issues and advances in more detail.

In what follows we examine these topics in more detail.

2. VERIFYING STRONG SERVICE IDENTITY

A service provider incurs risks of cloud mismanagement when making use of a cloud provider's infrastructure for hosting services. If the software that the service provider deploys to the cloud is tampered with or replaced for a different version, the service in production could deviate from the intended implementation and distress the service provider and users. The question we address is: How can cloud providers guarantee a strong identity between the software running on the cloud nodes and the service implementation?

2.1 Definitions and Approach

We focus on enforcing the property of *strong service identity* on a cloud platform. If S denotes the service software implementation produced by the service provider and S' an instance of the software service S hosted in the cloud, strong service identity is satisfied if and only if the invariant $S = S'$ holds for the entire lifecycle of S and in all the nodes where S is instantiated. The lifecycle of a service spans the period between its deployment until its decommissioning. Throughout this length of time, the service might be replicated or migrated across various cloud nodes. In *Infrastructure-as-a-Service* (IaaS) the service is deployed as a virtual machine image and instantiated in virtual machines (VMs). In *Platform-as-a-Service* (PaaS) the service is shipped as an application package and instantiated into objects in application containers.

To enforce strong service identity, a cloud platform could provide *trusted containers*. A trusted container hosts the state of a service instance in isolation from other tenants and from the cloud administrator. This protection is enforced throughout the service lifecycle. When migrating or replicating service instances to other nodes, the trusted container verifies that the sink is also a trusted container and transmits any relevant service code and data to the sink over an encrypted channel. The service provider can also verify that the target host offers trusted container protections before deploying the service. As a result, insofar as the service is instantiated in trusted containers, the strong service identity is satisfied.

The implementation of the trusted container semantics on the cloud nodes could be carried out by a privileged software system. A *trusted software system* offers a specific hosting abstraction and is crafted so that neither the administrator nor other tenants have access to service instances' state. Examples of such systems include CloudVisor [86], which leverages nested virtualization to protect the confidentiality and integrity of guest virtual machines in Xen. Other trusted software systems exist, for example, offering isolation at the process granularity [77]. These systems could be used not only to protect the state of the service instances, but also to protect the back-end cloud systems (e.g., database servers). The question then is how can remote parties verify that the cloud nodes execute a trusted software system rather than an insecure OS or hypervisor.

To provide such a validation capability, we leverage commodity Trusted Platform Module [40] (TPMs) chips deployed on the cloud nodes. TPM enables remote attestation of a cloud node. During bootstrap, a cloud node executes a sequence of programs and stores the hashes of these programs in the TPM's internal registers. Since these registers cannot be rewritten unless the machine reboots, their content reveals the bootstrap state of a node and the TPM enables to securely convey the state of these registers to a remote party using an attestation protocol. To prevent man-in-the-middle attacks, the TPM signs the registers' content with the private part of a cryptographic keypair that never leaves the TPM in plaintext. The remote party can then verify the signature and the content of the TPM registers using a public key certificate given by the cloud provider: if the trusted software system boots on the cloud node, its respective hash will show up in the TPM's registers.

By rooting trust in TPMs and on trusted software systems we require that both these components are correct. Under this assumption, strong service identity could be enforced in the presence of powerful adversaries. The TPM can protect the content of its registers from a malicious administrator with privileges to manage the cloud nodes from a remote site: he can reboot the nodes, access their local disks, install arbitrary software, and eavesdrop the network. TPMs, however, cannot defend against physical attacks. We assume that the hardware is protected by complementary mechanisms deployed within the cloud provider's premises.

In summary, by implementing the trusted container abstraction, a cloud platform architecture based on a trusted software system and TPMs deployed on the nodes could enforce the strong software identity. Through the use of attestation, this architecture enables service providers and users to obtain tangible evidence of compliance with the strong software identity property. Next, we examine existing work that materializes some of these concepts in concrete systems.

2.2 Existing Work

We briefly survey the existing work on 1) enforcing strong identity in IaaS, 2) leveraging TPMs in the cloud, and 3) implementing trusted containers on the cloud nodes. To the best of our knowledge, no system today implements strong service identity in PaaS platforms.

Strong software identity in IaaS. In IaaS, services are typically dispatched to the cloud provider in a virtual machine image. Enforcing strong identity, then, requires devising a hardened hypervisor that can offer trusted container semantics at the granularity of VMs. The hardened hypervisor must enforce VM state isolation from the cloud administrator. To ensure confinement of VMs only to cloud nodes running the hardened hypervisor, cloud nodes are attested based on the TPMs located on the nodes locally. To give users and service providers guarantees of service identity (i.e., that the VM image of the VM executing on the nodes is the VM image uploaded by the service provider and instantiated on the cloud) attestation can also be done from outside the cloud. This architecture was first proposed by Santos et al. [70]. To implement the role of the hardened hypervisor, CloudVisor [86] could be used.

Systems for leveraging TPMs in the cloud. Some systems have been developed that, while not offering directly the property of strong software identity, provide a building block for doing so. Schiffman et al. [72] proposed a system that allows for the remote attestation of cloud node's hypervisor and VM image from outside the cloud. A more advanced version of this system is Excalibur [71]. Excalibur prevents performance bottlenecks due to TPM inefficiency and offers an abstraction for sealing data based on policy such that only the nodes that satisfy that policy can unseal

and interpret the data. For example, by sealing a VM image to a policy designating CloudVisor as the trusted hypervisor, the service provider is guaranteed that only the nodes running CloudVisor could instantiate the VM image thereby abiding by the strong identity property. Excalibur can support other software stacks, not only hypervisors, a feature that might be relevant in PaaS. Excalibur also supports restrictions based on the node location, which gives service providers additional control over VM placement.

Systems for implementing trusted containers. While VMs have been the preferable hosting abstraction in the context of cloud computing [86, 20], other systems can offer alternative abstractions that could be more suitable for certain use cases. Systems like Nexus [77] provide trusted container abstractions at the process level. This could be more appropriate for cloud platforms that do not run VMMs on their cloud nodes. Maniatis et al. [56] propose trusted container abstractions as application sandboxes, which can be more suitable for isolation of web applications. Considerable amount of research was also geared toward offering trusted container abstractions while depending on a small trusted computing base so as to reduce the chance of vulnerabilities in the code that could lead to security breaches [86, 77].

2.3 Challenges and Scientific Directions

While the existing work has focused on supporting strong service identity for IaaS and designing specialized building blocks for cloud attestation and trusted container support, a considerable gap exists between what these mechanisms can offer and what is necessary to enforce strong service identity in PaaS. We highlight three main challenges.

High-level PaaS container abstractions. PaaS platforms typically offer its users programming abstractions that enable them to implement service applications with high level languages like Java or Python. The service implementation typically consists of a set of classes which make use of an API defined by the PaaS provider. These classes are then packaged, dispatched to the cloud, and instantiated by the PaaS platform in isolated containers. Containers typically depend on a software stack that includes the OS, a runtime engine (e.g., JVM), libraries, and back-end services (e.g., databases). In existing PaaS platforms, however, containers do not yet offer the property of strong service identity. To enforce this property, one direction is to enhance existing containers according to the trusted container semantics. This task, however, is challenging using the known mechanisms. On the one hand, trusted container abstractions based on VM [86] or process [77] are too low level to be useful for the PaaS users. On the other hand, trusted container abstractions offering application sandboxes [56] depend on a very large trusted computing base (TCB); with this approach it would be necessary to trust the entire PaaS stack therefore incurring TCB bloating. How to provide high-level PaaS abstractions with a small TCB is an open question.

Integration with PaaS back-end. When instantiated in a PaaS container, a service instance will normally make use of additional PaaS back-end services, which include for example databases and transaction monitors. When devising trusted containers for PaaS, it is necessary to account for the fact that the integrity of the service instance hosted by the container could be compromised by a back-end service. In fact, by yielding erroneous results, a back-end service could taint the code or data of a PaaS user's service instance, and introduce corruption that could violate the strong service identity that we wish for. This danger raises several questions: How can PaaS users know if a back-end service is reliable and therefore know if it can be used safely? How to handle the heterogeneity of back-end services, each of them featuring particular capabilities that raise various confidence levels with their users? How to deal with software updates of the back-end services and determine whether updates are secure? What implications will these issues have to the programming model offered to PaaS users?

Distribution and migration of PaaS service instances. In general, the PaaS-hosted services can be expected to be both multi-tiered and clustered. As a result, a service comprises multiple components which can be distributed across several cloud nodes. These components are hosted in independent containers and communicate among themselves over secure channels. It is also common that, for resource management reasons, a PaaS platform might migrate components around across different hosting containers, e.g., for balancing load. Components might also need to be instantiated in or eliminated from containers in order to accommodate the elastic variations in the service demand. To account for all these scenarios when implementing the trusted container semantics, it is then necessary to always attest a hosting cloud node before creating a component instance and to provide that the distributed component instances can authenticate and communicate securely. Existing systems that support attestation in the cloud have been used only in the context of IaaS for attesting hypervisors and VMs [72, 71]. In IaaS, however, the number of VMs that need attestation is significantly smaller than a potentially large number of PaaS service components. It is unclear if existing systems could withstand such a large attestation demand without incurring scalability bottlenecks.

3. VERIFYING FUNCTIONAL PROPERTIES OF CLOUD SERVICES

The techniques described in the previous section allow the PaaS services to be associated with a *strong identity*, which is being preserved throughout the entire software lifetime withstanding administration mistakes, and tampering attempts. In this section, we focus on a complementary question, namely, given a uniquely identified service instance deployed and running on the trusted PaaS platform, how can we efficiently verify that its behavior complies with the functional properties advertised by its provider?

Our approach to verifying functional properties of the PaaS services is based on the software testing paradigm. Conceptually, the software testing process can be viewed as consisting of the following three phases (which can be interleaved to improve performance):

Test suite generation: the specification and tested software are analyzed to extract effective *test cases* which are then assembled into a *test suite*.

Test suite execution: the software is subjected to the test suite produced at the previous stage.

Result validation: the traces generated by running the test suite are compared against those prescribed by

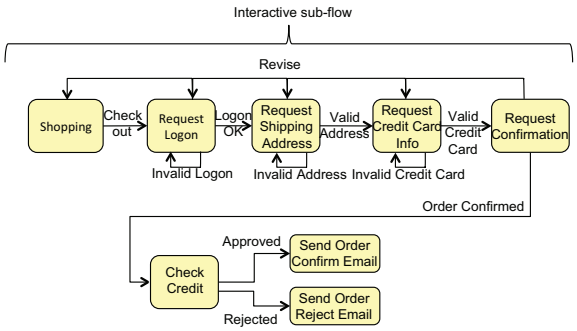

Figure 1: Specification of the Checkout Flow of an On-Line Shopping Site. The specification is modelled as a finite-state automaton consisting of 7 states, 5 of which belong to the interactive portion of the checkout process. Each of these 5 states allows the customer to return to any one of the preceding states to revise the data entered at that state. In addition, another 3 states in the interactive group have self-cycles allowing the customer to correct errors in the supplied information. The total number of cycles in the automaton graph is therefore, equal 17, and grows quadratically with the number of states.

the specification, producing "pass" or "fail" outputs for each compliant and non-compliant trace, respectively.

In order to make the above process amenable for testing PaaS services hosted in the cloud, the following challenges must be addressed.

First, since the cloud software is typically developed and distributed by a third party Software-as-a-Service (SaaS) provider, the service implementation code cannot be assumed to be available to the end users. This precludes the test suite generator from using *white-box* testing techniques (such as *symbolic execution* [47, 24, 28]), which utilize the knowledge of the code structure to achieve high quality coverage of possible execution paths. In Section 3.2, we discuss alternative approaches to implementing the test suite generator, and propose several solutions based on *black-box* testing.

Second, the cloud-based services are typically *interactive* (see Figure 1): i.e., they are being driven by on-line user inputs (e.g., supplied through a web-based interface), which are forwarded to the remote service implementation via an RPC-style protocol (such as, e.g., REST [38], or SOAP [2]). Consequently, executing the service test suite on the user premises might result in high communication costs, and slow down the entire testing process. Instead, the cloud provider must offer support for executing the test suite on the cloud infrastructure while minimizing the interaction with the user to the largest possible extent. The users must, however be offered tools to efficiently validate the test execution results to guard against the possibility of them being faked by a potentially dishonest cloud provider.

Third, the service logic can be fairly complex as it must be able to accommodate a wide-range of on-line interaction scenarios such as, e.g., undoing the effects of previously executed steps of an on-line transaction (e.g., resulting from the user pressing the "back" button in the browser), or timeouts following long periods of inactivity. As a result, even a

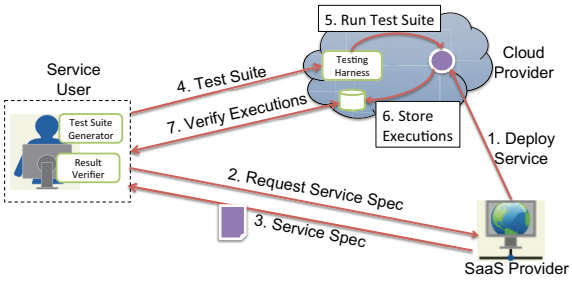

Figure 2: Verification Framework for Services in a Cloud.

service with a small number of interaction steps may end up exhibiting large numbers of acceptable behaviors resulting from repeated traversals through the interaction workflow cycles (see Figure 1). Exhaustive testing of all the resulting behaviors may end up producing large volumes of lengthy output traces whose validation may be too costly to conduct on a less powerful end user infrastructure.

To address the above challenges, we propose a new distributed testing framework enabling an efficient verification of services hosted on a remote cloud. Below, we discuss the framework architecture, and some of the challenges associated with its implementation.

3.1 Testing Framework Architecture

The architecture of our testing framework is depicted in Figure 2. Unlike the existing testing solutions, in our framework, the test suite execution and result validation phases are disjoint from each other, with the former being assigned to the *Testing Harness* component hosted in the cloud, and the latter being executed by *Result Verifier* installed on the user premises.

The service implementation is provided by the *Software-as-a-Service (SaaS)* provider, which is also responsible for advertising its specification. The user inspects the advertised specifications to select the service, whose specification is the closest match to the user requirements. To streamline the service selection process, the specification must be expressed in a standardized specification language, such as, e.g., Web Service Definition Language (WSDL) [1]. Here, we omit the details of the service specification framework, which is the subject of future work.

Next, the specification is analyzed by *Test Suite Generator* to produce a test suite using the *black-box* testing techniques [65] (Section 3.2). The resulting test suite is then submitted to Testing Harness, which deploys the service instance on the cloud-based execution platform, subjects the deployed instance to the submitted test suite, and stores the results on the cloud storage facilities. The Result Verifier can then validate the execution results using the techniques described in Section 3.4.

In the following sections, we discuss approaches to implementing Test Suite Generator, Testing Harness, and Result Verifier in more detail.

3.2 Test Suite Generator

A simple way to create a black-box test suite is to generate a collection of random sequences consisting of the input invocations as defined by the service API. Although this technique can be highly effective in finding bugs in real sys-

tems, it does not guarantee much in terms of the quality of coverage of the service specification.

In contrast, in a more sophisticated black-box methodology, known as *specification-based* or *model-based* testing [67], the test suite is derived from the service specification, modelled as a *state machine*. To guarantee exhaustiveness, the test suite must include a test case for each possible traversal through the specification automaton. Although the test suite constructed in this fashion does not necessarily check implementation-specific details, it provides an assurance that all observable behaviors of the service will be exercised. The test suite composition can be further adjusted to achieve a desired balance between the path coverage and performance, e.g., by excluding test cases exercising less interesting behaviors.

Note that the standard service API might not always be sufficient to exhaustively exercise all the behaviors prescribed by the service model. For example, the test cases necessary for exhaustive testing the credit check portion of the Checkout workflow in Figure 1 will be impossible to generate, using the service's standard API, without a priori knowledge of the real customer credit data.

To address this problem we could require the software provider to expose a special *testing* API that augments the standard service API with calls instrumented for the testing purposes (such as, e.g., those simulating requests from customers with low and high credit scores in the example above). Note, however, that supporting testing APIs in the cloud settings requires cooperation on behalf of the underlying cloud platform to ensure that the testing inputs are not activated during the normal service operation.

Generalizing this approach, of using a public API and a special testing API in order to generate an exhaustive testing suite without compromising security, into a complete solution applicable to realistic services is an interesting research problem, which we intend to pursue in the future.

3.3 Testing Harness

Testing Harness (TH) is responsible for executing the test suite submitted by the user on an instance of the service of interest. One important aspect that must be addressed by the TH implementation is the degree of isolation of the tested service instance from other applications concurrently running on the cloud. In particular, the functional correctness of the implementation code is best tested "in-vitro", that is, when its instance deployed in a fully dedicated runtime environment. To achieve this degree of isolation, the underlying trusted PaaS platform (see Section 2) must expose the necessary hooks which can be leveraged by TH to create an execution environment with well-defined isolation properties.

One limitation of the "in-vitro" testing is that it cannot guarantee that the functional properties, which passed validation when tested in isolation, will continue to hold when the service is deployed in a production environment that could be shared by a large number of other cloud tenants. For example, when not adequately protected against unauthorized accesses on behalf of other co-hosted services, the service might be compromised to exhibit a behavior that arbitrarily deviates from its specification.

To address this problem, TH must offer "in-vivo" testing [27] capabilities that will allow the service instance to be deployed and tested on a simulated or real multi-tenant runtime. In addition, the TH and underlying PaaS runtime must offer hooks that will monitor and log accesses to shared multi-tenant resources. Building such an in-vivo testing environment along with automating generation, testing, and verification of multi-tenancy related properties is an interesting research direction to pursue in the future.

3.4 Result Verifier

A straightforward implementation of the result validation phase would be to execute the specification automaton on each trace produced at the test suite execution phase. This approach may, however, be too expensive for analyzing long traces, such as those resulting from repeated traversals through the specification automaton cycles (see above). To validate such traces more efficiently, we propose to utilize a probabilistic technique known as *combinatorial property testing* [9, 29].

Roughly, this technique is based on the observation that a compliant trace, whose length exceeds the size of the specification automaton (in terms of the number of states), must visit the same state more than once. On the other hand, a trace that is too far from being compliant must contain enough states that do not fit any possible traversal on any cycle of the automaton. Hence, in order to verify the trace compliance with high probability, it is enough to check whether a *sample* of the trace, of size that depends on the number of cycles in the automaton graph, fits into the pattern of traversing cycles. A property testing algorithm then proceeds by sampling short (constant length) segments of the trace and checks whether they fit into some cyclic path on the automaton. Note that in this algorithm, we assume that the input trace is much longer than the longest cycle-free path in the automaton. This is because short traces can be verified exhaustively as described above.

In order to illustrate the advantages of this technique, consider a linear workflow W consisting of n states such that for each state s in W, there is an edge pointing back to a state s' such that s' is preceding s in W (see, e.g., the interactive sub-flow of the checkout workflow in Figure 1). The number of cycles c in W is then roughly on the order of n^2, with the average cycle length being $n/2$. Consequently, the average length of the trace τ produced by executing a test case exercising each cycle at least $k > 0$ times will be at least $kcn/2$, which is also equal to the complexity of the exhaustive compliance check of τ. In contrast, with the property testing-based compliance check, the complexity depends only on c, resulting in a significant speedup compared to the exhaustive check, for large values of k. For example, the complexity of the exhaustive compliance analysis for the trace resulting from one time traversal of each cycle of the interactive part of the checkout flow in Figure 1 will have traverse 47 states as compared to just about 13, which we expect to be the number of states, required for the property testing based analysis in practice.

4. VERIFYING PROPERTIES OF CLOUD STORAGE

Users increasingly rely on the cloud for storage, instantly uploading their photos, documents, scheduled system backups and more. In this section, we explore some of the properties expected by users from a cloud storage service and survey recent work on the verification of these properties.

4.1 Protecting Against a Byzantine Provider

We start by describing properties for which the known verification methods can overcome any adversarial cloud provider, even a fully malicious one.

Integrity. One of the basic properties expected from a storage system is data integrity. Users must be confident that their data is not altered while being stored or transferred to and from the storage service. A simple way to guarantee this is to use error detecting (or error correcting) codes. To protect against intentional tempering of the data, a client may use a cryptographic hash function and separately maintain the key. For large volume of data, hash-trees [61] are commonly used to verify data integrity without recomputing a hash of the entire data for the purpose of verification. The leaves of a hash-tree are hashes of data blocks, whereas its internal nodes are hashes of their children in the tree. A user is then able to verify any data block by storing only the root hash of the tree and performing a logarithmic number of cryptographic hash operations. When multiple users share data using a remote storage service, digital signatures allow the clients to verify data integrity.

Consistency. Although these methods guarantee that the storage will not be able to corrupt or forge the data, it does not prevent a storage service from simply hiding updates performed by one client from the others, or showing updates to clients in different orders. In fact, this would be impossible to detect without additional trust assumptions (such as TPM) or alternatively the clients being able to jointly audit the server's responses. Several solutions using trusted components were proposed [31, 84], guaranteeing strong consistency (i.e., linearizability [42]) even if the service is malicious. A different approach, not assuming any trusted components, was pioneered by Mazières and Shasha [58, 51], introducing untrusted storage protocols and the notion of fork-consistency. Intuitively, traditional strong consistency guarantees that all clients have the same view of the execution history. On the other hand, fork-consistency guarantees that client views form a tree, where forks in the tree are caused by a faulty server hiding operations of one client from another. To date, this is the strongest known consistency notion that can be achieved with a possibly Byzantine remote storage server where no trusted components are assumed and when the clients do not communicate with one another (once clients can communicate directly, they are able detect that their views were forked by the server). Multiple systems were based on this idea, starting with SUNDR [51], a network file system designed to work with a remote and potentially Byzantine server. Cachin et al. [21] implement an SVN system hosted on a potentially Byzantine server. In FAUST [22], authors study fork-consistency more formally, including a proof that guaranteeing this notion comes with a price on service availability, even when the server is correct, and propose a new consistency notion (weak-fork linearizability) that overcomes this limitation. Venus [76], a verification system built with Amazon S3, uses a weak-fork linearizable protocol as a building block but provides more traditional consistency semantics to its clients. When the server is correct, weak-fork linearizability allows Venus to guarantee a strong notion of liveness (i.e., service availability), where clients are not affected by failures of other clients. Venus uses direct automated emails among the clients to uphold strong consistency semantics and to provide eventual detection of storage failures. Feldman et al. introduced SPORC [36], a system which likewise guarantees a variation of fork-consistency, but for the first time allows not only to detect storage faults but also to recover from them by leveraging the conflict resolution mechanism of Operational Transformation. Finally, we note that a similar consistency notion [63] was recently used in a non-Byzantine setting to model consistency in the context of mobile clients performing disconnected operations [30], suggesting a yet to be explored connection between untrusted storage and disconnected operations or, more generally, with the traditional model of message passing with omission faults.

Similarly to storage failure detection using direct communication among clients, if a global trace of client operations and storage responses is available, many inconsistencies can be easily detected [11, 85, 81].

Finally, systems such as Intercloud Storage [13] and DepSky [15] replicate data over multiple clouds in order to mitigate integrity or consistency violations and potential unavailability caused by a provider failure.

Retrievability. How can clients assure that their data is still stored somewhere in the cloud and not lost by a provider trying to cut storage costs? As the amount of uploaded information grows, it is often infeasible for clients to check data availability by periodically downloading all the data. This challenge was addressed in the form of new verification schemes: Proofs of Retrievability (PORs) [46] and Proofs of Data Possession (PDP) [12]. These protocols guarantee with high probability that the cloud is in possession of the data using challenges submitted by the client. The basic idea is that a client submits requests for a small sample of data blocks, and verifies server responses (using small additional information encoded in each block or by asking for special blocks whose value is known in advance to the client). Recently, these schemes were generalized and improved, and prototype systems have been implemented [75, 18, 17]. This line of work has also lead to the development of schemes for verification of other properties, as we describe next.

4.2 Protecting Against an Economically Rational Cloud Provider

In what follows, the verification methods assume an *economically rational* adversary. Such cloud provider may cheat but will not do so if it requires spending more money or other resources compared to correct behavior.

Confidentiality. To prevent information leakage and provide data confidentiality, it is usually expected that stored data is encrypted. Clients can encrypt the information with their own keys before storing it to the cloud. However, this is often not desired as access to the unencrypted data allows the provider to offer a richer set of functionality, beyond storage, such as searching the data or sharing it with other authorized users. Instead, the provider is usually entrusted with encrypting the data. Recent incidents have shown that providers do not always uphold this expectation [10]. Authors in [80] have recently proposed a scheme to probabilistically ensure that the provider indeed stores the data in an encrypted form. The main idea is to ensure that the data is stored in the desired, encoded format G by encapsulating G into another format H, which the provider will store instead of G, and such that H has several desired properties. First, it should be easy for the provider to convert H back into G. Second, it should be difficult for the provider to cheat by computing some part of H on-the-fly, in order to respond to

a client's query, without processing the entire data in format G. And finally, there should be a certain lower bound on the time required to translate G into its encapsulation H, using assumptions on a constrained resource, such as the computational power of the provider, physical storage access times, network latencies, etc. The client can then challenge the cloud provider at a random time to produce random chunks of H, and require that the cloud provider does so within a time period τ. Timely correct response proves with high probability that the provider is indeed storing H and is not computing H on the fly. Obviously, this scheme does not prevent the provider to store another, unencrypted, copy of the data (incurring double the storage). Instead, it provides a strong negative incentive to do so, and hence assumes an economically rational provider.

Redundancy. Storage providers guarantee a certain level of reliability through data replication. For example, Amazon S3 promises to sustain the concurrent loss of data in two facilities whereas a cheaper offering from Amazon called the Reduced Redundancy Storage (RRS) guarantees to sustain the loss of a single storage facility. How can the client make sure that the offered level of redundancy is in fact being provided? Authors in [19] propose a scheme where, similarly to the ideas described above, random challenges for data are submitted to the storage provider and timely responses are expected. In the proposed scheme, the client and cloud provider agree upon the following: (a) an encoding G of the data with an erasure code that tolerates a certain fraction of block losses, and (b) a mapping of the blocks of G onto c drives, such that the data is spread evenly over the drives. The client then submits queries, each requesting c randomly selected blocks of data, such that each block is expected to reside on a different physical drive. The main idea is that the mechanics of commercial hard-disk drives guarantee a certain lower bound on the time it takes to retrieve the data, which for small data blocks is dominated by disk seek time. The scheme is built such that with high probability responding to the query within the expected time τ would in fact require c drives accessed concurrently. This algorithm works for a class of adversaries the authors call *cheap and lazy*, namely - they would like to decrease the cost of storage by storing less replicas, but would not take the effort of changing the data or cheating in other forms. The scheme is designed for hard-disk drives (and not, e.g., SSDs) and is not resilient to fully malicious adversaries, which, for example, may store the data in an encrypted form on c drives as required, but then store the encryption key in volatile memory or on a single drive.

Location. One of the biggest concerns users have when using a cloud storage service is that once in the cloud, the user is no longer sure where her data is physically located. Often, for some types of data, users and especially companies and organizations that maintain private user data, are bound by laws and regulations to store their data within a particular geographical region or country borders. To address this, storage location is frequently an integral part of cloud storage SLAs. Location is also important for disaster tolerance – the provider may promise not only to replicate the data but also that the replicas reside in disperse datacenters or geographical locations. How can a client then verify the actual location of her data in the cloud? Obviously, it is very difficult, if not impossible, to ensure that a storage provider does not store a copy of the data outside of the allowed geographical area. Thus, such verification can only work assuming a weaker adversarial model, such as an economically rational provider in the Proof Of Location (PoL) verification scheme recently proposed by Watson et al. [82]. Specifically, they assume that file replication (copy) is only performed by the service provider and for the purpose of providing guaranteed reliability for which the user is charged. Thus, the provider may try to cheat by storing a copy of the data in a different (perhaps cheaper) location, but only if this is done instead (and not in addition to) storing the data in the correct (promised) location. In [82], the cloud and client agree on the list of storage replicas and their locations. The client then uses a combination of an Internet Geolocation system, that can determine the location of a server using network latencies, with a PoR scheme that can prove that this server actually possesses the data. More specifically, the scheme uses a number of trusted auxiliary servers as *landmarks*, whose location is known and that can send challenges to the storage servers claiming to hold the data, in order to verify their location. A novel PoR protocol introduced in [82] allows the client to encode the data once, after which the server re-codes it multiple times and stores a different encoding of the data on the different replicas. Storing slightly different encodings of the data on different servers prevents the servers from colluding, where a server can claim to have the data, while in fact fetching it from another server on-demand to answer the verification query.

5. VERIFYING PERFORMANCE AND DEPENDABILITY PROPERTIES

5.1 Background

Non-functional properties of cloud services represent different aspects of quality-of-service (QoS), such as performance, dependability, security, etc., and are quantified with different metrics. Performance metrics include service request *latency* which is the necessary time to respond to a service requested by a client, and service *throughput* which is the amount of requests processed by the service per unit of time. Dependability metrics include service availability and service reliability [49]. Availability may be measured as service *abandon rate*, that is the ratio of, on the one hand, the time the cloud service is capable of returning successful responses to the clients, and on the other hand, the total time; cloud service availability is measured during a period of time, usually a year or a month. Availability may also be represented by service *use rate* that is the ratio of time a cloud service is used to the total time. Cloud service reliability may be measured as the ratio of successful service client requests to the total number of requests, during a period of time. Reliability may also be quantified as *mean time between failures (MTBF)* which is the predicted elapsed time between inherent failures of the service, or *mean time to recover (MTTR)* which is is the average time that a service takes to recover from a failure. Other metrics may be considered to render cloud service costs, such as *energetic cost* that reflects the energy footprint of a service, or the *financial cost* of using a cloud service.

Thus, a QoS metrics is a means to quantify the service level with regard to a QoS aspect. One might want a service level to attain a given objective, that is the *Service Level Objective (SLO)*. A SLO has usually one of the following

forms: provide a QoS metrics with a value higher/lower than a given threshold, maximize/minimize the QoS metrics, etc. Therefore, a *Service Level Agreement (SLA)* is a combination of SLOs to meet and is negotiated between two parties, the cloud service provider and its customer. A simple example of SLA is the following: "*99.5% of requests to cloud services should be processed, within 2 seconds, and with a minimal financial cost*". This SLA includes three SLOs respectively related to service availability, performance and financial cost.

5.2 Related Work

The control of services to guarantee the SLA is a critical requirement for successful performance and dependability management of services [53, 57, 60]. Much related work has been done in the area of system QoS management, an interesting survey is provided in [41]. In the context of Cloud Computing, existing public cloud services provide very few guarantees in terms of performance and dependability [14]. In the following, we review some of the existing public cloud services regarding their levels of performance and dependability. Amazon EC2 compute service offers a service availability of 99.95% [3]. However, in case of an outage Amazon requires the customer to send it a claim within thirty business days. And Amazon S3 storage service guarantees a service reliability of 99.9% [4]. Here again, to be reimbursed, the customer has the responsibility to report Amazon S3 request failures, and to provide evidence to Amazon within ten business days. On the other hand, Amazon cloud services do not provide performance guarantees or other QoS guarantees. Similarly, Rackspace Cloud Servers compute service offers a service availability of 99.86%, and Rackspace Cloud Files storage service provides a service reliability of 99.9% [5]. Azure Compute guarantees a service availability level of 99.95% [6], and Azure Storage guarantees that 99.9% of storage requests are handled within fixed maximum processing times [7].

Some recent works consider Service Level Agreement (SLA) in cloud environments [26, 54]. SLA is a contract negotiated between a cloud service provider and a cloud customer. It specifies service level objectives (SLOs) that the cloud service must guarantee in the form of constraints on quality-of-service metrics. Chhetri et al. propose the automation of SLA establishment based on a classification of cloud resources in different categories with different costs, e.g. on-demand instances, reserved instances and spot instances in Amazon EC2 cloud [26]. However, this approach does not provide guarantees in terms of performance nor dependability. Macias and Guitart follow a similar approach for SLA enforcement, based on classes of clients with different priorities, e.g. Gold, Silver, and Bronze clients [54]. Here again, a relative best-effort behavior is provided for clients with different priorities, but no performance and dependability SLOs are guaranteed.

5.3 Main Challenges and Scientific Directions

As far as we know, no clouds adequately address service performance and dependability guarantees, leaving the following questions open:

How to address and combine multiple cloud SLOs in a consistent and flexible way? Today's clouds do not provide guarantees in terms of service performance. One would like the cloud to allow the customer to specify expectations regarding service request response time not to exceed a given maximum value, or cloud service throughput not to be below a given minimum value, etc. Regarding dependability, there are some initiatives in terms of guaranteed levels of cloud service availability and reliability [3, 4, 5, 6]. However, the provided SLOs are fixed by the cloud provider in an ad-hoc way and can not be specified by cloud customers in a flexible way. Furthermore, one would expect the cloud provider to allow the customer to combine multiple SLOs regarding performance, dependability, security, cost, etc. How could these SLOs be combined in a consistent way, knowing that some of them may raise antagonism and trade-offs?

How to provide better than heuristics-based and best-effort cloud QoS? Existing cloud services provide best-effort QoS management, usually based on over-provisioning resources and other heuristics for managing cloud resources. However, cloud customers would expect strict guarantees regarding cloud service performance, dependability, cost, etc. How could a cloud provide such strict QoS guarantees?

How to help system designers build QoS-oriented clouds and assess QoS guarantees? Cloud services are usually designed as black-boxes. Adding QoS management on top of these black-boxes is far from being trivial, and raises a challenging question: How to observe the behavior of cloud services in an accurate and non-intrusive way? On the other hand, with current cloud services the customer has the responsibility to report violations of QoS levels by analyzing log files [3, 4]. However, this is against one of the main motivations of cloud computing, that is hiding the administration complexity.

To address these challenges in a principled way, we call for the definition a new cloud model where QoS and SLA are first-class citizens. The new model should enrich the general paradigm of cloud and is orthogonal to Infrastructure-as-a-Service, Platform-as-a-Service, Software-as-as-Service and other cloud models, and may apply to any of them. The new cloud model must take into account both the cloud provider and cloud customer points of view. From the point of view of the cloud provider, autonomic SLA management must be provided to handle non-functional properties of cloud services, such as performance, dependability and financial cost in the cloud. On the other hand, from the point of view of the cloud customer, cloud SLA governance must be provided. It allows cloud customers to be part of the loop and to be automatically notified about the state of the cloud, such as SLA violations or cloud energy consumption. The former provides more transparency about SLA guarantees, and the latter aims to raise customer awareness about cloud service energy footprint. The new cloud model must allow the customer to choose the terms of the SLA with the cloud service provider, to specify the set of (possibly weighted) SLOs he requires, and to agree on the penalties in case of SLA violation. The SLOs can be expressed as thresholds to meet, or as QoS metrics to minimize or maximize.

To provide better than best-effort cloud QoS, a control-theoretic approach should be followed to design fully autonomic cloud services. First, a utility function should be defined to precisely describe the set of SLOs as specified in the SLA, the weights assigned to these SLOs if any, and the possible trade-offs and priorities between the SLOs. The cloud service configuration (i.e. combination of resources) with the highest utility is the best regarding SLA guarantees. Thus, how to find such a cloud service configura-

tion? Control theory techniques through modelling cloud service behavior, and proposing control laws and algorithms are good candidates for fully autonomic SLA-oriented cloud services [55]. The challenges for modelling cloud services are to build accurate models that are able to capture the non-linear behavior of cloud services, and that are able to self-calibrate to render the variations of service workloads. The challenge for controlling cloud services is to propose accurate and efficient algorithms and control laws that calculate the best service configuration, and rapidly react to changes in cloud service usage. Largely distributed cloud services would require efficient distributed control based on scalable distributed protocols.

To help build SLA-oriented clouds, cloud services should be designed to be controllable by construction. The services should allow to observe their behavior online, to monitor their changing QoS, and to apply changes on service configuration (i.e. resource set) while the service is running. To help system designers assess cloud service QoS guarantees, benchmarking tools are necessary to inject realistic workloads, data loads, fault loads, and attack loads into a cloud service, and to measure their impact on the actual performance, dependability and security of the service [8, 50, 69].

6. VERIFYING SECURITY POLICIES

For many years now, SLAs have been standard practice when setting up the terms of QoS for a service provision. However, SLAs normally steer clear of any explicit security commitments, possibly since cloud providers are reserved about the security guarantees of their services.

This point is proved by sources such as last year's report of CA Technologies and Ponemon Institute [44], where it was found out that, out of 127 cloud service providers in the US and Europe, over 80% do not believe that securing their services gives them a competitive advantage. Then, how can consumers protect their data and applications?

6.1 Requirements, Policies and Compliance

In order to secure cloud services, providers employ security measures that depend on a set of requirements. These requirements stem from two sources: external sources (e.g., laws and regulations), and particular requirements that users could request. Security policies express accurately both kinds of requirements. To make sure that such policies are respected, there are tools to enforce policies or verify compliance with policies.

External security requirements. To protect cloud users, official security requirements stem from two main sources: laws and regulations, and standards that providers should abide by. Sensitive data protection has been the target of EU and US laws for several years now, be it in healthcare or telecommunications. In Europe, for example, directive 95/46/EC protects personal data (among others, it forbids the collection and disclosure of such data without the subject's consent); in the US, the Health Insurance and Portability Act [79] aims to restrict access to computer systems that contain sensitive patient data, as well as to prevent interception or deletion of such data by unauthorised parties. In terms of security standards and guidelines, the most active sectors are healthcare and banking, with examples ranging from Health Level 7 to PCI's Data Security Standards [34]. The focus of such standards is securing healthcare and payment transactions. In all, such external requirements affect cloud consumers within a single domain or country, as well as across multiple jurisdictions. It is an open problem, outside the scope of regulations, what tools to use and how to employ them in order to satisfy such requirements, for both cloud providers and users.

Security policies. Unlike regulations and laws that can specify general security constraints in a text form, security policies are the machine-understandable specification of what a user considers to be accepted or allowed system behavior. A security policy of a cloud consumer can specify, for instance, that customer-identifiable data should not be propagated to other services; or that the owner should be notified of any backups or reconfigurations done to their service. Security policies can impose restrictions on: how to access and use system resources or the provided service; user accountability; key management; configuration of the back-end system (e.g., when to erase application data, when to do backups, connections to security services). Many enterprises have such policies already in place either as a good practice, or for auditing or certification purposes.

Tools to enforce or verify compliance. Enforcing a security policy means performing the actions to ensure that the application complies with that policy. Examples of security enforcement tools are Axiomatics XACML Policy Server, IBM's Tivoli Security Manager, or XML gateways such as Vordel. In a cloud setting, users can either: (1) set up their own enforcers, when they have control over some part of the infrastructure, or (2) rely on another party to enforce their policies, and then verify that the enforcement is done correctly. An after-the-fact verification usually involves analyzing *execution logs* provided a reporting service is in place and its output is provided to the user; at runtime clients can randomly probe the application to discover policy violations (fast but imprecise), or actively monitor application or service output (which can be a performance burden and involves an analysis architecture and process).

6.2 Existing work

Expressing security constraints. Surprisingly, it is only very recently that the notion of *security service-level agreements* has been proposed in the cloud context: one of the first is an HP report [62] suggesting that clients should negotiate those security needs that they can understand, predict and measure by themselves. Examples include: 95% of serious security incidents should be solved within one hour from detection; an up-to-date antivirus to scan the system every day; minimum network availability in case of an attack; the percentage of unpatched or unmanaged machines. In a similar vein, Jaatun et al. [45] suggest that a security SLA should include: the security requirements that the provider will enforce, the process of monitoring security parameters, collecting evidence, and assembling it to infer any security incidents; problem reporting; compensation and responsibilities. To this list, Breaux and Gordon [39] add the dimension of constraint changes across jurisdictions when regulations share a common focus. Further, a comparison of the potential languages that can be used by cloud users to express such requirements, has been made by Meland et al. [59]. The authors examine several languages usable for cloud SLAs, among which there are also XACML, WS-Agreement, LegalXML; however, they conclude that prior to choosing how to express security requirements, it is more stringent to converge towards common concepts of security

contracts. A step in that direction has been made on the industry side: the Cloud Security Alliance has issued CSA-STAR [32], a public registry of the security controls offered by popular cloud security providers.

Malicious insiders. The role of the system administrator has become much more prominent in the cloud, and administrators are scarce resources. First, they have to be competent at managing intricate multitenant systems that still require an amount of manual maintenance; second, cloud administrators have an exacerbated security responsibility because their actions can affect sensitive data and numerous users. With root privileges, an administrator can read log files, configurations, patch binaries and run executables. As shown with the four attacks exemplified in [68], a malicious or sloppy administrator hence violate user data confidentiality, integrity, and even the availability of cloud nodes. In order to harden compute nodes, Bleikertz et al. suggested a solution [16] to minimize administrator privileges during maintenance operations. The authors identify five privilege levels that an administrator can have over a node; security policies deployed on each node define the transitions between privilege levels, while enforcing these policies and system accountability are performed with SELinux mechanisms.

Monitoring and resource management. Recent work has shown more and more attacks on cloud resources, of which there are denial of service attacks exploiting cloud underprovisioning [52]; fraudulent consumption for Web resources [43]; or even several vulnerabilities of the Xen's resource scheduler that allows malicious customers to use resources at the expense of somebody else, in Amazon EC2 [87]. These examples show that even if virtualisation is supposed to ensure isolation among customers, this isolation is not complete and there is always another shared resource (cache, memory, network, etc.) that can be exploited. Sharing thus becomes very hard to measure, since providers are very likely to charge incorrectly or even increase their expenses. To bridge this gap, Sekar and Maniatis proposed the notion of *verifiability* by which customers can check their resource consumption, with the help of a trusted consumption monitor which validates consumption reports [73]. Such monitor is faced with several challenges: reporting can clog bandwidth, and performance might suffer because of too frequent measurement. The authors suggest offloading the monitoring to dedicated entities, sampling and periodic snapshots of resource consumption. A similar solution is suggested by Wang and Zhou [81], who aim to provide a dedicated service to collect and monitor evidence that a multitenant platform is accountable.

6.3 Main challenges and next directions

Cloud consumers and providers are often tempted to consider encryption as the only security tool for sufficient protection when using the cloud. Yet, encryption is not a panacea for everything: data and resources are shared, applications are outsourced, and schemes to control the access to such resources sometimes fall short in the face of sloppy users, malicious insiders, or system misconfigurations. Moreover, when this happens, users are often expected to provide proof of the security violations. We therefore believe that users may benefit from additional tools that may allow them to (1) articulate the desired security policy, (2) gain evidence in case violations happen, and (3) choose a better provider in case such violations are too frequent. We further expand on these points below.

First, clients may benefit from being able to express and reason about the security of their data or services. Cloud users often expect protection not only for their personal data when it is stored or when it propagates, but also for their service access information and usage habits. As of now there is no agreed upon way to express constraints on how your data should be handled by a provider (e.g., data lifetime, redundancy schemes, usage and propagation in other countries).

Second, it is hard to prove that a policy violation has actually occurred. Application monitoring and offline analysis can be used for this purpose. Monitoring can be done by the client, with the help of tools that can intercept and examine relevant events in the cloud. To enforce information flow constraints, some solutions have already been suggested: gateways or security proxies control the data flows both to and from service providers, and are already on the market as mentioned before. Similarly, solutions like CloudFilter [64] can intercept HTTP traffic, and filter out sensitive data that had been labelled internally; with such labels and contextual information, a security decision is made based on a set of policies on data treatment. In all, these solutions are better suited for those customers able to set up their own policy compliance checking proxies, somewhere in the cloud. But this is often not possible, nor straightforward, in which case may prefer to use third party services to enact various type of security constraints. Such services need not only consider regulatory requirements that apply to each jurisdiction and domain (e.g., encryption key size, anonymisation, etc), but also specific security requirements of the clients, that can be measured at runtime.

Third, it may be useful for users to be able to compare providers in terms of security assurance. This proposition is currently challenging for a user who is in search of a provider, and we feel that more research should concentrate in that direction. Conversely, if a user uses two cloud providers at the same time, such a comparison can be achieved using reporting tools similar to those used to measure accountability of user actions and their resource consumption. Furthermore, cloud providers currently offer little or no proofs to their customers that what they billed them was right; nor can the users prove that they did, or did not, use more resources than they should have. Some existing approaches [81, 83] suggest relying on an external accountability monitors, but there are still several challenges in that respect: performance, trust model used, and privacy. In terms of performance, it is a challenge to draw the line between how often to report service activity so as not to clog the communication lines, and how much information in the report is actually relevant for later analysis. In terms of trust model, it is important to determine to what extent the consumer and the provider should trust each other in reporting truthfully. Tools for ensuring timestamping and log tamper-resistance are already in place. In terms of privacy, reporting should be sufficient to detect faults and at the same time should not expose private user data.

7. CONCLUSIONS

This paper surveys the tools and methods that cloud users and service providers can employ to verify that cloud services behave as expected. We focus on the verification of

several properties: the identity of the service and of the nodes the service runs on; functional correctness of a service; SLA-imposed parameters like performance and dependability; and lastly the compliance of the service with security requirements as specified by a security policy. We discussed state of the art in these areas and identified gaps and challenges, which explain the lack of sufficient tools for monitoring and evaluation of cloud services. In each of these areas we highlighted new and promising directions that we believe to be instrumental in developing such tools in the future. We hope that our paper will encourage future research in this area.

Acknowledgement

The authors would like to thank Rüdiger Kapitza and the other organisers of the Dagstuhl seminar 12281 "Security and Dependability for Federated Cloud Platforms" (July 2012), who have bolstered this collaboration.

8. REFERENCES

[1] Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, 2001.

[2] SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). `http://www.w3.org/TR/soap12-part1`, 2007.

[3] Amazon EC2 SLA. `https://aws.amazon.com/ec2-sla/`, 2012.

[4] Amazon S3 SLA. `https://aws.amazon.com/simpledb/`, 2012.

[5] Rackspace SLA. `http://www.rackspace.com/cloud/legal/sla/`, 2012.

[6] Windows Azure Compute SLA. `https://www.microsoft.com/download/en/details.aspx?displaylang=en\&id=24434`, 2012.

[7] Windows Azure Storage SLA. `https://www.microsoft.com/windowsazure/features/storage/`, 2012.

[8] D. Agarwal and S. K. Prasad. Azurebench: Benchmarking the storage services of the azure cloud platform. In *IPDPS Workshops*, pages 1048–1057. IEEE Computer Society, 2012.

[9] N. Alon, M. Krivelevich, I. Newman, and M. Szegedy. Regular languages are testable with a constant number of queries. In *Proc. 40th IEEE Symposium on Foundations of Computer Science*, pages 645–655, 1999.

[10] American Express may have failed to encrypt data. `http://www.scmagazine.com/american-express-may-have-failed-to-encrypt-data/article/170997/`.

[11] E. Anderson, X. Li, M. Shah, J. Tucek, and J. Wylie. What consistency does your key-value store actually provide. In *Proceedings of the Sixth international conference on Hot topics in system dependability*, pages 1–16. USENIX Association, 2010.

[12] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 598–609. ACM, 2007.

[13] C. Băsescu, C. Cachin, I. Eyal, R. Haas, A. Sorniotti, M. Vukolić, and I. Zachevsky. Robust data sharing with key-value stores. In *Proc. Intl. Conference on Dependable Systems and Networks (DSN)*, June 2012.

[14] S. A. Baset. Cloud SLAs: present and future. *SIGOPS Oper. Syst. Rev.*, 46(2):57–66, July 2012.

[15] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. In *Proc. 6th European Conference on Computer Systems (EuroSys)*, pages 31–46, 2011.

[16] S. Bleikertz, A. Kurmus, Z. A. Nagy, and M. Schunter. Secure cloud maintenance – protecting workloads against insider attacks. In *ASIACCS ACM Symposium on Information, Computer and Communications Security*, 2012. to appear.

[17] K. Bowers, A. Juels, and A. Oprea. Hail: a high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 187–198. ACM, 2009.

[18] K. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 43–54. ACM, 2009.

[19] K. Bowers, M. van Dijk, A. Juels, A. Oprea, and R. Rivest. How to tell if your cloud files are vulnerable to drive crashes. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 501–514. ACM, 2011.

[20] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy. Self-service Cloud Computing. In *CCS*, 2012.

[21] C. Cachin and M. Geisler. Integrity protection for revision control. In *Applied Cryptography and Network Security*, pages 382–399. Springer, 2009.

[22] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. *SIAM Journal on Computing*, 40(2):493–533, 2011.

[23] C. Cachin and M. Schunter. A Cloud You Can Trust. `http://spectrum.ieee.org/computing/networks/a-cloud-you-can-trust`, 2011.

[24] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[25] R. Cellan-Jones. The Sidekick Cloud Disaster. `http://www.bbc.co.uk/blogs/technology/2009/10/the_sidekick_cloud_disaster.html`, 2009.

[26] M. B. Chhetri, Q. B. Vo, and R. Kowalczyk. Policy-Based Automation of SLA Establishment for Cloud Computing Services. In *The 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*, pages 164–171, Washington, DC, USA, 2012.

[27] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 265–278, New York, NY, USA, 2011. ACM.

[28] V. Chipounov, V. Kuznetsov, and G. Candea. The s2e platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, Feb. 2012.

[29] H. Chockler and O. Kupferman. ω-regular languages are testable with a constant number of queries. *Theor. Comput. Sci.*, 329(1-3):71–92, 2004.

[30] B. Chun, C. Curino, R. Sears, A. Shraer, S. Madden, and R. Ramakrishnan. Mobius: unified messaging and data serving for mobile apps. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 141–154. ACM, 2012.

[31] B. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. *ACM SIGOPS Operating Systems Review*, 41(6):189–204, 2007.

[32] Cloud Security Alliance. CSA - Security, Trust, and Assurance Registry. `https://cloudsecurityalliance.org/star/`, 2011.

[33] CNN Money. Amazon EC2 outage downs Reddit, Quora. `http://money.cnn.com/2011/04/21/technology/amazon_server_outage/index.htm`, 2011.

[34] P. S. S. Council. PCI Data Security Standard, v2. https://www.pcisecuritystandards.org/security_standards/documents.php?document=pci_dss_v2-0#pci_dss_v2-0, 2010.

[35] R. DeVries. RichardDeVries's Journal: How Google handles a bug report. http://slashdot.org/~RichardDeVries/journal/225229, 2009.

[36] A. Feldman, W. Zeller, M. Freedman, and E. Felten. Sporc: Group collaboration using untrusted cloud resources. *OSDI, Oct*, 2010.

[37] A. Ferdowsi. The Dropbox blog: Yesterday's Authentication Bug. https://blog.dropbox.com/?p=821, 2011.

[38] R. T. Fielding. *Chapter 5: Representational State Transfer (REST)*. University of California, Irvine, 2000. Ph.D. Thesis.

[39] D. G. Gordon and T. D. Breaux. Managing multi-jurisdictional requirements in the cloud: towards a computational legal landscape. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 83–94, New York, NY, USA, 2011. ACM.

[40] T. C. Group. TPM Main Specification Level 2 Version 1.2, Revision 130, 2006.

[41] J. Guitart, J. Torres, and E. Ayguadé. A survey on performance management for internet applications. *Concurr. Comput. : Pract. Exper.*, 22(1):68–106, 2010.

[42] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[43] J. Idziorek and M. Tannian. Exploiting cloud utility models for profit and ruin. *2012 IEEE Fifth International Conference on Cloud Computing*, 0:33–40, 2011.

[44] P. Institute. Security of Cloud Computing Providers Study. http://www.ca.com/~/media/Files/IndustryResearch/security-of-cloud-computing-providers-final-april-2011.pdf, 2011.

[45] M. Jaatun, K. Bernsmed, and A. Undheim. Security SLAs - An Idea Whose Time Has Come? In *Multidisciplinary Research and Practice for Information Systems*, volume 7465 of *Lecture Notes in Computer Science*, pages 123–130. Springer Berlin Heidelberg, 2012.

[46] A. Juels and B. S. Kaliski, Jr. Pors: proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 584–597. ACM, 2007.

[47] J. C. King. A new approach to program testing. In *Proceedings of the international conference on Reliable software*, pages 228–233, New York, NY, USA, 1975. ACM.

[48] M. Krigsman. Intuit: Pain and Pleasure in the Cloud. http://www.zdnet.com/blog/projectfailures/intuit-pain-and-pleasure-in-the-cloud/14880, 2011.

[49] J. C. Laprie. Dependable Computing and Fault-Tolerance: Concepts and Terminology. In *IEEE Int. Symp. on Fault-Tolerant Computing (FTCS)*, 1985.

[50] A. Lenk, M. Menzel, J. Lipsky, S. Tai, and P. Offermann. What are you paying for? performance benchmarking for infrastructure-as-a-service offerings. In L. Liu and M. Parashar, editors, *IEEE CLOUD*, pages 484–491. IEEE, 2011.

[51] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *OSDI*, pages 121–136, 2004.

[52] H. Liu. A new form of dos attack in a cloud and its avoidance mechanism. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, CCSW '10, pages 65–76, New York, NY, USA, 2010. ACM.

[53] C. Loosley, F. Douglas, and A. Mimo. *High-Performance Client/Server*. John Wiley & Sons, Nov. 1997.

[54] M. Macias and J. Guitart. Client Classification Policies for SLA Enforcement in Shared Cloud Datacenters. In *The 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*, pages 156–163, Washington, DC, USA, 2012.

[55] L. Malrait, S. Bouchenak, and N. Marchand. Experience with ConSer: A System for Server Control Through Fluid Modeling. *IEEE Transactions on Computers*, 60(7):951–963, 2011.

[56] P. Maniatis, D. Akhawe, K. Fall, E. Shi, S. McCamant, and D. Song. Do You Know Where Your Data Are? Secure Data Capsules for Deployable Data Protection. In *HotOS*, 2011.

[57] E. Marcus and H. Stern. *Blueprints for High Availability*. Wiley, Sept. 2003.

[58] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 108–117. ACM, 2002.

[59] P. H. Meland, K. Bernsmed, M. G. Jaatun, A. Undheim, and H. Castejon. Expressing cloud security requirements in deontic contract languages. In *CLOSER*, pages 638–646. SciTePress, 2012.

[60] D. A. Menascé and V. A. F. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall, 2001.

[61] R. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and privacy*, volume 1109, pages 122–134, 1980.

[62] B. Monahan and M. Yearworth. Meaningful Security SLAs. http://www.hpl.hp.com/techreports/2005/HPL-2005-218R1.html, 2008.

[63] A. Oprea and M. Reiter. On consistency of encrypted files. *Distributed Computing*, pages 254–268, 2006.

[64] I. Papagiannis and P. Pietzuch. Cloudfilter: practical control of sensitive data propagation to the cloud. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, CCSW '12, pages 97–102, New York, NY, USA, 2012. ACM.

[65] R. Patton. *Software Testing*. SAMS Publishing, second edition, 2005.

[66] S. Pearson and A. Benameur. Privacy, security and trust issues arising from cloud computing. *Cloud Computing Technology and Science, IEEE International Conference on*, 0:693–702, 2010.

[67] M. Pezze and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[68] F. Rocha and M. Correia. Lucy in the sky without diamonds: Stealing confidential data in the cloud. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, DSNW '11, pages 129–134, Washington, DC, USA, 2011. IEEE Computer Society.

[69] A. Sangroya, D. Serrano, and S. Bouchenak. Benchmarking Dependability of MapReduce Systems. In *The 31st IEEE International Symposium on Reliable Distributed Systems (SRDS 2012)*, 2012.

[70] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards Trusted Cloud Computing. In *HotCloud*, 2009.

[71] N. Santos, R. Rodrigues, K. Gummadi, and S. Saroiu. Policy-Sealed Data: A New Abstraction For Building Trusted Cloud Services. In *USENIX Security*, 2012.

[72] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel. Seeding Clouds with Trust Anchors. In *WCCS*, 2010.

[73] V. Sekar and P. Maniatis. Verifiable resource accounting for cloud computing services. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 21–26, New York, NY, USA, 2011. ACM.

[74] SensePost Blog, DEF CON 17 Conference. Clobbering the Cloud, 2009. http://www.sensepost.com/blog/3706.html.

[75] H. Shacham and B. Waters. Compact proofs of retrievability. *Advances in Cryptology-ASIACRYPT 2008*,

pages 90–107, 2008.

[76] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, pages 19–30. ACM, 2010.

[77] 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 Computing. In *SOSP*, 2011.

[78] The Guardian. PlayStation Network hack: why it took Sony seven days to tell the world. `http://www.guardian.co.uk/technology/gamesblog/2011/apr/27/playstation-network-hack-sony`, 2011.

[79] United States Congress. Health Insurance Portability Act. `http://www.gpo.gov/fdsys/pkg/PLAW-104publ191/html/PLAW-104publ191.htm`, 1996.

[80] M. van Dijk, A. Juels, A. Oprea, R. Rivest, E. Stefanov, and N. Triandopoulos. Hourglass schemes: how to prove that cloud files are encrypted. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 265–280. ACM, 2012.

[81] C. Wang and Y. Zhou. A collaborative monitoring mechanism for making a multitenant platform accountable. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 18–18, Berkeley, CA, USA, 2010. USENIX Association.

[82] G. Watson, R. Safavi-Naini, M. Alimomeni, M. Locasto, and S. Narayan. Lost: location based storage. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, pages 59–70. ACM, 2012.

[83] J. Yao, S. Chen, C. Wang, D. Levy, and J. Zic. Accountability as a Service for the Cloud. *Services Computing, IEEE International Conference on*, 0:81–88, 2010.

[84] A. Yumerefendi and J. Chase. Strong accountability for network storage. *ACM Transactions on Storage (TOS)*, 3(3):11, 2007.

[85] K. Zellag and B. Kemme. How consistent is your cloud application? In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 6. ACM, 2012.

[86] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *SOSP*, 2011.

[87] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram. Scheduler vulnerabilities and attacks in cloud computing. *IEEE International Symposium on Networking Computing and Applications*, 2011.