Abstract

Accountability in Cloud Computing and Distributed Computer Systems

Hongda Xiao

2014

Traditionally, research in computer security has focused on *preventive* techniques such as passwords, authentication protocols, and encryption. With the rise of internet technologies, especially cloud computing and distributed computer systems, preventive measures have proven inadequate, and security researchers have sought to complement them with *accountability mechanisms*. Despite widespread agreement on its importance, “accountability is not yet a unified research area. This thesis advances the state of accountability research by systematically comparing a significant amount of existing work in the area and designing practical accountability mechanisms for realistic scenarios in cloud computing and distributed computer systems.

First, we propose a framework to categorize research on accountability mechanisms with respect to time, goal, information, and action. Our systematization effort shows that more sparing use of the word “accountability” is desirable, because it is currently used by different researchers to mean different things. Our conception of the term dispels the mistaken notions that accountability precludes anonymity and privacy and that it requires centralized authority.

Second, we present a privacy-preserving structural-reliability auditor (P-SRA) for cloud-computing systems. P-SRA enables evaluation of the reliability of cloud infrastructure without compromise of cloud-infrastructure providers privacy. We present the privacy properties of P-SRA and evaluate a prototype implementation built on the Sharemind SecreC platform [BK13]. P-SRA is not only a mechanism for holding cloud-service providers accountable but also an interesting application of *secure multi-party computation* (SMPC), an extensive body of privacy technology that has not often been used on graph problems, which are inherent in structure-reliability auditing.

Third, we extend our study of the accountability of cloud-service providers to cloud-service users; rather than focusing only on reliability, we consider general properties of cloud infrastructure. We develop the

notion of *cloud user infrastructure attestation*, which enables a cloud-service provider to attest to a cloud-service user that the infrastructure as a whole has the properties that the cloud-service user has requested. Here, “infrastructure including both the computing nodes on which the users virtual machines run and the interconnection of these virtual machines. We propose a novel type of secure-hardware component called a *Network TPM* to guarantee the integrity of the cloud infrastructure information, and we design attestation protocols that leverage existing verifiable-computation techniques. Our protocols protect the privacy of the cloud-service provider, who does not need to reveal the physical infrastructure and its details to the cloud-service user or to any third parties.

Finally, we study accountability in the operation of cloud-scale data centers — specifically on the actions that should be taken when violations of system policies or other abnormal events are detected. We focus on rapid reallocation of virtual machines in response to threat detection. We formally define virtual-machine reallocation as an optimization problem and explain how it differs from the general virtual-machine allocation problem. Virtual-machine reallocation is NP-hard, but we provide an efficient, two-layered, heuristic algorithm that decomposes the problem and then applies optimization techniques to much smaller problem instances. Our approach incurs only small optimality losses and may be applicable to other aspects of data-center and cloud security.

Accountability in Cloud Computing and Distributed Computer Systems

A Dissertation

Presented to the Faculty of the Graduate School

of

Yale University

in Candidacy for the Degree of

Doctor of Philosophy

by

Hongda Xiao

Dissertation Director: Professor Joan Feigenbaum

Dec 2014

Acknowledgements

First, I would like to thank my advisor, Professor Joan Feigenbaum, for guiding me into the research field of computer security, sharing with me her enthusiasm and insights, supporting me when I was in the most difficult time during my PHD studies, teaching me how to write good papers, helping me edit my papers and presentation slides, encouraging me to fight for my dreams, and so many other things that I dont have space for here. Because of her, my PHD studies become an exciting and rewarding journey.

Next, I would like to thank my wife Grace, who always believes in me unconditionally and will bring me an amazing gift from God our son Jason. I would like to thank my parents, Tianlong Xiao and Bing He, who always support me no matter what happens and have given me everything that they have to help me pursue my dreams. I would like to thank my grandfather, Wenbing He, for appreciating the progress I have made, and to wish him rest and peace in Heaven. I would like to thank my grandmothers, Shuqing Wang and Yuchan Liu, who are always proud of me. And I thank my parents-in-law, Hao Guo and Xuenong Kang, Uncle Tianya Xiao, Uncle Tianzhi Xiao, Uncle Tianbao Xiao, Uncle Pei Guo, Uncle Xuejun Kang, Aunt Xiuqin Wang, Aunt Dandan He, Aunt Jun He, Aunt Lichun He, Aunt Lihua He, Cousin Hongyu Xiao, Cousin Chong Zheng, and Cousin Yinggang Li for giving me so many wonderful moments in my life.

I would also like to thank my co-authors and collaborators for their enthusiasm in research and for all of their hard work: Professors Bryan Ford and Joan Feigenbaum for guiding me and helping me to finish the work of Chapter 3; Professors Jakub Szefer and Joan Feigenbaum for proposing the interesting problem of cloud user infrastructure attestation in Chapter 4; Professor Jakub Szefer for all of his efforts on the work of Chapter 5; Professors Aaron Jaggard and Rebecca Wright for collaborating on the work in Chapter 2; Ennan Zhai, David Wolinskey, Hongqiang Liu, Xueyuan Su, and Professor Bryan Ford for our joint work on the SRA system [ZWX$^+$]; and fellow PhD students Debayan Gupta and Aaron Segal for helpful discussions about SMPC.

Finally, I acknowledge the National Science Foundation for supporting my research with grant CNS 1016875.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traditionally, computer-security research has focused on *preventive* approaches to security and privacy. Although preventive techniques, such as passwords and authentication protocols, are still indispensable in computer systems, purely preventive approaches have proven to be inadequate in the internet era. As more and more daily activity moves online, users in different administrative domains must exchange information and transact business without the benefit of a common set of policies and credentials. With the development of cloud computing and large-scale, distributed computing systems, information-security researchers have realized the importance of *accountability mechanisms* to complement preventive mechanisms. Outsourcing computations to powerful third parties and distributed organizations of large-scale systems accelerates the process of data sharing and cooperation between different parties who do not necessarily trust each other. It is difficult, if not impossible, to guarantee service-level agreements between cloud-service users and providers or to enforce the policies of the distributed systems by pure preventive mechanisms.

Despite widespread agreement about its importance, accountability is not yet a unified research area yet. Different researchers use the term "accountability" to mean different things. In Chapter 2, we propose a framework to systematize much of the existing computer-science research on accountability. We provide a high-level perspective on the appropriate focus of accountability work in computer science, and we categorize existing work along three axes: time, information, and action. With respect to time, we consider five standard approaches to violations and potential violations of security policies: prevention, detection, evidence, judgment, and punishment. With respect to information, we examine the type(s) of credentials used by system participants, the components of evidence of compliance with or violation of a security policy, and who must have access to credentials and evidence for the system to function. With respect to action, we examine the operational structures of accountability mechanisms and the systems that use them to achieve privacy and security. Our systematization effort has revealed the need for more sparing use of the word "accountability" and, more generally, for more precise and consistent terminology. Our formulation of accountability also dispels the mistaken notions that accountability precludes anonymity and privacy and that it requires centralized authority.

We then proceed to design accountability mechanisms for practical cloud computing and distributed computing systems. In cloud-computing systems, cloud-service users need to hold cloud-service providers accountable for providing reliable cloud services. It is natural for cloud-service providers to use redundancy to achieve reliability. For example, a provider may replicate critical state in two data centers. However, if the two data centers use the same power supply, a power outage will cause them to fail simultaneously; replication *per se* does not, therefore, enable the cloud-service provider to make strong reliability guarantees to its users. On the other hand, cloud-service providers may be unwilling to reveal sensitive information about their equipment and operational procedures to cloud-service users or any third parties, because doing so may compromise their businesses. In Chapter 3, we present a privacy-preserving structural-reliability auditor (P-SRA), discuss its privacy properties, and evaluate a prototype implementation built on the Sharemind SecreC platform [BK13]. P-SRA is not only an accountability mechanism that enables cloud-service users to hold cloud-service providers accountable; it is also an interesting application of *secure multi-party computation* (SMPC), which has not often been used for graph problems. It achieves acceptable running times even on large cloud structures by using a novel data-partitioning technique that may be useful in other applications of SMPC. Moreover, it demonstrates that accountability can be achieved without compromising the privacy of the system participants.

In Chapter 4, we continue our study of techniques that allow cloud-service users to hold cloud-service providers accountable. Rather than focusing on just one property (reliability), as we did in Chapter 3, we consider the general question of how to determine whether the promised properties of cloud infrastructures have actually been delivered. Cloud-service users need assurance that they have received cloud resources with the properties that they paid for and that their virtual machines function as expected. A practical accountability mechanism should enable cloud-service providers to attest to cloud-service users that the resources they have provided satisfy the users' requirements. In particular, in addition to attestation that the computing resources delivered have the desired properties, such as processor speed or amount of memory, cloud-service users need attestation about the networking infrastructure that connects the virtual machines they are running in the cloud. We develop the notion of *cloud user infrastructure attestation*, which attests to the properties of the whole infrastructure that a cloud-service user has requested from a cloud-service provider, including both the computing nodes where the virtual machines run and the interconnection of these virtual machines. We propose the use of a novel secure-hardware component called a *Network TPM* to attest to the properties of the networking infrastructure; it is meant to work in concert with traditional TPMs that attest to the properties of the computing nodes. On top of these hardware security anchors, we build protocols that are able to attest to the properties of the infrastructure that the user has leased from the provider. In addition, our protocols protect the privacy of the provider, who does not need to reveal details of its physical infrastructure to the user or to any third parties.

Finally, we study accountability in the operation of cloud-scale data centers — specifically on the actions that should be taken when violations of system policies or other abnormal events are detected. In Chapter 5, we investigate the data-integrity and data-security problems in cloud-based data centers, which are often organized in a distributed manner. Unexpected and sudden events in data centers, such as detection of an unauthorized physical intruder or a critical security event, require prompt responses if data integrity and security are to be protected. Standard responses include resource reallocation and migration of computation or data away from the affected hosts. Previous research in this area has focused on implementing fast and efficient migration strategies. However, past work has not often explored how to select the target machines to which computation and data should be migrated after the occurrence of sudden events. We focus on rapid reallocation of virtual machines in response to threat detection. We formally define virtual-machine reallocation as an optimization problem and explain how it differs from the general virtual-machine allocation problem. Virtual-machine reallocation is NP-hard, but we provide an efficient, two-layered, heuristic algorithm that decomposes the problem and then applies optimization techniques to much smaller problem instances. We use large-scale simulations to demonstrate that the two-layered approach is fast enough for the configurations of real-world data centers, with only small and tolerable optimality losses. Our layered approach may be applicable to other aspects of data-center and cloud security problems.

In Chapter 6, we present some open problems and directions for future research on accountability in cloud services and distributed computing systems.

Chapter 2

Systematizing Accountability in Computer Science

2.1 Introduction

Traditionally, computer-science researchers have taken a *preventive* approach to security and privacy in online activity: Passwords, authentication protocols, and other before-the-fact authorization mechanisms are designed to prevent users from violating policies and to obviate the need to adjudicate violations and punish violators. Purely preventive approaches to security and privacy have proven to be inadequate as more and more daily activity moves online, and users in different administrative domains must exchange information and transact business without the benefit of a common set of policies and credentials. Many information-security researchers have thus sought *accountability mechanisms* to complement preventive mechanisms. Despite widespread agreement that "accountability" is important in online life, it is not yet a unified research area. Indeed, the word is used by different researchers to mean different things and is not always defined explicitly.

It is our thesis of this chapter that the lack of agreement about definitions and formal foundations is impeding progress on accountability research and adoption of accountability technology. We offer this systematization as a step toward remedying this situation. Our starting point is a succinct, high-level perspective on the appropriate focus of accountability work in computer science: Accountability mechanisms should enable actions to be tied to consequences and, in particular, enable violations to be tied to punishment. Guided by that fundamental goal, we categorize existing work on accountability along three axes: time, information, and action.

With respect to time, we consider five standard approaches to violations and potential violations of security policies: prevention, detection, evidence, judgment, and punishment. Roughly speaking, these approaches can be linearly ordered in time. First, one tries to prevent violations. When that cannot be done, the goal is to detect violations. If a violation is detected, or even suspected, it may be necessary to gather evidence that can later be used to render a judgment about precisely what happened and whom or what to blame. Finally, actions can be tied to consequences by meting out punishment to the violator. A single accountability mechanism can address one or more of these five phases; most do not address them all. A stand-alone authentication or authorization mechanism that is purely preventive should not be called an "accountability" mechanism, but before-the-fact authorization can be part of a larger system that also addresses the later phases of accountability.

With respect to information, we examine the type(s) of credentials the system participants use, what constitutes evidence of compliance with or violation of a security policy, and who must have access to credentials and evidence for the system to function. To what extent does the system rely on participants' identities, and how is "identity" defined? If identity is used, how broadly does a participant's identity become known? Who learns about a violation when one occurs, and how soon after the fact of the violation does he learn it? The role of identity is important because of the widespread but mistaken perception (discussed below) that accountability is inherently in tension with anonymity. Interestingly, some of the works that we cover in this systematization effort regard identification of wrongdoers as the final step in a process—as judgment and punishment, in the terms introduced above. In these systems, an act that violates a security policy triggers the identification of the violator who, until he committed the violation, was anonymous. It is assumed that identification *per se* will ensure that the violator is held accountable, but precisely what it means for someone to be "held accountable" is not specified. At the other end of the spectrum, some of the works that we cover assume that all participants have persistent identities, *i.e.*, that anonymity is not an issue, and deal exclusively with formal protocols for presenting evidence, adjudicating a claimed violation, and meting out punishment if the claim is validated. This lack of agreement about the scope of "accountability" research is one of our main motivations for undertaking this systematization effort.

With respect to action, we examine the operational structures of accountability mechanisms and the systems that use them to achieve privacy and security. Are system actions centralized or decentralized? What actions must be taken to deal with a violation? In particular, does a violation trigger *automatic* punishment (such as the destruction of anonymity discussed above), or must evidence of a violation be presented to a *mediator*, who invokes a formal adjudication protocol and, if necessary, a punishment protocol? If there is a mediator, is it an entity that is already part of the system, or is it someone external to the system (like a judge who is only called in for the purpose of adjudicating)? To what extent does the functioning of the system assume continued participation by or access to the violator? That actions could be tied to consequences automatically, *e.g.*, without identification of the actors or the invocation of a formal adjudication protocol, is not a new or radical idea but rather one that has been the subject of extensive study in at least one discipline, namely Economics, in which the design of *incentive-compatible* systems and protocols is a standard goal. The simplest and best-known example of an incentive-compatible protocol in Economics is the 2nd-price Vickrey auction. The policy that bidders are supposed to comply with is "bid your true value."[1] For many natural distributions on the bidders' values, no bidder can improve his utility by lying; indeed, with positive probability, his utility will be decreased if he lies about his value. Thus, actions are *automatically* tied to consequences, and no explicit punishing action is taken. The violator is not identified, and, in fact, no one else even knows that there was a violation.

One barrier to unification and systematization of this technical area is the word "accountability" itself. In common parlance, "holding him accountable" connotes "making him account for himself" or "making him stand up and be counted." The sentiment conveyed therein has considerable social value, and it causes people to resist using the term to describe approaches that may not entail an official "account" by the wrongdoer. This erroneous assumption that "accountability mechanisms" must require the identification of those who violate policies so that violators can be brought to "account" is widespread in the technical community as well, where it raises the hackles of those who conclude that accountability is inherently in tension with anonymity. The fact that "tying actions to consequences" can be accomplished without identifying wrongdoers, as the study of incentive compatibility in Economics clearly demonstrates, gives us hope that this erroneous assumption can be corrected and that the technical community will embrace accountability as an effective tool in situations where preventive measures are inadequate and will recognize that it does not preclude anonymity.

2.2 Related work

The focus of this chapter is on accountability solutions and formalizations in Computer Science; that type of related work is discussed in detail below. Other work on accountability in Computer Science includes arguments by Weitzner *et al.* [WABL$^+$08] and by Lampson [Lam09] about the need for accountability and security-by-deterrence (such as might be provided by accountability). In early work on accountability in Computer Science, Nissenbaum [Nis97] studied barriers to accountability in contexts involving software.

[1] This might not be explicitly stated as a policy requirement, but that does not affect incentive compatibility. In considering this in the context of accountability, we may assume that we are in a setting in which this goal is an explicit policy and that we want to ensure that violations of this policy are punished.

Chockler and Halpern [CH04] build on the Halpern–Pearl [HP05] framework of causality to give formal definitions of both responsibility (the extent to which something is a cause of an event) and blame (which additionally considers the epistemic state of an agent who causes an event). This does not directly provide a definition of accountability, but these notions might be used to inform actions (such as punishment) taken in response to a policy violation.

Outside of Computer Science, Mulgan [Mul00] has traced the evolution of "accountability" in Public Administration from its core meaning of being able to be called to give an account (*e.g.*, of one's actions). Grant and Keohane [GK05] have given a definition in the context of nation states interacting with each other. Our focus is not on approaches outside of computer science, of which these are but a small sample, so we will not discuss them in more detail here. (Feigenbaum *et al.* [FHJ$^+$11, FJW11] provide more discussion of non-Computer Science approaches to accountability.)

2.3 Aspects of Accountability

As we survey approaches to accountability, we evaluate how they address three broad aspects of accountability: time, information, and action. In our analysis, we typically think of accountability with respect to some policy violation (in a very general sense); the "time" aspect considers when the system is invoked relative to the time of the violation. The "information" aspect considers what is known and by whom, while the action aspect concerns what is done and by whom.

2.3.1 Time/Goals

In surveying approaches to accountability, it becomes clear that different systems are focused on different times relative to a policy violation; this often corresponds to different goals for the system. As one example, the formal framework of Küsters *et al.* [KTV10] explicitly models (and focuses on) judgments or verdicts, *i.e.*, declarations that a violator is guilty of committing a violation. By contrast, the formal framework of Feigenbaum *et al.* [FJW11] focuses on punishment, which typically follows a declaration of guilt. (Within this punishment-focused framework, there need not be a judgment that identifies an individual entity as guilty; so this is indeed distinct from a focus on judgment.)

Motivated by such differences, we consider a spectrum of times, relative to a policy violation, at which each system/framework/mechanism might play a role. We identify the points below on this spectrum. While we categorize, and refer to, these based in terms of their goals/effects and not in terms of strict temporal relationships, there is a natural temporal ordering of these effects.

Prevention The system is (at least partially concerned) with preventing violations and plays a role before the violation occurs.

Detection The system facilitates, enables, *etc*., detection of a violation (either as it occurs or after it occurs).

Evidence The system helps gather or preserve evidence about a violation that may be used against the accused violator (*e.g.*, in a court of law); in some settings, this may be connected to detection.

Judgment The system renders a verdict about an entity's guilt with respect to a policy violation. (This might be a verdict in a court of law or, *e.g.*, a determination by a system administrator that a particular user violated local policy.)

Punishment The system punishes a policy violator in some way.

As we will observe below, a single system might be involved at multiple points on this spectrum.

2.3.2 Information

One question about accountability is the extent to which it implicates privacy. Two aspects of this are the information learned about a violation and the information learned about the violator (or even individuals who do not violate any policy). In studying this, we ask the following related questions about accountability systems:

- Is identity required to participate in the system? If so, how broadly is a participant's identity known (*e.g.*, is it only learned by a trusted third party, is it learned by a limited set of entities, or is it potentially learned by all participants)?

- Are violations disclosed? If so, how broadly (with the same set of possible answers as for identity)? How soon after the violation is this information learned?

- Is the violator identified as such? If so, how broadly is this identification made (with the same set of possible answers as above)?

2.3.3 Action

We identify different aspects of actions within the system, both in general operation and to detect and punish policy violations.

- Is the system (as it operates in the absence of a detected violation) centralized or decentralized?

- Does the system respond to a violation (in the gathering of evidence, judgment, and punishment) in a centralized or decentralized way?
- If violators are punished, is this done (in the terms of Feigenbaum *et al*. [FJW11]) "automatically" or in a "mediated" manner? If there is a mediator, is this an entity that is already part of the system, or is it a specialized external entity?
- To what extent does the functioning of the system rely upon continued participation by, or access to, the violator? For example, is the violator only punished if he continues to interact with the system?

2.3.4 Applicability of this framework

The three broad aspects described above can be used to characterize various approaches to accountability in Computer Science; we do this in the following section. As new accountability systems and approaches are developed, they can also be analyzed within this framework.

In addition to being broadly applicable, we argue that our framework captures essential aspects of accountability at a useful level of granularity. Insofar as "accountability" relates to violations (of policy, law, *etc*.), either actual or possible, the "time" aspect of our framework allows us to compare the relative times at which different systems have effects. The "information" and "action" aspects separate system characteristics that should be compared separately without producing an unmanageably high-dimensional framework.

2.4 Survey of Approaches

There are many different Computer Science approaches to accountability. We discuss a variety of accountability solutions (in Section 2.4.1) and accountability formalizations (in Section 2.4.2) that take on different values along the axes we identified in Section 2.3.

Table 2.1 summarizes our analysis of systems and formalizations that exemplify broader areas of accountability research in Computer Science. The columns correspond to the aspects and sub-aspects of accountability discussed in Sec. 2.3; the reasoning that supports the entries in the table is described in the section of this chapter listed in the leftmost column. Where applicable, the discussion in the text also notes other possible answers or answers that might arise in related but distinct solutions or formalizations. Some systems are defined in general ways that do not enforce a particular categorization for some or all of the columns; we discuss below the range of values they allow or what the most likely categorizations are.

		Time/Goals					Information			Action			
Section	Approach/Paper	Prevention	Detection	Evidence	Judgment	Punishment	Identity Requirements for Participation	Violation Disclosed?	Violator Identified as Such?	Centralization without Violation?	Centralization with Violation?	Punishing Entity?	Requires Ongoing Involvement?
Solutions													
2.4.1	PeerReview [HKD07]		✔	✔	✔		Broad	Broad	Broad	Decent.	Decent.	N/A	No
2.4.1	PEREA [TAKS08]					Med.	Unique	Unique	No	Cent.	Cent.	Internal	No
2.4.1	ASMs [MOR01]	✔	✔	✔			Broad	Broad	Broad	Decent.	Decent.	N/A	No
2.4.1	E-Cash [CHL06]		✔	✔	✔		Unique	Broad	Broad	Cent.	Cent.	N/A	No
2.4.1	iOwe [LSL$^+$11]		✔	✔	✔	Med.	Broad	Broad	Broad	Decent.	Decent.	Internal	No
2.4.1	Buchegger & Boudec [BLB03]					Med.	Broad	Broad	Broad	Decent.	Decent.	Internal	Yes
2.4.1	A2SOCs [FZML02]			✔	✔	Med.	Unique	Broad	Broad	Cent.	Cent.	Int./Ext.	
Formalizations													
2.4.2	Küsters *et al.* [KTV10]		✔	✔	✔		Broad			Decent.	Cent.	N/A	No
2.4.2	Bella & Paulson [BP06]			✔			Limited	Limited	Limited	Cent.	Cent.	N/A	No
2.4.2	Yumerefendi & Case [YC04, YC05, YC07]		✔	✔	✔		Broad	Broad	Broad	Cent.	Cent.	N/A	No
2.4.2	Feigenbaum *et al.* [FJW11]					A/M							
2.4.2	Jagadeesan *et al.* [JJPR09]				✔		Broad	Broad	Broad	Decent.	Decent.	N/A	No
2.4.2	Barth *et al.* [BDMS07]				✔		Broad	Broad	Broad	Cent.	Cent.	N/A	No
2.4.2	Kailar [Kai96]			✔			Broad					N/A	
2.4.2	Backes *et al.* [BDD$^+$06]			✔	✔		Broad					N/A	

Table 2.1: Overview of accountability approaches.

2.4.1 Accountability solutions

PeerReview

The PeerReview system of Haeberlen, Kouznetsov, and Druschel [HKD07] provides a notion of accountability in distributed systems. They take an "accountable" system to be one that "maintains a tamper-evident record that provides non-repudiable evidence of all nodes' actions." In the asynchronous setting considered by Haeberlen *et al.*, the possible violations are not responding to a message (to which a response is prescribed by the protocol) or sending a message that is not prescribed by the protocol. The potential for message delays means that the former cannot be conclusively proved; this gives rise to a distinction between suspicion and certainty, both of which are included in the system.[2]

The design of PeerReview includes, at each node in the network, a detector module that implements the system; this will indicate either suspicion or certainty that another node is violating the protocol. It makes use of a tamper-evident log that each node maintains of its own interactions (and that can be obtained by other nodes as they need it). Taken together, these range over the *detection*, *evidence*, and *judgment* parts of the **Information** aspect of our framework.

Nodes must be identified to participate in a distributed protocol that incorporates PeerReview; for the **Information** aspect of our framework, their identity is made known to a *broad* set of other participants. The security goals for PeerReview include that every node that fails to acknowledge a message is eventually suspected of violating the protocol by *every* node that does follow the protocol, so the disclosure of a violation and the identification of the violator as such are *broad*/*broad*. Under the **Action** aspect, PeerReview is *decentralized* both without and with violations and there is no punishing entity (*not applicable*). If a violator no longer participates, then that node will be viewed as not responding to messages and will be suspected by other nodes; thus, the system *does not* require ongoing involvement on the part of the violator.

Anonymous blacklisting systems

Like e-cash systems, anonymous blacklisting systems allow anonymous participation. In contrast to e-cash, participants in these systems are not identified when they commit a violation; instead, they are blacklisted (*i.e.*, their credentials for participation are revoked) without identifying them. Henry and Goldberg have recently surveyed this space of systems [HG11] and identified three broad subspaces thereof: pseudonym systems, Nymble-like systems, and revocable anonymous credential systems. These provide varying levels of privacy (ranging from pseudonyms to complete anonymity without trusted third parties); however, as the privacy guarantees are strengthened, the feasibility of implementation decreases.

As an exemplar of this class of systems, we will take the PEREA revocable anonymous credential system of Tsang, Au, Kapadia, and Smith [TAKS08]. The user must first register with the system. Depending on the setting, this might require some form of identity; however, the user obtains a credential that can subsequently be used to authenticate herself to the service provider without revealing her identity. The service provider may subsequently revoke the client's credential for any reason, without requiring a trusted third party to do so; this prevents the client from authenticating herself in the future, but it does not reveal anything about her

[2] For example, one system goal is that nodes that ignore messages should eventually be suspected, in perpetuity, by all honest nodes even though they cannot be certain that the ignoring node is in fact ignoring messages.

identity to anyone (nor does it link her various anonymous actions, among other properties).

For the **Time/Goals** aspect, this provides *punishment (mediated)*, because the punishment is carried out by the service provider (in blacklisting the anonymous credential). While this is presumably based on the detection of some violation and the judgment of guilt, PEREA itself is not used to do these things. For our **Information** aspect of accountability, the system *might* require some sort of identity to register, so we categorize this as *unique*. Importantly, however, the violator is not identified as such (although the violation is known by the service provider), so we categorize the last two sub-aspects of this as *unique/none*. The registration and authentication require some *centralized* aspects (regardless of whether there is a violation); the punishing entity is part of the system (*internal*), but punishment does not require the ongoing participation of the violator (*does not*).

Accountable signatures

When digital signatures allow multiple potential signers, either because many individuals could generate the signature or because a valid signature requires multiple signers to generate it, "accountability" has the potential to become an issue in ways that it is not when there is only one potential signer. There are many different approaches to signatures with multiple potential signers; as an exemplar of this area, we take the work by Micali, Ohta, and Reyzin on "accountable-subgroup multisignatures" [MOR01] that explicitly took "accountability" as a goal. Their definition of this goal was

> Accountability means that, without use of trusted third parties, individual signers can be identified from the signed document.

As noted by Micali *et al.*, other approaches with multiple potential signers allow sets of individuals (possibly just a single individual) to generate signatures on behalf of a larger set of individuals in such a way that the individual(s) who produced the signature cannot be identified.

For accountable-subgroup multisignatures as defined by Micali *et al.*, all members of the group run a key-generation protocol once; the signing protocol takes, from each signer, a description of the set of signers and their public keys, the message being signed, and the individual signer's secret key. The signers then produce the signature, which can be verified (when input with a message and a set of purported signers) by anyone. This is secure (and Micali *et al.* describe a secure scheme for accountable-subgroup multisignatures) if an attacker cannot (except with negligible probability) produce a valid signature for a message m where the set S of individuals who purportedly signed m includes an honest participant who did not execute the signing protocol.[3] The set of purported signers provides a guarantee to the verifier of the signature; the signers may not know each other. As Micali *et al*. note [MOR01]:

> Then, assuming that P_2 [one purported signer] has not been corrupted, P_1 [another purported signer] is assured that the verifier will deem the signature valid only if the person whom the verifier knows as P_2 actually participated in the signing protocol on [m and S].

This approach provides accountability through identity; from the perspective of holding policy violators "accountable," it neither judges nor punishes violators. The **Time/Goals** properties that this approach does provide arguably depend on the type of policy under consideration: the security definition provides *prevention* of successful forgeries and *detection* of forgery attempts, while it provides *evidence* of violations that are carried out by someone using his own identity for signing (analogous to, *e.g.*, an officer of an company signing his/her own name to an improper corporate check). With respect to the **Information** aspects of this approach, identity is definitely requited, and a participant's identity is potentially known to a *broad* set of other individuals. We may take two different views of the questions of whether the violation is disclosed and whether the violator is identified as such. Under the first, the violation consists of an attempted forgery. This is detected by the verifier, but the violator might not be identified; we categorize this case as *unique/none*. Under the second view, no forgery is attempted but the (valid) signature on the message indicates that the signers have committed some (non-identity) violation. In this case, the violation is disclosed (because it is embedded in the message), and the violators (the signers) are identified as such, to a broad set of participants; we categorize this as *broad/broad*. For the **Action** aspects of this approach, the signatures can be generated in a *decentralized* way (with or without a violation); this does not incorporate punishment, so we consider the punishing entity to be *not applicable*; finally, this *does not* require ongoing involvement by violators.

E-cash

Pioneered by David Chaum in the early 1980's [Cha82, Cha83], e-cash was designed to have the anonymity and untraceability properties of physical cash: A user should be able to withdraw money from the bank and spend it with a merchant without revealing her identity, and the merchant should be able to deposit the money received into his account without the bank learning how individual users spent their money. Due to the replicable nature of the strings of bits that represent digital money, a primary issue to resolve in realizing e-cash is how to prevent or deter "double-spending," in which users or merchants make and spend (or deposit) multi-

[3] Micali *et al*. [MOR01] discuss issues of adaptive corruption and prove the equivalence of security notions that involve attackers of formally different abilities; those distinctions do not affect our analysis.

ple copies of electronic coins. A solution to this [CFN90] provides consequences for double spending using cryptographic mechanisms that break the anonymity of double spenders. These solutions rely on identity for accountability. Depending on the context of the system, loss of anonymity might or might not be sufficient punishment in and of itself. If not, the system would need to rely on an external mechanism to provide any additional punishment.

Chaum's solutions, and many that grew out of them, have a model in which the bank is a centralized party that checks for double spending. Chaum's initial proposals [Cha82, Cha83] were "on-line," in the sense that the bank must be involved in every transaction in order to prevent double-spending. Chaum, Fiat, and Naor [CFN90] introduced "off-line" e-cash, in which double-spending was not strictly prevented, but the identity of double-spenders would be revealed by the bank after-the-fact, including providing an incontestable proof of the violation (including protecting against a cheating merchant who might try to collude with a customer in order to undetectably allow double spending and/or attempting to frame an innocent customer as a double-spender).

While a complete survey of e-cash schemes is beyond the scope of our work, we note that there have been many proposals that take different approaches and provide different properties, including differences in prevention vs. detection, centralization vs. decentralization, and security vs. efficiency. An interesting example in trading off security and efficiency is Rivest and Shamir's MicroMint [RS97], which is designed so that small-scale fraud will be unprofitable, while large-scale fraud will be detectable.

A recent exemplar of the off-line approach, proposed by Camenisch, Hohenberger, and Lysyanskaya [CHL06], explicitly addresses accountability as a goal to be balanced with privacy, while extending the accountability goals beyond double spending. Specifically, in addition to detection of double spending, their work supports spending limits for each merchant, motivating by concerns that anonymous e-cash can allow undetectable money laundering. A user's anonymity and untraceability is guaranteed as long as she does not violate either policy (double spending or spending limits). Violations can be detected, including determining whether a user or a merchant cheated. When a violation is detected, the bank becomes (mathematically) able to identify the violating user as well as trace the other activities of the violating user. For **Time/Goals**, the system therefore does not rely on prevention. It includes *detection*, *evidence*, and *judgment*. The consequence of detecting cheating is that a user loses her anonymity and untraceability. As noted above, this might be considered to provide sufficient punishment, but in general could need to be supplemented with additional punishment external to the system. Regarding **Information**, identity is an inherent part of the system, but honest parties are guaranteed anonymity. The bank learns about violations and the identity of violators at the time that coins violating the policy are deposited with the bank. For **Action**, the system relies on the bank as a central authority. Users can spend coins at merchants without the involvement of the bank, but users must obtain all coins from the bank and merchants must deposit all coins with the bank, at which time violations can be detected.

iOwe

As we have discussed, many e-cash systems rely on the use of a centralized authority in order to provide their security and accountability properties. Given the decentralized context of peer-to-peer systems, it can be undesirable to rely on a centralized authority for monetary purposes. To this end, a number of decentralized currency systems have been proposed, including [ZCRY03, LSL$^+$11, Nak].

We study the iOwe currency system [LSL$^+$11] as an exemplar of such systems. iOwe allows peers in a decentralized peer-to-peer system to exchange currency backed by system resources. Peers create “iotas” as promises of future work. Iotas can be exchanged for work as payment, or “redeemed” with their originators for work, along the line of standard “IOU”s, but with greater liquidity. iOwe does not prevent double-spending, but addresses it by using signature chains that allow detection by a peer (possibly but not necessarily the originator) seeing the same iota twice, using the two signature chains as a proof of misbehavior, and applying a punishment mechanism that expels detected cheaters and all iotas they issued from the system. Thus, on the **Time/Goals** aspect, iOwe uses *detection*, *evidence*, *judgment*, and *punishment (mediated)*.

iOwe peers have a persistent identity within the system, but these identities need not be tied to external identities and users are not prevented from creating multiple peers (or “Sybils”) within the system. iOwe limits the potential for a user to benefit from double spending using by adding a layer of reputation to the system. Specifically, peers build up trust of other peers by participating in the system (creating, spending, and redeeming iotas), and peers only accept iotas that were both issued by peers they trust and only ever held by peers they trust. In this way, “Sybil” peers are not able to create iotas, because they have not been able to build up trust. A peer therefore can deflect blame for double-spending to another Sybil node it has created, but the peer will be punished by losing the value of any outstanding iotas it holds that were issued by or ever held by the expelled Sybil. In terms of our **Information** aspect, violators are identified by their (weak) identities within the system. The violation is disclosed to any peer that receives the duplicate iota, possibly when returned to the issuer but possibly earlier. In keeping with the decentralized nature of peer-to-peer systems, the **Action** aspect is entirely decentralized. Both the normal operation and the handling of violations are done in a decentralized way, with individual peers able to detect and verify double spending and to implement their own part of the punishment by no longer trusting the violator. (Similar punishment is extended to peers who refuse to redeem iotas they have issued; we omit discussion of that component of the system here.)

Reputation systems

Reputation systems have received much attention in various settings. Even when not explicitly motivated by "accountability," aspects of accountability are closely related to the natural use of these systems. In particular, an action that depends on the reputation of another node could very easily (unless the other node is always indifferent to which action is chosen) be viewed as potential punishment.

As an example of a reputation system, we consider the one for mobile ad-hoc networks proposed by Buchegger and Boudec [BLB03]. Each node i in the network has, for each other node j that it tracks, a trust rating and a reputation rating. The reputation rating, which affects how i behaves towards j, is affected by both i's direct interactions with j and information obtained about j from other nodes (in particular, nodes that either i trusts or that have experiences with j that are similar to i's experiences). If i's view of j is sufficiently bad, then i will avoid routing through j, and i will ignore future route requests from j. (While we view this as punishing j for misbehaving in the routing protocol, Buchegger and Boudec explicitly note that they do *not* punish nodes that give inaccurate reports in the reputation system.) The particular (modified Bayesian) approach to updating reputation is unrelated to the accountability properties of this system.

For the **Time/Goals** aspect of accountability, this provides *punishment (mediated)* through the avoidance of a node in routing and ignorance of its route requests. Arguably, this is also providing a sort of *judgment*, but in an average sense (over many different violations and non-violations); because of that averaging, we will not categorize this as providing *judgment*. (Similarly, this requires that violations are detected, but the reputation system propagates that information instead of actually doing the detecting.) For the **Information** aspect, identity is definitely required[4] and is known to a *broad* set of other participants. Similarly, the point of a reputation system is to identify violators as such (in a fairly broad way), disclosing the violations, so we categorize this as *broad/broad*. For the **Action** aspect, this is *decentralized* both without and with a violation. There are punishing entities—the other nodes in the network, which are *internal*—but punishment *does* rely on the continued participation of the violator (because punishment takes the forms of routing around and ignoring the violator).

[4] Identity is required to be "persistent, unique, and distinct."

A2SOCs

Farkas, Ziegler, Meretei, and Lörincz [FZML02] described an approach (Anonymous and Accountable Self-Organizing Communities, or A2SOCs) to "anonymous accountability" with multiple levels of identities (including pseudonyms). They use both "internal" and "external" notions of accountability and give protocols to provide these. The former notion means that a pseudonym can be "held responsible" for its actions (even under different pseudonyms that are not publicly linked); this is done by the other members of the virtual community. By contrast, "external accountability" is used to mean that the real-world entity connected to the pseudonyms is identified and this real-world identity may be given to, *e.g.*, the police when a real-world crime has been committed. Both the linking of different pseudonyms that belong to the same agent and the release of an agent's real-world identity require broad community agreement (although this assumes that the trusted third party has, as required, deleted keys that it initially used to register pseudonyms).

Within the **Time/Goals** aspect of our framework, we categorize the approach of Farkas *et al.* as providing *evidence* (*e.g.*, through the linking of pseudonyms and providing real-world identities to outside agencies), *judgment* (because the virtual community can, and must, agree to help link different pseudonyms or to reveal a real-world identity), and *punishment (mediated)* (via either the community or the external authorities). Within the **Information** aspect of our framework, identity is initially needed only for registration, which reveals it to the *unique* trusted third party. However, violations are disclosed in a *broad* manner, and (either as a pseudonym or as a real-world identity), violators are identified as such to the *broad* community. The trusted third party means that, in our **Action** aspect, the system is *centralized* both with and without violations. Depending on the level of the violation (and whether pseudonyms are linked or a real-world identity is revealed), the punishing entity can be either internal or external to the community, so we classify this as *both*. The punishment might (*e.g.*, for within-community punishment) or might not (*e.g.*, for banishing a user or for external punishment) require ongoing involvement by the violator, so we do not classify the system in this respect.

2.4.2 Formalizations of accountability

There have been several proposed formalizations of accountability. These, too, take different interpretations of accountability and therefore can apply to different solutions or to different properties of those solutions. We discuss different approaches to formalizing accountability, as well as one that formalizes the related notion of auditability.

Accountability through judging

Küsters, Truderung, and Vogt provide a model for accountability. In an intuitive description of their definitions, a protocol provides accountability if a specified "judge" (who might or might not have an additional role in the protocol) is able to issue "verdicts" about misbehaving participants ("violators," in our terminology) in a way that is both fair and complete. Specifically, the judge should never blame protocol participants who behave honestly (fairness), and, whenever the protocol fails to meet its specified goals, the judge should blame at least some misbehaving participants (completeness). It is left external to this analysis what the consequences for violators should be and how they should be enforced. Thus, for our **Time/Goals** aspect, the model addresses *detection*, *evidence*, and *judgment*. Note that the judge is not required to produce evidence in the form of proofs that others can use, but, if the system provably satisfies fairness, the very existence of the judge's verdict in fact serves as that evidence.

The required verdicts in their model are positive Boolean formulae "built from propositions of the form $\mathsf{dis}(a)$, for an agent a, where $\mathsf{dis}(a)$ is intended to express that a misbehaved." Thus, for our **Information** aspect, this method relies on participating agents to have identities in whatever system is being analyzed. They do, however, allow for the possibility of verdicts that do not identify individual violators, by allowing disjuncts. In this sense, a violator might or might not be explicitly identified as such. (However, they point out that individual accountability, in which individual violators are identified, is highly desirable in practice to deter parties from misbehaving.) Violations are disclosed at least to the judge, as well as to any parties to whom the judge shares the verdict.

For **Action**, the definition requires the existence of a judge to provide the required verdicts, suggesting a centralized system. However, one could imagine applying their definitions to a decentralized system such as iOwe [LSL$^+$11], described in Sec. 2.4.1, where different parties can act as judges for different violations, for example, proving results such as: if party P double-spends an iota, then another party Q can act as a judge and hold P accountable.

Connecting actions to identities

Bella and Paulson [BP06] have formalized properties of two particular "accountability protocols" and verified these using the Isabelle tool; these protocols connect actions to identities.[5] The particular protocols that they studied were for non-repudiation [ZG96] and certified email [AGHP02]; here, we focus not on these

[5] This broad approach is also embodied in the Accountable Internet Protocol [ABF$^+$08] of Andersen *et al.* They identify the lack of accountability with the fact that "the Internet architecture has no fundamental ability to associate an action with the responsible entity."

protocols individually but on the class of accountability protocols that they exemplify (*i.e.*, that corresponds to the properties identified in [BP06]).

In the approach of Bella and Paulson,

> [a]n accountability protocol gives agents lasting evidence, typically digitally signed, about actions performed by his peer.

They note that many authentication protocols prove the involvement of a participant to one other participant (*e.g.*, via an encrypted nonce), but that these do not provide evidence that is suitable to take to a third party. Indeed, one of the two goals they identify for accountability protocols is that

> an agent is given evidence sufficient to convince a third party of his peer participation in the protocol.

(The other goal is a notion of fairness in which either both participants receive, or neither participant receives, evidence about the other's participation.) Bella and Paulson explicitly note that judging is left to humans who are not modeled in their analysis. For the **Time/Goals** aspect of our framework, we thus say that their approach (and the accountability solutions that fall within their model) provide *evidence* but not other parts of this aspect.

Both protocols considered by Bella and Paulson involve two regular participants a trusted third party; all three of these parties learn the identities of the participants, but those identities are not broadcast further. For the **Information** aspect of our framework, we will thus say that identity is required in a *limited* sense. Violations (captured in the protocol exchanges, not attempts to circumvent the protocols themselves) are revealed through the evidence that the protocols provide, and the violators are identified as such; because this information is provided to the other participant but not broadcast, we say that the other two parts of this aspect are *limited* and *limited*.

For the **Action** aspect of our framework, the trusted third party is required regardless of whether there is a violation, so we identify the Bella–Paulson approach as *centralized/centralized*. There is no punishing entity (*not applicable*), and there is no requirement that a violator continue to participate in the protocol (*does not*).

Accountability for network services

Yumerefendi and Chase [YC04] have outlined an approach to accountability for network services that respond to client requests. In so doing, they have articulated a definition of accountable services, described a general method for achieving this, and sketched its application to three different settings; they subsequently used this approach in a more detailed description of network storage with accountability [YC07]. Here, we are interested in their general definition and method, which may be applied beyond the settings they noted.

Yumerefendi and Chase say that accountable systems should have actions that are: provable and non-repudiable; verifiable (with the ability to prove misbehavior to any third party); and tamper-evident (regarding the states of the system). (These foster the goal they identified in related work [YC05], which argued for accountability as a design goal, of "assign[ing] responsibility for states and actions[.]") Considering the **Time/Goals** aspect of our framework, this means that the systems provide both *detection* and *evidence*. It is envisioned that auditors are involved (who might examine the evidence that the service has behaved correctly); while these might arguably be viewed as lying outside of this system, we will include them (clients may verify that the service correctly maintained its state) and so also view this approach as providing *judgment*. (The subsequent extension of this approach to network storage [YC07] reinforces audit as an important component of this approach and the fact that any participant may act as an auditor.)

This approach to accountability has systems publish signed, non-repudiable digests of their internal states. As Yumerefendi and Chase observe, client actions (and potentially identities) may be incorporated into the services' states, so, under our **Information** aspect, we categorize this as requiring identity that may be revealed (or at least checkable by) a *broad* set of participants. Violations are also identified to a *broad* set, and violators are likewise identified to a *broad* set of participants.

For the **Action** aspect of our framework, regardless of whether there is a violation, the service plays a central role in providing digests and proofs of its correct behavior, so we identify this as *centralized/centralized*. There is no punishing entity (*not applicable*). Although the service publishes digests of its state, it needs to respond to later requests to provide proofs of its correct behavior. Insofar as the system is aiming to provide proofs of correct behavior, this requires ongoing participation; however, if others will be at least suspicious if no proof is forthcoming, then this *does not* require the ongoing participation of the violator.

Accountability in terms of punishment

Feigenbaum, Jaggard, and Wright [FJW11] give an abstract definition of accountability in terms of punishment and then capture this formally in terms of traces and utility functions. Their definition of accountability includes punishment that is "automatic" in the sense that it is not meted out in conscious response to a violation (which would be "mediated" punishment as noted above). Coupled with this, they also explicitly do not require identity, and they note the possibility of punishment occurring (thus providing accountability) without anyone other than the violator knowing that a violation occurred.

The Feigenbaum *et al*. framework can capture systems with, *e.g.*, varying identity requirements, so the classifications that we have been using for the **Information** and **Action** aspects of accountability are not at all determined without considering a particular system. For the **Time/Goals** aspect, this framework addresses only *punishment (automatic and mediated)* and no other sub-aspects.

Accountability through audit

As one exemplar of accountability through the use of auditing, we consider the work of Jagadeesan, Jeffrey, Pitcher, and Riely [JJPR09], who describe a formal operational model for distributed systems with a notion of accountability that is obtained through auditing. In particular, the auditor(s) in a system may "blame" a set of participants for a violation, *i.e.*, name the members of that set as potential violators. This gives rise to multiple desiderata (such as whether everyone blamed is a violator and whether all non-violators are able to ensure that they are not blamed) for the audit system; these are treated as accountability properties, but they do not change the underlying approach of blaming (sets of) individuals for violations.

We identify the blaming of individuals in the Jagadeesan *et al*. model with *judgment* within the **Time/Goals** aspect of our framework. (While the auditors rely upon evidence to make their judgments, the notion of accountability captured by this framework seems to fit much more with the judgment itself.) Because sets of individuals are blamed using their identities, some sort of identity is required to participate; while this might not be broadcast throughout the system, there are no restrictions on it, so, within the **Information** aspect, we say that the identity required to participate is *broad*. Violations are disclosed, and violators are identified as such, in similar ways, so we identify those parts of this aspect as *broad/broad*. While auditors are trusted in this system, they do not have a global view (*i.e.*, they interact with the system as participants); for the **Action** aspect of accountability, we thus say that the centralization without/with a violation is *decentralized/decentralized*. This framework is not concerned with punishment, so the punishing entity is *not applicable*. Judgment can be made without the presence of the violator, so this system *does not* require the ongoing participation of a violator.

As a second exemplar of accountability through audit, we note the work of Barth, Datta, Mitchell, and Sundaram [BDMS07], who defined a logic for utility and privacy that they applied to models of business practices (such as healthcare systems). In their application to healthcare systems, agents in the system are responsible for things like tagging messages (*e.g.*, to ensure that sensitive health information is not forwarded to the agents responsible for scheduling patient appointments). Barth *et al*. say that an agent is accountable for a policy violation if the agent did an action that occurred before the violation (from some perspective on the system's behavior) and also did not fulfil his responsibilities. They then give an algorithm to identify accountable agents (via communication logs). While an "accountable agent" might not be the cause of the violation in question, this can be determined by a human auditor, who can repeat the process until the agent who caused the violation is identified.

Within the **Time/Goals** aspect of our framework, this approach has elements of *detection*, *evidence*, and *judgment*. (The last is as with the work of Jagadeesan *et al.*; here we also include the first two elements because *evidence* is provided through the tagging requirements, and *detection* is provided by the notion of "suspicious" events, which can be used to find incorrectly tagged messages.) With respect to identities, the disclosure of violations, and the identification of violators, this approach is similar to that of Jagadeesan *et al.*; in the **Information** aspect of our system, we thus say that the approach of Barth *et al.* is *broad/broad/broad*. Here, the auditing engine (which is used even in the absence of a violation) and the human auditors (who determines whether an agent is the cause of a violation) appear to be centralized, so we say that, in the **Action** aspect, centralization without/with a violation is again *centralized/centralized*. Similarly to the Jagadeesan *et al*. approach, the punishing entity is *not applicable*, and the system *does not* require the ongoing participation of a violator.

Analyzing accountability in logical frameworks

Kailar [Kai96] developed a logical framework for analyzing accountability in communication protocols and considered sample applications to electronic-commerce protocols. He defined accountability as

> the property whereby the association of a unique originator with an object or action can be proved to a third party (*i.e.*, a party who is different from the originator and the prover).

Accountability goals in a protocol might include that a customer can prove that a business agreed to sell a particular item at a particular price or that the business can prove it provided that item to the customer. Once these goals are formalized for a particular protocol, and the message contents are formalized, Kailar's framework can be used to derive information about who can prove what to whom. These results can then be compared against the original accountability goals.

Within the **Time/Goals** aspect of our framework, we categorize Kailar's approach as providing *evidence* because the analysis of a particular protocol can determine whether an association between an agent and an action/object can be proved to a third party (although it is the underlying protocol itself that actually provides the evidence). Considering the **Information** aspect of our framework, the use of identities are inherent in Kailar's definition of accountability. It is most natural for these identities to become broadly known through participation in a protocol (*e.g.*, when signing messages that might be seen by any agent on the network), and there is no restriction on their distribution built in to the logical framework, so we classify this as *broad*. The disclosure of violations and the identification of violators as such might vary across protocols analyzed using Kailar's framework, so we do not classify it in these respects. Similarly, the different components of the **Action** aspect of our framework are not relevant at this level of abstraction, so we do not classify Kailar's framework with respect to those.

Backes, Datta, Derek, Mitchell, and Turuani [BDD$^+$06] used a protocol logic (similar to one originally used for authentication) to prove properties of contract-signing protocols. One of these properties was accountability, which they defined as follows:

> Accountability means that if one of the parties gets cheated as a result of [the trusted third party] $\hat{T}$'s misbehavior, that it will be able to hold $\hat{T}$ accountable. More precisely, at the end of every run where an agent gets cheated, its trace together with a contract of the other party should provide non-repudiable evidence that $\hat{T}$ misbehaved.

As an example of such evidence, Backes *et al.* give the example of terms that can be used (in the logic they define) to derive a term that captures the dishonesty of the trusted third party.

Considering the approach of Backes *et al.* within the **Time/Goals** aspect of our framework, we say that this is focused on determining whether the protocols they study provide *evidence*. (Because this is defined in terms of being able to derive a judgment of dishonesty using the protocol logic, this also has aspects of *judgment*.) Within the **Information** aspect of our framework, we say that this requires *broad* knowledge of identity (because this concerns the behavior of trusted third parties); the disclosure of the violation and the identification of the violator as such are not determined by the Backes *et al.* framework (although these would likely be *broad*). The presence of the trusted third party means that the contract-signing protocols have a centralized aspect to them, but this requirement is not imposed on the accountability analysis (although it seems that the proof of dishonesty would typically be derived without centralization). There is no punishing entity, and ongoing involvement by the dishonest trusted third party is also not determined.

Chapter 3

Structural Cloud Audits that Protect Private Information

3.1 Introduction

Cloud computing and cloud storage now play a central role in the daily lives of individuals and businesses. For example, more than a billion people use Gmail and Facebook to create, share, and store personal data, 20% of all organizations use the commercially available cloud-storage services provided both by established vendors and by cloud-storage start-ups [But13a, But13b], and programs run on Amazon EC2 and Microsoft Azure perform essential functions.

As people and organizations perform more and more critical tasks "in the cloud," reliability of cloud-service providers grows in importance. It is natural for cloud-service providers to use redundancy to achieve reliability. For example, a provider may replicate critical state in two data centers. If the two data centers use the same power supply, however, then a power outage will cause them to fail simultaneously; replication *per se* does not, therefore, enable the cloud-service provider to make strong reliability guarantees to its users. This is not merely a hypothetical problem: Although Amazon EC2 uses redundant data storage in order to boost reliability, a lightning storm in northern Virginia took out both the main power supply and the backup generator that powered all of Amazon's data centers in the region [Ore12]. The lack of power not only disabled EC2 service in the area but also disabled Netflix, Instagram, Pinterest, Heroku, and other services that relied heavily on EC2. This type of dependence on common components is a pervasive source of vulnerability in cloud services that believe (erroneously) that they have significantly reduced vulnerability by employing simple redundancy.

Zhai *et al.* [ZWX$^+$] propose *structural-reliability auditing* as a systematic way to discover and quantify vulnerabilities that result from common infrastructural dependencies. To use their SRA system, a cloud-service provider proceeds in three stages: (1) It collects from all of its infrastructure providers (*e.g.*, ISPs, power companies, and lower-level cloud providers) a comprehensive inventory of infrastructure components and their dependencies; (2) it constructs a service-wide fault tree; and (3) using fault-tree analysis, it estimates the likelihood that critical sets of components will cause an outage of the service. Prototype implementation and testing presented in [ZWX$^+$] indicates that the SRA approach to evaluation of cloud-service reliability can be practical.

A potential barrier to adoption of SRA is the sensitive nature of both its input and its output. Infrastructure providers justifiably regard the structure of their systems, including the components and the dependencies among them, as proprietary information. They may be unwilling to disclose this information to a customer so that the latter can improve its reliability guarantees to *its* customers. Fault trees and failure-probability estimates computed by the SRA are also proprietary and potentially damaging (to the cloud-service provider as well as the infrastructure providers). All of the parties to SRA computation thus have an incentive not to participate. On the other hand, they have a countervailing incentive *to* participate: Each party stands to lose reputation (and customers) if it promises more reliability than it can actually deliver because it is unaware of common dependencies in its supposedly redundant infrastructure.

In this chapter, we investigate the use of *secure multi-party computation* (SMPC) to perform SRA computations in a privacy-preserving manner. SRA computation is a novel and challenging application of SMPC, which has not often been used for graph computations[1] (or, more generally, for computations on complex, linked data structures). We introduce a novel data-partitioning technique in order to achieve acceptable running times for SMPC even on large inputs; this approach to SMPC efficiency may be applicable in other contexts. Our preliminary experiments indicate that our P-SRA (for “private structural-reliability auditing”) approach can be practical.

3.2 Related Work

3.2.1 Secure Multi-Party Computation

The study of secure multi-party computation (SMPC) began with the seminal papers of Yao [Yao82, Yao86] and has been pursued vigorously by the cryptographic-theory community for more than 30 years. SMPC allows n parties $P_1,\dots,P_n$ that hold private inputs $x_1,\dots,x_n$ to compute $y = f(x_1,\dots,x_n)$ in such a way that they all learn y but no P_i learns anything about x_j, $i \neq j$, except what is logically implied by the result y

[1] A notable exception is the work of Gupta *et al.* [GSP$^+$12] on SMPC for interdomain routing.

and the particular input x_i that he already knew. Typically, the *input providers* $P_1, \ldots, P_n$ wish not only to compute y in a privacy-preserving manner but also to do so using a protocol in which they all play equivalent roles; in particular, they don't want simply to send the x_i's to one trusted party that can compute y and send it to all of them. Natural applications include voting, survey computation, and set operations. One of the crowning achievements of cryptographic theory is that such privacy-preserving protocols can be obtained for any function f, provided one is willing to make some reasonable assumptions, *e.g.*, that certain cryptographic primitives are secure or that some fraction of the P_i's do not cheat (*i.e.*, that they follow the protocol scrupulously).

Many SMPC protocols have the following structure: In the first round, each P_i splits its input x_i into *shares*, using a *secret-sharing scheme*, and sends one share to each P_j; the privacy-preserving properties of secret sharing guarantee that the shares do not reveal x_i to the other parties (or even to coalitions of other parties, provided that the coalitions are not too large). The parties then execute a multi-round protocol to compute shares of y; the protocol ensures that the shares of intermediate results computed in each round also do not reveal x_i. In the last round, the parties broadcast their shares of y so that all of them can reconstruct the result. Alternatively, they may send the shares of y to an outside entity (resp., to a subset of the P_j's) if none (resp., only a subset) of the P_j's is supposed to learn the result. The maximum size of a coalition of cheating parties that the protocol must be able to thwart and the "adversarial model," *i.e.*, the capabilities and resources available to the cheaters, determine which secret-sharing scheme the P_i's should use. Because secret-sharing-based SMPC is common (and for ease of exposition), we will refer to parties' "sharing" or "splitting" their inputs. Note, however, that some SMPC protocols use other techniques to encode inputs and preserve privacy in multi-round computation.

The past decade has seen great progress on general-purpose platforms for SMPC, including Fairplay [MNPS04], FairplayMP [BDNP08], SEPIA [BSMD10], VIFF [DGKN09], and Tasty [HKS$^+$10]. For our prototype implementation of P-SRA, we use the Sharemind SecreC platform [BK13]. Thorough comparison of SMPC platforms is beyond the scope of our work, but we note briefly the properties of SecreC that make it a good choice in this context. Because it has a C-like programming language and optimizing compiler, assembler, and virtual machine, programmers can more easily write efficient programs with SecreC than with most of the other SMPC tools. Scalability to large numbers of input providers and reliable predictions of running times of programs are better in SecreC than in other SMPC environments. SecreC makes it easy for programs to use both private data (known to only one input provider) and public data (known to all parties to the computation) in the same program – something that is useful in our reliability-auditing context but is not provided by all

SMPC platforms. On the downside, SecreC is not especially flexible or easily configurable.

3.2.2 Cloud Reliability

The case for "audits" as a method of achieving reliability in cloud services was originally put forth by Shah *et al.* [SBMS07], who advocated both *internal* and *external* auditing. Internal audits use information about the structure and operational procedures of a cloud-service provider to estimate the likelihood that the provider can live up to its service-level agreements. To the best of our knowledge, the first substantial effort to design and implement a general-purpose internal-auditing system that receives the structural and operational information directly from the cloud-service providers is the recent work of Zhai *et al.* [ZWX^+]; the privacy issue and the possibility of addressing it with SMPC were raised in [ZWX^+] but were not developed in detail.[2] External audits use samples of the cloud-service output provided by third parties through externally available interfaces to evaluate the quality of service; they have been investigated extensively, *e.g.*, in [SSB08, WCW^+13, WRLL10, WWRL10, WWR^+11, YJ12]. Bleikertz *et al.* [BSP^+10] present a cloud-auditing system that Shah *et al.* [SBMS07] would probably classify as "internal," because it uses structural and operational information about the cloud services to estimate reliability rather than using sampled output, but it obtains that structural and operational information through external interfaces rather than receiving it directly from the cloud-service providers.

In addition to auditing, technical approaches that researchers have taken to cloud reliability include *diagnosis*, the purpose of which is to discover the causes of failures after they occur and, in some cases, to mitigate their effects, *accountability*, the purpose of which is to place blame for a failure after it occurs, and *fault tolerance*. Further discussion of these approaches and pointers to key references can be found in Section 6 of [ZWX^+].

3.2.3 Fault Trees

Fault-tree analysis [VGRH81] is a deductive-reasoning technique in which an undesirable event in a system is represented as a boolean combination of simpler or "lower-level" events. Each node in a fault tree[3] represents either an event or a logic gate. Event nodes depict failure events in the system, and logic gates depict the

[2] Concurrently with this work, Zhai, Chen, and Ford [ZCWF13] also explored the problem of privacy in cloud-reliability analysis, with a different goal of recommending good cloud configurations in a privacy-preserving manner. Their work was done independently of ours and differs from ours both in its target problem and in its technical approach. Briefly, Zhai *et al.* [ZCWF13] simply compute the set of components that are common to two cloud-service systems, while our P-SRA system involves more participants and a richer set of outputs. The main technical tool in [ZCWF13] is privacy-preserving set intersection, whereas P-SRA does a variety of privacy-preserving distributed computations using a general-purpose SMPC platform.

[3] What are called "fault trees" in the literature are not, in general, trees but rather DAGs. Because it is standard and widely used, we adopt the term "fault tree" in this chapter.

logical relationships among these events. The links of a fault tree illustrate the dependencies among failure events. The root node represents a "top event" that is the specific undesirable state that this tree is designed to analyze. The leaf nodes are "basic events," *i.e.*, failures that may trigger the top event; in order to use a fault tree, one must be able to assess (at least approximately accurately) the probabilities of these basic failures. Figure 3.4 is an example of a fault tree.

Fault-tree analysis has been applied very widely, *e.g.*, in aerospace, nuclear power, and even social services [EI00], but, to the best of our knowledge, was first used for cloud-service-reliability auditing by Zhai *et al.* [ZWX$^+$]. It is an appropriate technique in this context for at least two reasons. First, the architectures of many cloud platforms can be accurately represented as leveled DAGs; therefore, potential cloud-service failures are naturally modeled by fault trees. Second, to construct a fault tree, one must uncover and represent the dependency relationships among components in a cloud system, and this inventory of dependencies is itself helpful in identifying potential failures (especially correlated failures).

3.3 Problem Formulation

In order to specify in full detail the goals of our P-SRA system and how it achieves them, we start with a brief explanation of the SRA system of Zhai *et al.* [ZWX$^+$]. Here and throughout the rest of this chapter, a *failure set* (FS) is a set of components whose simultaneous failure results in cloud-service outage.[4] For example, the main and backup power supplies in the Amazon EC2 example described in Section 3.1 are an FS. A *minimal FS* is an FS that contains no proper subset that is also an FS.

The first necessary and nontrivial task of SRA is *data acquisition*. SRA's *data-acquisition unit* (DAU) collects from a target cloud-service provider *S* and all of the service providers that *S* depends on the details of network dependencies, hardware dependencies, and software dependencies, as well as the failure probability of each component. Using this inventory of components and the dependencies among them, SRA builds a model of *S* and the services on which it depends in the form of a *dependency graph*. Zhai *et al.* [ZWX$^+$] assume that the dependency graph of a cloud service is a leveled DAG, and we also make this assumption; we are aware that it is a simplification (see Section 6.2), but it is an important first step. There are many potential technical and administrative challenges involved in modeling cloud components, discovering their dependencies, and assigning realistic failure probabilities; in particular, all cloud-service providers that participate in an SRA computation must agree on a taxonomy of components and types of dependencies. We defer to Subsection 3.2 of Zhai *et al.* [ZWX$^+$] for discussion of these challenges and for details about data

[4] These are called *cut sets* in the fault-tree literature.

acquisition and dependency-graph construction in SRA. Here, we merely assume that cloud providers have *some* usable modeling and dependency-gathering infrastructure.

The next step in SRA is fault-tree analysis for the target cloud service *S*. Ideally, the output of this step is a complete set of minimal FSes for *S*. Note that an outage may occur because multiple entities that *S* relied on for redundancy had a common dependency on a set of components that failed; so accurate reporting of all minimal FSes requires information about all of the other service providers (*e.g.*, ISPs, power suppliers, and lower-level cloud services) that *S* uses. For some of these other services, the information might be publicly available (or, in any case, available to *S*) and thus not require the other service to participate in the computation; for example, SRA makes the simplifying assumption that it can obtain the information it needs about the ISPs and power suppliers that *S* uses without their participation, and we continue with that assumption in P-SRA. In other cases, participation in the SRA computation by other service providers is required; this is true, for example, of lower-level and peer cloud services on which *S* depends. If the ideal of reporting all minimal FSes is unattainable because it is too time-consuming, then SRA may produce a collection of (not necessarily minimal) FSes using a *failure-sampling algorithm*; this algorithm uses both the dependency graph and the individual components' failure probabilities.

Figure 3.3 depicts a simple dependency graph. Figure 3.4 depicts a corresponding fault tree. The semantics of an OR gate in the fault tree are that, if any input fails, the output of the gate is "fail." For an AND gate, only if all of the inputs fail does the gate output "fail." So Data Center #1 fails if Power #1 fails or if both Router #1 and Router #2 fail. Note that the logic-gate nodes in Figure 3.4 cannot be inferred from Figure 3.3; SRA collects additional information during its data-acquisition phase that is needed for fault-tree construction. The minimal FSes for Cloud Service #1 are $\{$Data Center #1, Data Center #2$\}$, $\{$Router #1, Router #2$\}$, $\{$Power #1, Power #2$\}$, $\{$Power #1, Data Center #2$\}$, and $\{$Data Center #1, Power #2$\}$.

The goal of P-SRA is to perform structural-reliability auditing in a privacy-preserving manner; to do this, we must modify all phases of SRA – data acquisition, fault-tree construction and analysis, and delivery of output. Our basic approach to the first two is to use SMPC. Instead of sending their data to one machine that integrates them and performs fault-tree analysis, P-SRA participants split their data into shares and perform fault-tree construction and analysis in a distributed, privacy-preserving fashion. However, the output of this computation cannot simply be a comprehensive list of *S*'s minimal FSes, as it was in SRA, because these sets may contain infrastructural components that are used only by other service providers (*i.e.*, not by *S*). So the first technical challenge in the design of P-SRA is to specify SMPC outputs that reveal to *S* the components of its own infrastructure that could cause an outage while not revealing private information about other service providers' infrastructure. The second technical challenge is to reduce the size of the data sets that are input to the SMPC; the complete dependency graph of a cloud-service provider could have millions of nodes, which is more than current SMPC technology can handle, even in an off-line procedure like reliability auditing. P-SRA deals with this challenge by requiring each service provider that participates in the SMPC to partition its components into those that are known to be "private" and those that might be shared with other participants. For example, if the storage devices in a data center owned and operated by *S* are not accessible by anyone outside of *S*, then their failure cannot cause any service other than *S* to fail – they can be marked "private" by *S* and not entered individually into the SMPC. Rather, *S* can collapse certain "private" subgraphs of its dependency graph into single nodes, treat each such node as a "component" when entering its input to the SMPC, and perform SRA-style fault-tree analysis on the private subgraph locally. We refer to this data-partitioning technique as *subgraph abstraction*. Finally, P-SRA must provide useful, privacy-preserving output to cloud-service users as well as cloud-service providers. These three technical challenges are addressed in detail in Section 3.4.

In Section 3.5, we present a P-SRA prototype implemented on the Sharemind SecreC platform. The properties of this platform guarantee security in the semi-honest (or honest-but-curious) adversarial model. See the beginning of Section 3.5 for a more detailed explanation of Sharemind's computational model and adversarial model.

3.4 System Design

3.4.1 System Overview

There are three types of participants in the P-SRA system: the P-SRA host, cloud-service providers, and cloud-service users; see Figure 3.1. The input supplied by each cloud-service provider is its topology information; this is private information and cannot be revealed to any other participants. The input supplied by the P-SRA host is the SMPC protocol. The inputs supplied by the cloud-service users are the set of cloud-service providers that they use or plan to use. The inputs of the P-SRA host and the cloud-service users are not private.

The P-SRA host consists of two modules. One is the SMPC execution unit (SMPC), which is responsible for execution of the SMPC protocol. The other is the coordination unit, which is responsible for establishing the SMPC protocol and coordinating the communication among the P-SRA host and the other participants.

Each cloud-service provider installs and controls a **P-SRA client** that processes local data and communi-

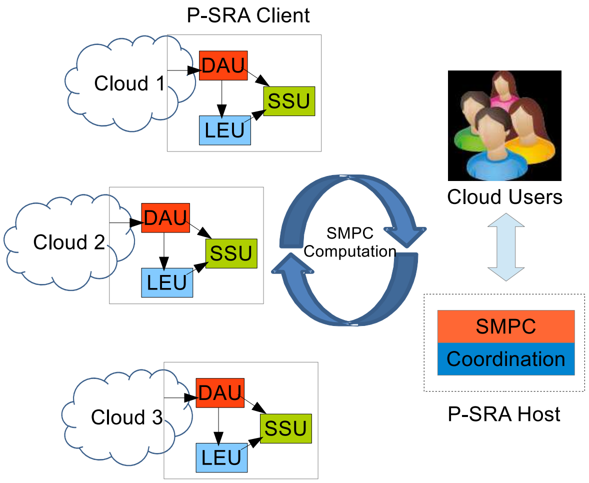

Figure 3.1: System Overview

cates with the P-SRA host. The P-SRA client consists of three modules: the Data-Acquisition Unit (DAU), the Secret-Sharing Unit (SSU), and the Local-Execution Unit (LEU). The DAU collects component and dependency information from the cloud-service provider and stores it in a local database. The SSU (1) abstracts the dependency information of "private" components in order to reduce the size of the input to the SMPC, (2) splits the dependency information into secret shares, and (3) connects to the P-SRA host and the SSUs of other cloud-service providers to execute the SMPC. The LEU performs local structural-reliability analysis within each "abstracted" macro-component.

Step 1: Privacy-preserving dependency acquisition: The DAU of the P-SRA client in each cloud-service provider S collects as much dependency information as possible from S, including network dependencies, hardware dependencies, software dependencies, and component-failure probabilities. The DAU stores this information in a local database. Because the P-SRA client is fully controlled by S, and the DAU does not communicate with any other cloud-service providers or the P-SRA host, there is no risk that private information will leak through the DAU.

Step 2: Subgraph abstraction. After data acquisition by the DAU, the SSU processes the dependency information and creates the macro-components to generate the SMPC input according to some abstraction policy. For instance, if cloud-service provider S_1 uses cloud-service provider S_2 as a lower-level infrastructure provider, S_1 can abstract S_2 as a macro-component that its services depend on. The SSU treats macro-components, the number of which is much smaller than total number of components in a cloud-service provider, as individual inputs to the SMPC. We leave the choices of abstraction policies to the cloud-service providers, which can tailor the policies based on the features of their architectures. However, we provide a standard example of subgraph abstraction in Subsection 3.4.3.

Step 3: SMPC protocol execution and local computation. After the subgraph-abstraction step, the SSUs of the cloud-service providers and the P-SRA host execute the SMPC protocol. The SSU of each cloud-service provider first adds some randomness to conceal statistical information about the input (without changing the output) and splits the randomized input into secret shares. It then establishes connections with the SSUs of other providers and the P-SRA host to execute the SMPC protocol, which identifies common dependency, performs fault-tree analysis, and computes reliability measures in a privacy-preserving manner. Meanwhile, the SSU passes the dependency graphs of the macro-components to the LEU, which performs local computation. The LEU mainly performs fault-tree analysis to obtain minimal FSes of the macro-components. After both the SSU and the LEU finish their execution, the SSU combines the results of the SMPC protocol and local computation to generate the comprehensive outputs for the cloud-service providers and users.

Step 4: Privacy-preserving output delivery. The output of the P-SRA system should satisfy two requirements: preserving privacy of the cloud-service providers and illustrating reliability risk caused by correlated failure. The SRA system of Zhai *et al.* [ZWX$^+$] fully reveals all minimal FSes; P-SRA cannot do this, because the full specification of all minimal FSes may compromise the privacy of cloud-service providers. Although P-SRA is flexible in that cloud-service providers can specify the output sets that are most appropriate for them, we recommend some sets of benchmark outputs for cloud-service providers and users. For cloud-service providers, we recommend *common dependency* and *partial failure sets*. For cloud-service users, we recommend *common dependency ratio*, *failure probabilities of relevant cloud services*, and a small set of *top-ranked FSes*. All the outputs are delivered by an SMPC protocol in a privacy-preserving manner. We discuss these recommended outputs in Subsection 3.4.5.

3.4.2 Privacy-preserving Data Acquisition

The DAU of each cloud-service provider collects as much information as possible about the components and dependencies of this provider and then stores the information in a local database for later use by the P-SRA's other modules. The DAU can collect network dependencies, hardware dependencies, software dependencies, and failure probabilities of each component. For network dependencies, it collects information about a variety of components in the cloud structure including servers, racks, switches, aggregation switches, routers, and power stations, as well as the connections between these components within the cloud infrastructure and from the cloud infrastructure to the Internet. For hardware dependencies, the DAU inventories the CPUs, network cards, memory, disks, and drivers, and collects product information about each piece of hardware, including vendor, machine life, model number, and uptime. For software dependencies, the DAU analyzes the cloud-service provider's software stacks to determine the correlations between programs within the applications running on servers and the calls and libraries used by these programs. Failure probabilities can be obtained via a variety of methods, including examining the warranty documents of a vendor or searching online based on hardware type and serial number.

The dependency information can be encoded in XML files to store in the local databases of the cloud providers. We use the *topology-path form* to store graph information. The definition of the topology-path form and our reasons for choosing it are given in Subsection 3.4.4.

3.4.3 Subgraph Abstraction

Recall that a macro-component is an abstracted (or virtual) node in the dependency graph of a cloud-service provider that can be considered an atomic unit for the purpose of SMPC protocol execution. Creating macro-components allows us to reduce the input-set size to something that is feasible for SMPC execution. A subgraph H of the full dependency graph of a cloud-service provider S should have two properties in order to be eligible for abstraction as a macro-component. First, all components in H must be used only by S; intuitively, this is a "private" part of S's infrastructure. Second, for any two components v and w in H, the dependency information of v with respect to components outside of H is identical to that of w; that is, if v has a dependency relationship (as computed by the DAU) with a component y outside of H, then w has exactly the same dependency relationship with y. (Note that y may be inside or outside of S.) Abstraction of a subgraph that does not satisfy these properties would destroy dependency information that is needed for structural-reliability auditing.

Recall that different cloud-service providers may wish to use different abstraction policies. That is, we do not *require* that all subgraphs that satisfy the two properties given above be abstracted – some providers may wish to use a more stringent definition of a macro-component.

Suppose that G is the full dependency graph of cloud-service provider S and that G contains macro-component H. To transform G into a smaller graph G' via subgraph abstraction of H, S "collapses" H to a single node in G'; that is, S replaces H with a single node, say h, and, for every node y in G but not in H, replaces all dependency relationships in G of the form (w, y, ℓ), where w was a node in H and ℓ is a label that describes the nature of the dependency relationship between w and y, with a single dependency relationship

(h, y, ℓ) in G'. (Note that there will, in general, be many nodes y that have no dependency relationships with nodes in H.) Of course, there may be more than one subgraph H that is abstracted before the reduced dependency graph is entered into the SMPC. After S receives the results of the SMPC, it combines them with the results of local fault-tree analysis of the macro-components H. For example, if F is an FS of G', $h \in F$, and f is an FS of H, then $(F - \{h\}) \cup f$ is an FS of G.

As promised, we now provide a standard example in order to illustrate the abstraction process. In this example, the SSU creates a macro-component to represent all of the components in a data center. In most cloud structures, the data centers are eligible for subgraph abstract. First, all the nodes in the data centers are owned and used by exactly one cloud-service provider. Second, all nodes in a data center communicate with the rest of the world only through the data-center gateways; they therefore have identical dependency relationships with components outside of the data center.

Figures 3.2 and 3.3 illustrate this process. Suppose that Figure 3.2 is the full dependency graph of cloud-service provider C_1, which contains a storage-cloud service. C_1's users' files are stored in server S_2, with two backup copies stored in server S_5 and S_7. The components inside the red box belong to a data center DC_1, which has the two properties required for asbtraction. After abstracting both DC_1 and another data-center subgraph, the SSU obtains Figure 3.3 as the input to the SMPC. After the abstraction process, the SSU executes the SMPC protocol with the abstracted inputs and passes the dependency information within the macro-components (such as the red box in Figure 3.2 for DC_1) to the LEU for local computation.

Because the number of components in a data center is often huge, this kind of abstraction can be a crucial step toward the feasibility of SMPC.

3.4.4 SMPC and Local Computation

SSU Protocol:

Fault-tree construction: Recall that a fault tree contains two kinds of information: dependency information about components (events and links) and logical relationships among components (represented as logic gates). As we said in Subsection 3.4.2, we use the *topology-path form* to store dependency information. That is, we represent a leveled DAG as a set of (directed) paths in which the first node of each path is the root node of the leveled DAG, the last node is one of the leaf nodes of the leveled DAG, and the other nodes form a path from the root to the leaf. Figure 3.5 depicts the topology-path form of the dependency graph in Figure 3.3. The topology-path form of a DAG can, in the worst case, be exponentially larger than the DAG itself; thus, subgraph abstraction is crucially important, because we need to start with modest-sized DAGs.

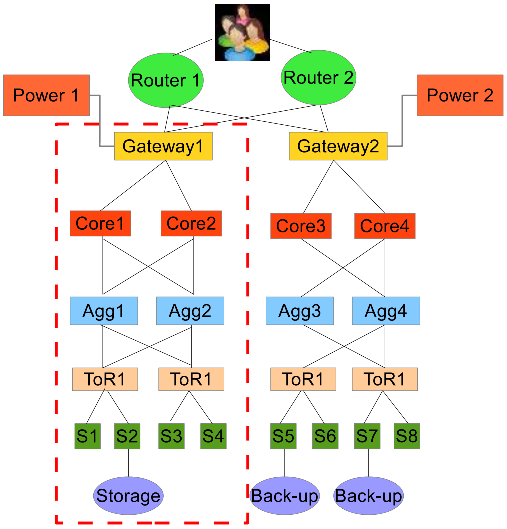

Figure 3.2: Full Dependency Graph of C_1

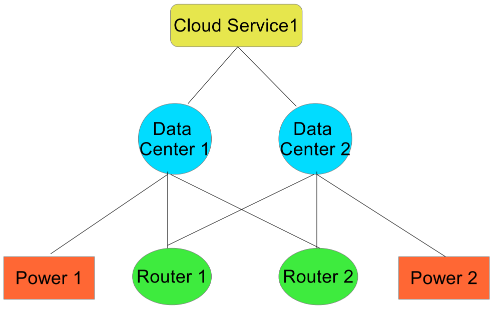

Figure 3.3: Abstracted Dependency Graph, suitable for SMPC

On the positive side, the topology-path form enables us to avoid using conditional statements in our SecreC code – something we must do to avoid leaking private information.

In order to capture the logical relationships among components of a cloud-service provider, we extend this representation to what we call the *topology-path form with types*. The SSU builds a “disjunction of conjunctions of disjunctions” data structure by assigning different “types” to the topology paths. Failure of the top event in the fault tree is the OR of a set of “type failures”; if any “type” that is an input to this OR fails, then the top event fails. Each “type failure” is the AND of failures of individual topology paths in the type; the “type failure” occurs only if all of the topology paths in that type fail. Failure of a topology path is the OR of failures of individual nodes on the path.

The SSU assigns a “type ID” to each topology path; the type ID is a function of the component IDs in

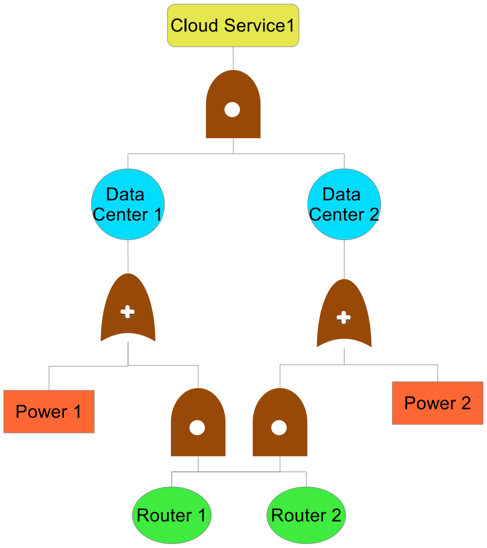

Figure 3.4: Fault Tree Based on Dependency Graph in Figure 3.3

the nodes on the path. Type IDs and the mapping from sets of component IDs to type IDs can be agreed upon by all of the relevant cloud-service providers and stored in a table before the P-SRA execution starts; so the SSU simply needs to look up type IDs during the protocol execution. To construct the fault tree from the topology-path form with types, the SSU traverses each path and constructs an OR gate for each path, the inputs to which are the nodes on the path. It then constructs an AND gate for each type of path, the inputs to which are the outputs of the OR gates of the paths in the type. Finally, the SSU constructs an OR gate whose inputs are the outputs of all the AND gates in the previous step.

For example, starting with the fault tree of Figure 3.4, the SSU can classify the topology paths of Figure 3.5 into two types. Type 1 includes the two topology paths ($Cloud\ Service_1$, DC_1, $Power_1$) and ($Cloud\ Service_1$, DC_2, $Power_2$). Type 2 includes the other four topology paths. It can be verified that the minimal FSes of the fault tree generated by the topology path form with types are the same as the minimal FSes of the fault tree in Figure 3.4; we refer to [ZWX$^+$] for more formal statements of the necessary details of fault-tree analysis.

Generate input for the SMPC: After constructing the topology paths with types, the SSU "pads" the paths so that they all have the same length L, where L is an agreed-upon global parameter distributed by the P-SRA host. Padding is accomplished by adding the required number of "dummy" nodes in which the component ID is 0. (Here, "0" is any fixed value that is *not* a valid, real component ID.) Similarly, the SSU adds a random number of "0 paths," which are topology paths with types in which all of the nodes have

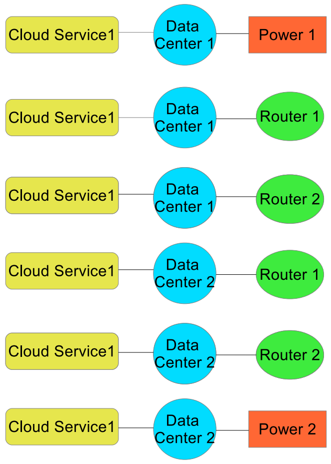

Figure 3.5: Topology-path Form of Dependency Graph in Figure 3.3.

```
Algorithm 1: Common-Dependency Finder
  Input: Fault tree $T_i$, $i = 1$ $to$ $N$, where N is the number of participating cloud-service providers
  Output: Common Dependency
foreach $T_I$ $and$ $T_J, I \neq J$ do
    private mask.clear();
    foreach $node_i \in T_I$ $and$ $node_j \in T_J$ do
        $private\ mask[i][j] = (node_i.ID == node_j.ID)$;
    private CommonDep.clear();
    foreach $node_i \in T_i$ $and$ $node_j \in T_j$ do
        $private\ CommonDep[i] = mask[i][j] \times node_j.ID + CommonDep[i]$;
    private CommonDependent.append(CommonDep);
return private CommonDependent;
```

component ID 0. The types of these 0 paths can be assigned randomly, because they do not affect the result – the 0 paths never fail. The purpose of this padding step is to prevent leakage of structural information about the cloud-service providers' architectures, including the number of topology paths or the size of each path. Finally, the SSU splits the padded paths into secret shares that are input to the SMPC protocol.

Identify common dependencies: A component is in the *common dependency* of cloud-service provider S_i if it is in the fault tree of S_i *and* in the fault tree of at least one other cloud-service provider S_j, $j \neq i$. Conceptually, the common dependency is very easy to compute by doing multiple (privacy-preserving) set intersections, followed by one (privacy-preserving) union. However, we need to do this computation without conditional statements; see Algorithm 1 for a method of doing so.

Calculate failure sets: Finally, the SMPC protocol integrates the fault trees of all participating cloud-service providers into a unified, global fault tree and performs fault-tree analysis. It can execute either algo-

Algorithm 2: Minimal-FS algorithm

```
Input: Global Fault tree $T$
Output: MinimalFS
foreach $private\ path_i \in T$ do
    foreach $private\ node_j \in private\ path_i$ do
        $private\ path_i.FS.apped(node_j)$;
        /* each path corresponds to an OR gate with input as the nodes along the path */
foreach $AndGate_i \in T$ do
    $AndGate_i.FS.clear()$;
    foreach $path_j \in AndGate_i$ do
        $AndGate_i.FS \leftarrow AndGate_i.FS \times path_j.FS$;
        /* process the AndGate for each type of topology paths                     */
        /* FS of $AndGate_i$ is the Cartesian Product of $AndGate_i.FS$ and $path_j.FS$.       */
private minimalFS.clear();
foreach $AndGate_i \in T$ do
    $minimalFS.append(AndGate_i.FS)$;
    /* process the OR gate connecting to the And Gates                             */
/* reduce redundant items in $minimumFS$ and assign the result to $minimalFS$, and then
   simplify minimalFS.                                                             */
$minimalFS \leftarrow reduce\_redundancy(minimalFS)$;
$minimalFS \leftarrow simplify(minimalFS)$;
return minimalFS;
```

rithm 2, which computes minimal FSes, or algorithm 3, a heuristic "failure-sampling" algorithm that is faster than algorithm 2 and computes FSes but does not guarantee that the FSes returned are minimal.

Algorithm 2 works as follows. Let T denote the unified, global fault tree; because we represent fault trees as padded, topology paths with types, T is simply the union of the fault trees of the individual cloud-service providers. The algorithm traverses T, producing FSes for each of the visited events. Basic events generate FSes containing only themselves, while non-basic events produce FSes based on the FSes of their child events and their gate types. For an OR gate, any FS of one of the input nodes is an FS of the OR. For an AND gate, we first take the cartesian product of the sets of FSes of the input nodes and then combine each element of the cartesian product into a single FS by taking a union. The last step of algorithm 2 reduces the top event's FSes to minimal FSes.

Algorithm 3 works as follows. For each sampling round, the algorithm randomly assigns 1 or 0 to the basic events (leaves) of the fault tree T, where 1 represents failure and 0 represents non-failure. Starting from such an assignment, the algorithm can assign 1s and 0s to all non-basic events in T, using the logic gates. At the end of each sampling round, the algorithm checks whether the top event fails. If the top event fails, then the failure nodes in this sampling round are an FS. The algorithm runs for a large number of sampling rounds to find FSes. In [ZWX$^+$], it is proven that most of the critical FSes can be found in this fashion but that the

Algorithm 3: Failure-Sampling Algorithm

```
Input: Global Fault tree $T$ and the number of samples $N$
Output: FSes
private FSes.clear();
for $i \leftarrow 1$ to $N$ do
    foreach private $path_j \in T$ do
        private tmp = 0;
        foreach private $node_s \in path_j$ do
            foreach private $node_k \in T$ do
                private random = 0 or 1 based on randomly flipping a fair coin;
                $tmp += random \times (node_s.ID == node_k.ID)$;
        /* calculate whether $path_j$ fails */
        $path_j.failure = (tmp > 0)$;
    foreach $AndGate_i \in T$ do
        $AndGate_i.failure = true$;
        foreach $path_j \in AndGate_i$ do
            $AndGate_i.failure = AndGate_i.failure$ && $path_j.failure$;
    private serviceFailure = false;
    foreach $AndGate_i \in T$ do
        $serviceFailure = AndGate_i.failure \parallel serviceFailure$;
    open(serviceFailure);
    if $serviceFailure$ then
        FS.clear();
        foreach $path_i \in T$ do
            FS.append($path_i.failure$);
        FSes.append(FS);
return FS;
```

FSes are not necessarily minimal.

LEU Protocol:

The LEU in the P-SRA client of cloud-service provider *S* performs fault-tree analysis on *S*'s macro-components. The LEU can use algorithm 2 or algorithm 3. Note that these computations are done locally and do not involve SMPC; so, large macro-components are not necessarily bottlenecks in P-SRA computation. It is very advantageous when a cloud-service provider can partition its infrastructure in a way that produces a modest number of large macro-components, each one of which is a “virtual node” in the SMPC.

3.4.5 Privacy-preserving Output Delivery

Recall that P-SRA performs an SMPC on dependency information that is potentially shared by multiple cloud-service providers and performs local computation on dependency information that is definitely relevant to only one provider. The intermediate results include common dependency and minimal FSes (or FSes if algorithm 3 was used). We now turn our attention to the outputs that P-SRA delivers to cloud-service providers and to cloud-service users. P-SRA gives cloud services the flexibility to choose exactly what should be output. However, we argue that the outputs should not compromise the privacy of cloud-service providers and must be illustrative of correlated-failure risk and reliability. We propose some specific outputs that satisfying these two requirements.

Output for Cloud-Service Providers

Common dependency: The common dependency set, as defined in Subsection 3.4.4, includes components shared by more than one cloud-service provider. It is useful for cloud-service providers, in that it can make them aware of unexpected correlation with other providers. They can then deploy independent components as backups to mitigate the impact of the common dependency or switch to independent components to improve the reliability of their service and decrease the correlation with other cloud-service providers.

Partial failure sets: If F is a (minimal) FS for cloud-service provider S, then the corresponding partial (minimal) FS is simply all of the components in F that are used by S. Such a partial FS gives S information about components whose failure may lead to an outage because equipment that is controlled by some other service provider fails. If S can build enough redundancy into its internal infrastructure to avoid failure of all of the components in this partial FS, then it will not suffer an outage because of F, regardless of what happens outside.

Sometimes the number of FSes is huge. If this is the case, we need to rank the FSes first and only output the partial failure sets of the top-ranked FSes. Ranking of comprehensive failure sets can be either probability-based or size-based [ZWX$^+$].

Output for Cloud-Service Users

Common-dependency ratio: Cloud-service users can obtain a *common-dependency ratio* for each cloud-service provider. We define the common-dependency ratio of cloud-service provider S as the fraction of components in S that are shared with at least one other cloud-service provider. Intuitively, the larger the common-dependency ratio, the higher the risk of correlated failure. In the extreme case, if a cloud service is deployed entirely on an external cloud infrastructure (as is the case with some Software-as-a-Service providers), then its common-dependency ratio is 1. If a cloud-service provider shares no components with other providers, then its common-dependency ratio is 0. Cloud-service users can evaluate risk and choose cloud providers in part based on this ratio. This common-dependency ratio does not reveal any information about internal architecture of the cloud providers.

Overall failure probabilities of cloud services: Cloud-service users can compare these failure probabilities with the reliability measures promised by the providers in their service-level agreements and evaluate whether they are subject the risk of unexpected, correlated failure. Failure probabilities, like common-dependency ratios, do not reveal the architectures of the service providers.

Top-ranked failure sets: Recall from Subsection 3.2.1 that, in its SMPC, P-SRA computes the *secret shares* of the (minimal) FSes of the cloud-service providers. As we have seen, an SMPC program can compute from those shares the partial (minimal) FSes that are delivered to the providers. However, an alternative SMPC program could use those shares to rank the (minimal) FSes based on failure probability or size. Then a small set of *top-ranked* (minimal) FSes can be delivered to cloud-service users. Just a few top-ranked sets can give users useful information about how to avoid correlated failures; they reveal some information about the cloud-service architectures, but this may be tolerable in some markets.

3.5 Implementation

3.5.1 P-SRA Prototype

The Sharemind SecreC platform includes a set of **miners** to execute the SMPC protocols and a controller to coordinate the miners. The SMPC protocols run by the miners are coded in **SecreC**, a C-like programming language for SMPC programs. Variables in SecreC may be declared as **public** or **private**. The language supports basic arithmetic as well as some matrix and vector operations. SecreC uses a client/server model, with multiple clients providing (secret-shared) input to the miners, which execute the SMPC protocol.

Our implementation of P-SRA is illustrated in Figure 4.8. The miners are installed in the SMPC module of the P-SRA host. The P-SRA clients and P-SRA host upload their SecreC scripts to the miners. The SecreC scripts are executed by the P-SRA clients remotely through the C++ interface of the controller or by the P-SRA host locally. The P-SRA clients execute the SecreC scripts to split their inputs into secret shares and to read and write shares of inputs or intermediate results from the miners' secure databases. The P-SRA host executes the SecreC scripts to perform the SMPC protocol that identifies common dependencies and performs fault-tree analysis. SecreC uses SSL for secure communication between miners and clients.

From Figure 4.8, it is not immediately obvious what one gains from using the Sharemind platform and SMPC instead of a trusted-party SRA as in [ZWX$^+$]: All of the miners, *i.e.*, the nodes that execute the SMPC protocol, run inside the P-SRA host; if they share information, then together they constitute a trusted party.

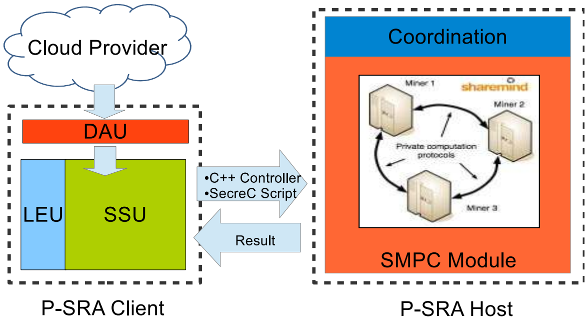

Figure 3.6: Implementation in Sharemind SecreC

However, this system configuration is merely the default of the currently available Sharemind "demo," and we have used it only in order to be able to build this proof-of-concept prototype as quickly as possible. In a real, deployed P-SRA (or any real SMPC-based application coded in SecreC), the miners would run on separate, independently administered machines and communicate over a network; no substantive changes to the SecreC compiler are needed to create executables that run on separate networked nodes, and we expect future Sharemind releases to create them. Thus, moving to P-SRA from the SRA of Zhai *et al.* [ZWX$^+$], in which one trusted auditor handles all of the sensitive information supplied by the cloud-service providers, is tantamount to "distributing trust" over a number of independently administered auditors no one of which is trusted with any sensitive information, in the sense that each receives only a secret share of every input; if the independent owners of the networked nodes that run the auditors do not collude, then the clients' inputs will remain private. This SMPC architecture, in which clients (or "input providers"), rather than executing an SMPC protocol themselves, instead send their input shares to independently administered computational agents that then execute the SMPC protocol, is known as *secure outsourcing* in the SMPC literature; see, *e.g.*, Gupta *et al.* [GSP$^+$12] for more information about secure outsourcing's history, its practical advantages, and its use in a routing application.

The SecreC compiler relieves programmers of the need to code standard cryptographic functionality. In particular, it generates secret-sharing code automatically. Currently, it uses additive secret sharing and thus guarantees privacy only against honest-but-curious adversaries. We expect future releases to incorporate more elaborate secret-sharing techniques and hence to protect input providers against stronger classes of adversaries.

The DAU and LEU in the P-SRA client are written in Python. The DAU uses the SNMPv2 library support from NetSNMP to collect network dependencies; it uses lshw, a lightweight tool that extracts detailed hardware configuration from the local machines, to collect hardware dependencies; it uses ps and gprof to collect software dependencies. The LEU uses the Network-X library [net] to process the dependency-graph data structures.

3.5.2 Case Study

This section outlines a case study to illustrate the prototype's operation. Let CS_1 denote a cloud service provided by cloud provider C_1. To improve the reliability of CS_1, C_1 decides to use providers C_2 and C_3 for redundant storage. Only C_1 serves users directly, while C_2 and C_3 provide lower-level services to C_1. This architecture is analogous to iCloud, Apple's storage service, which uses Amazon EC2 and Microsoft Azure for redundant backup storage.

Suppose Alice, a user of CS_1, wants to deploy a MapReduce function using CS_1. Alice deploys the MapReduce Master on a data center DC_1 of C_1, and C_1 uses a data center DC_2 of C_2 and a data center D_3 of C_3 as backup for the MapReduce Master. However, as in Figure 3.7, C_1, C_2, and C_3 depend on the same power station P_1. Alice and all three cloud providers are unaware of this situation. Therefore, they may overestimate the reliability of the MapReduce Master and underestimate the risk of correlated failure. If P_1 goes down, Alice's MapReduce may not work, because all the backup data centers may fail simultaneously.

The P-SRA system can help to identify P_1 as the common dependency in the cloud structure supporting CS_1 and provide multiple measures of reliability and correlated failure risk (the failure probability for Alice and partial FSes for C_1), without revealing significant private information about C_1, C_2, and C_3. Alice need not learn private topological information about the three cloud providers (or even learn of the existence of C_2 and C_3) but can accurately assess the failure risk via the P-SRA system. Meanwhile, C_1 can improve the reliability of CS_1 by connecting to alternative power stations or seeking redundancy from cloud providers other than C_2 and C_3, without learning private topological information about C_2 and C_3.

To further illustrate P-SRA, we display the details within a data center. There are a large number of components in data centers including servers, racks, switches, aggregate switches and routers. For simplicity, we generate the same topology for all the data centers and show only the components in DC_1 – see Figure 3.8. The MapReduce Master is installed on server 5 of DC_1. The DAUs of C_1, C_2, and C_3 collect the dependency information of each cloud provider. Then the SSUs abstract macro-components for each cloud provider using standard data-center abstraction. The SSUs pass the information within the data centers to the LEUs and establish connections with each other and the P-SRA host to execute the SMPC protocol. The LEUs perform fault-tree analysis on the dependency information within the data centers locally. The results of the

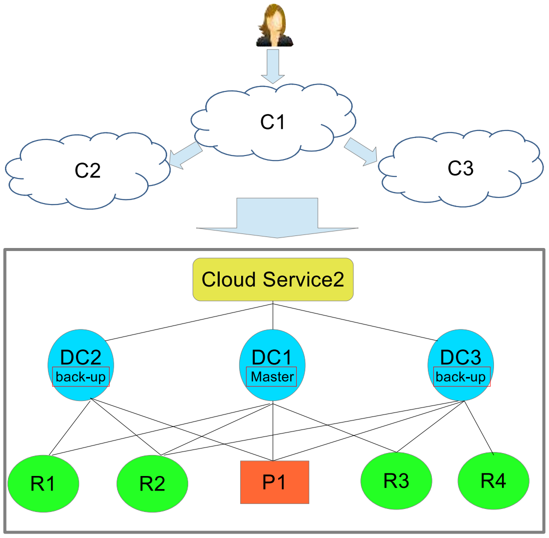

Figure 3.7: Multi-level Structure of Cloud Service

SMPC and the local computation are then combined as explained in Subsection 3.4.3.

The P-SRA system is practical in this case study. Even using a laptop with little computational power, equipped only with a 2.5GHz 2-core Intel i5 CPU and 2.00GB of memory, the running time used by the SSUs and P-SRA host to find the common dependency was approximately 20 seconds; the time to perform the fault-tree analysis was approximately 13 minutes using the minimal-FS algorithm and 55 seconds using the failure-sampling algorithm with 100 rounds. The running time for the LEUs deployed on servers equipped with two 2.8GHz 4-core Intel Xeon CPUs and 16GB of memory was less than 30 seconds for both the minimal-FS algorithm and the failure-sampling algorithm.

3.5.3 Large-Scale Simulation

This section evaluates the P-SRA prototype using larger-scale simulations. Our data set is synthesized based on the widely accepted three-stage fat-tree cloud model [Lei85] and scaled up to what we expect to find in real cloud structures. For the SMPC protocol run by the P-SRA host and the SSUs of P-SRA clients, we test the running time of the common-dependency-finder Algorithm 1, the minimal-FS Algorithm 2, and the failure-sampling Algorithm 3. Our output for both cloud-service providers and users can be computed efficiently from the common dependency and the (minimal) FSes.

We test the five cases summarized in Table 3.1. For simplicity, we generate only homogeneous cloud

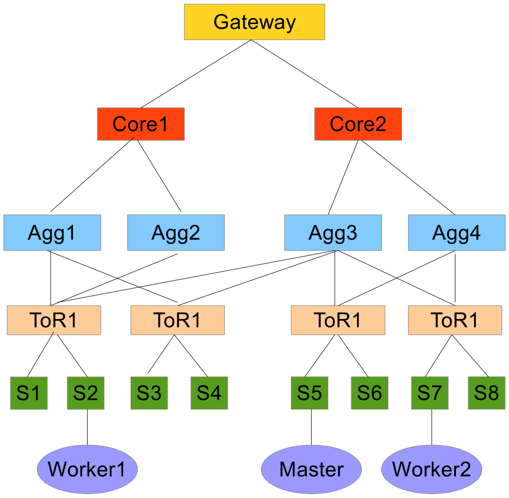

Figure 3.8: Components in Data Center DC_1: Core, Agg, and ToR represent core router, aggregation switch, and top-of-rack switch.

providers. In Table 3.1, the numbers of data centers, Internet routers, and power stations are numbers per cloud provider. The common-dependency ratio is as defined in Subsection 3.4.5. The padding ratio is the number of zeros with which the topology paths were padded divided by the total number of nodes on the topology paths after padding.

The five cases are intended to be illustrative of configurations broadly comparable to realistic multi-cloud services. To the best of our knowledge, it is uncommon for any cloud services to be deployed on more than three cloud providers or distributed over more than 10 data centers, because the total number of data centers worldwide is limited, and cloud-service management costs increase quickly as data centers are added. Amazon, one of the giant cloud providers, owns only 15 data centers globally [Amaa]; Microsoft Azure has fewer than 10 data centers [Azu].

Measured P-SRA computation performance is summarized in Figure 3.9. The P-SRA host and SSUs of the P-SRA clients were run on laptops with 2.5GHz 2-core Intel i5 CPU and 2.00GB of memory. We used these machines because the SecreC platform supported only Microsoft Windows when we started this work. We expect that performance would improve using higher-powered machines.

The common-dependency finder exhibits reasonable efficiency in all five cases, the runtimes of which are all less than 3 minutes. The minimal-FS algorithm yields exact minimal FSes (but takes exponential time in the worst case, because the problem in NP-hard), while the failure-sampling algorithm produces FSes approximating the minimal FSes and runs in polynomial time. In Cases 4 and 5, the minimal-FS

	Case 1	Case 2	Case 3	Case 4	Case 5
# of cloud providers	2	2	3	3	2
# of data center	1	3	8	10	3
# of internet router	3	5	10	15	5
# of power stations	1	2	3	5	2
ratio of common dep.	0.8	0.2	0.2	0.2	0.2
ratio of padding	0.0	0.0	0.0	0.0	0.5

Table 3.1: Configuration of Test Data Sets

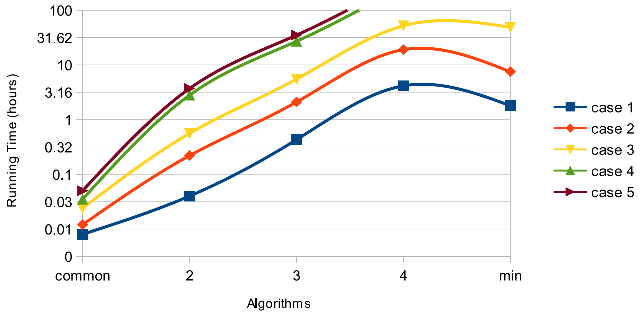

Figure 3.9: Performance of algorithms. On the X axis, “Common” represents the common-dependency finder, 2 through 4 represent the failure-sampling algorithm with sampling rounds at various powers of 10, and “Min” represents the minimal-FS algorithm.

algorithm was aborted before it finished, and thus no results are shown for them in Figure 3.9. The runtimes of other simulations of the minimal-FS algorithm and the failure-sampling algorithm range from 1 to 50 hours depending on the configuration. As the number of nodes increases, the efficiency of fault-tree analysis drops quickly. Case 5 shows that the cost of padding to conceal the statistical information of each topology path is high. Therefore, subgraph abstraction to reduce the size of the dependency graphs is important for the efficiency of fault-tree analysis in P-SRA.

For the LEUs in the P-SRA clients performing local computations, we also test the running times of both the minimal-FS algorithm and the failure-sampling algorithm. For the LEUs running on servers with two 2.8GHz 4-core Intel Xeon CPUs and 16GB of memory, the failure-sampling algorithm with 10^6 rounds on a data center with 13,824 servers and 3000 switches takes around 6 hours. For details, see Table 3.2. “FS round 10^n” denotes the running time (in minutes) of the failure-sampling algorithm running 10^n rounds’ “minimal FS” denotes the running time of the minimal-FS algorithm.

Table 3.2: Performance of the LEU of a P-SRA client

Configuration	Case 1	Case 2	Case 3	Case 4	Case 5
# of switch ports	4	8	16	24	48
# of core routers	4	16	64	144	576
# of agg switches	8	32	128	288	1152
# of ToR switches	8	32	128	288	1152
# of servers	16	128	1024	3456	13824
Total # of components	40	216	1360	4200	16752
Running time (minutes)					
FS round 10^3	< 0.7	< 0.7	< 0.7	< 0.7	< 0.7
FS round 10^4	0.7	0.7	1.7	2.3	6.9
FS round 10^5	0.8	0.9	5.3	28.1	6.9
FS round 10^6	1.7	4.5	65.0	243.5	462.9
FS round 10^7	28.3	56.6	512.1	NA	NA
Minimal FS	0.8	14.8	309.7	NA	NA

Chapter 4

Cloud User Infrastructure Attestation

4.1 Introduction

Customers running virtualized infrastructure in the cloud need attestation that they have received the resources they have paid for. For performance, reliability, and security reasons, a customer needs concrete assurance that his virtualized infrastructure, also called his "user infrastructure" (as opposed to "the cloud infrastructure," which is all the hardware and software owned by a cloud provider), satisfies his request. While cloud providers enter into service-level agreements (SLAs) with their customers, there is usually little attestation that can back up the SLA guarantees in practice. In particular, these agreements focus on "99.99...%" up time and other vague metrics. There is need for more concrete attestation to back up a provider's promises to customers.

While more assurance and attestation should attract more users, cloud providers may be unwilling to reveal their cloud infrastructure to the users or to active third parties[1]. Their competitive advantage may be diminished if others are able to learn about the settings and hardware that they use. Number of physical servers, types of servers, networking equipment used, networking bandwidths between specific nodes, etc., are all details about the cloud infrastructure that the cloud provider may want to keep secret.

Without a way to provide attestation to the users without revealing critical, proprietary information, providers may not be willing to adopt new attestation techniques. The users, however, are interested in making sure they get what they paid for. In particular, the users want attestation about their leased resources. Hence, our work aims to satisfy both users and cloud providers.

Previous work on this problem uses active, trusted third parties who act as intermediaries between the

[1] Here, an "active" third party is a trusted third party that must be online when the cloud provider and the cloud user execute the attestation protocols in order to help the cloud provider generate the proof or to help the cloud user verify it. By contrast, the trusted third party in our work can stay offline after certificating the infrastructure. The cloud provider and user can generate and verify the proof without the trusted third party's help.

cloud provider and the users. The provider is required to trust the active third party and to give it access to some information about the infrastructure. The users also trust the third party and believe that it will given them correct information about the provider (without actually giving the user any details about provider's infrastructure). The active third party often simply replies *yes* or *no* as to whether the user received satisfactory resources. There is, however, significant risk in disclosing infrastructure details to a third party, who presumably would know such details about other providers as well. Such information may be misused or even stolen by competitors.

Our solution, on the other hand, focuses on hardware security anchors, or roots of trust, installed in the individual servers of the cloud provider. We leverage well established TPM technology and also propose a novel component called a *Network TPM*. Both of these hardware components are used to collect information about the cloud infrastructure and attest to properties requested by users. In particular, our attestations refer not to the actual physical infrastructure but only to its properties. Thus, users never learn details of the infrastructure.

TPMs on the servers where the user's VMs run are used to attest to the properties of server infrastructure and to give users assurance about the properties of the servers. Our new proposed Network TPMs installed in these servers are used to attest to the properties of the network infrastructure and to give users assurance about the properties of the interconnection of their VMs. The TPMs and Network TPMs have to be trusted for correct operation; however, they never release any information to an outside third party. Meanwhile, our attestation protocol uses digitally signed data and verifiable computation mechanisms [PHGR13, TRMP12, SMBW12, SVP+12, CMT12] to ensure that the attestation is correct and trustworthy. Because the keys used by the hardware could be used to identify specific nodes in the infrastructure, and even to let outsiders enumerate all the physical components, Direct Anonymous Attestation (DAA) [BCC04] and Property-based Attestation (PBA) [SS04] are used together with the zero-knowledge property of certain verifiable computation mechanisms to prevent cloud users from tying the measurement to a specific physical node or link, even while receiving meaningful attestation.

4.2 Related Work

Trusted Computing: The Trusted Computing Group (TCG) [TCG] proposed a set of hardware and software technologies to enable the construction of trusted platforms. In particular, the TCG proposed a standard for the design of the trusted platform module (TPM) chips that are now bundled with commodity hardware. The TPM provides tamper-resistant cryptographic identities and functionalities, as well as random-number generators to support cryptographic operations. Leveraging the properties of TPM, trusted platforms [GPC$^+$03,PSvD06,SJV$^+$05] enable remote attestation, in which remote users can verify that the servers are secure and trustworthy. At boot time, the TPM on each server computes a measurement list (ML) that is stored in a set of tamper-resistant registers in the TPM. The remote party challenges the server with a nonce and asks the local TPM to authenticate the server using the ML, the nonce, and the private endorsement key of the TPM.

To address concerns that trusted platforms and remote attestation may leak private information about the attested machines, Direct Anonymous Attestation (DAA) [BCC04] and Property-based Attestation (PBA) [SS04] were proposed. DAA achieves anonymity of the attested machines using a type of group signature in which the signer is anonymous; that is, the verifier can determine whether the signer is a member of the group but cannot determine the signer's identity. PBA enables the verifier to check that certain properties of the configurations of the servers are met without releasing which configurations the servers are in.

Leveraging the trusted-computing works, several trusted cloud-computing platforms have been proposed (see [SGR09] for a survey) to protect the confidentiality and integrity of cloud users' data. Trusted cloud-computing platform (TCCP) [SGR09] leveraged a trusted third party to maintain a list of trusted nodes in the cloud providers and authenticate the nodes whenever a cloud user wants to launch or migrate a virtual machine. Excalibur [SRGS12] proposed a policy-sealed data abstraction, in which only those nodes whose configurations match the policy could unseal the data.

Our work leverages the TPM chip but also proposes a novel component called a "network TPM." Moreover, rather than focus on the hosts, as most TCG work does, our approach focuses both on the hosts (servers) and the network that connects them.

Accountability of Cloud Computing: Accountability in cloud computing has been explored; for example, Haeberlen [Hae10] proposed an accountable-cloud design in which each action is undeniably linked to the node that performs it, the system maintains a secure record of past actions that can be audited for signs of faults, and the audit results can be verified by an independent third party. It uses tamper-evident logs, virtualization-based replay, and trusted timestamping to detect faults in cloud providers. Meanwhile, Haeberlen *et al.* [HARD10] proposed an accountable virtual machine (AVM) that records non-repudiable information that allows auditors to check whether the software behaves as intended. Other works, such as [KJM$^+$11], outline the key issues and challenges in accountability of cloud-computing systems and proposed a framework to achieve accountability and trust via technical and policy-based approaches.

Verifiable Computation: Attestation that remote services are running correctly is related to the idea of verifiable computation. There has been a great deal of work on protocols for verifiable, outsourced computation, in which a user submits inputs to a server that is supposed to compute a predefined function; after receiving the result, the user should be able to verify that the function was computed correctly. Efficient mechanisms have long been known for verifiable, outsourced computation of constrained classes of functions [MWR99, GM01, DG05]. General purpose solutions were proposed [GMR89, Kil92, GKR08, Gro10, GGP10, CKV10] but often relied on complex cryptographic protocols such as Fully Homomorphic Encryption [Gen09] and Probabilistic Checkable Proofs [AS98]. Recent works [TRMP12, SMBW12, SVP$^+$12, CMT12] considerably improved the efficiency of the general-purpose solutions, but the verification protocols were not quite practical. Most recently, Parno *et al.* [PHGR13] proposed a system for efficiently verifying general computations that produces a small (288-byte) proof for each computation performed and that anyone with a public verification key can check.

Rather than focus on verifying specific computation, this work focuses on providing attestation to the user that he or she received the requested resources, and then any computation can be performed on these.

4.3 Cloud User Infrastructure

We focus on attestation of the cloud user infrastructure, which we define in this section. A cloud user infrastructure is essentially an instance of a user's requested infrastructure (VMs and their interconnection) on the cloud provider's infrastructure. The term "cloud user infrastructure" also captures the user's requirements with respect to the properties that the hardware on which the VMs run should have and how the physical servers hosting the VMs should be connected, *e.g.*, with redundant links for reliability. We describe the concepts of user infrastructure, cloud infrastructure, and cloud user infrastructure below.

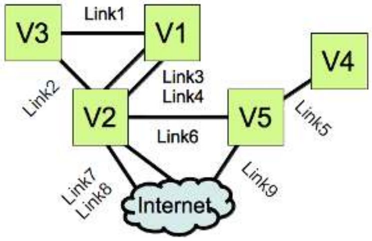

Figure 4.1: User infrastructure example with multiple VMs and links among them

User Infrastructure: The basic request that a user can make when leasing cloud resources is to request a set of VMs, V_x, with certain properties, $ps_V = Prop(V_x)$. These properties may include amount of memory, processor speed and number of virtual CPUs, and disk storage. Such requests, however, do not say anything about how the VMs are connected. The idea of *user infrastructure* thus also includes the interconnection of the VMs with each other and with the external internet, which we denote $ps_{Link} = Prop(Link_x)$. These virtual properties of the VMs and their interconnection comprise the basic user infrastructure that one could request; an example is shown in Figure 4.1. [2]

The users may also have requirements for the physical properties on which the VMs and their network are realized. For example, certain VMs will need to be connected with redundant links, and some may have to be placed on separate servers or on servers with special properties, *e.g.*, “equipped with a TPM chip.”

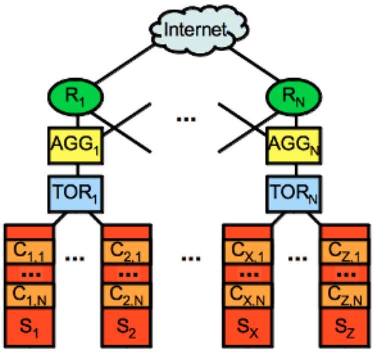

Figure 4.2: Cloud infrastructure example.

Cloud Infrastructure: The user infrastructure will eventually be mapped into and instantiated on a cloud infrastructure; an example is shown in Figure 4.2. Each cloud provider has an infrastructure that consists of physical compute and networking elements. Typically, there are compute nodes (C), server racks (S) each of which houses multiple compute nodes, and top-of-the-rack (TOR) switches that connect all the compute nodes in a server rack and provide connection to the aggregation switches (AGG). Aggregation switches are connected to routers (R), which eventually provide connectivity to the outside world. This model is based on the popular fat-tree networking model, originally applied to supercomputing and now common in data centers [Lei85].

Each of the physical components has certain properties, or “attributes,” $attr = Attr(C, S, \ldots)$. The server properties include memory, processor speed, and disk and may also include special properties such as “equipped with a TPM chip.” The networking components have bandwidth properties and may also have redundant links

[2] Note that there are even more complex ways of defining user infrastructure. For example, in a “cloud resident datacenter” [KDSR11], a customer can design and manage the servers, network, storage, and middleboxes, as in a private data center. This may be too complex for average users, however; our user infrastructure definition occupies a middle ground between simply requesting VMs and a full-blow virtual datacenter specification.

for reliability.

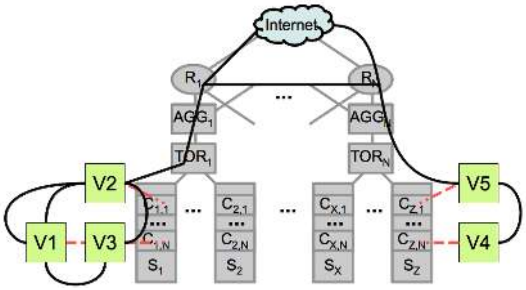

Figure 4.3: Cloud user infrastructure example, showing mapping of the user infrastructure, Figure 4.1, onto the cloud infrastructure, Figure 4.2.

Cloud User Infrastructure: Given the user infrastructure request, the cloud provider allocates to the user infrastructure resources on its physical infrastructure; this creates the “cloud user infrastructure,” an example of which is shown in Figure 4.3. The VMs are mapped to run on specific servers. Virtual links may be mapped to physical links between servers or may remain purely virtual if two VMs that share a link happen to be mapped to the same server. After the instantiation of the user infrastructure on the cloud infrastructure is done, the cloud user infrastructure has been created, and the user seeks attestation that the properties he requested for his VMs, ps_V, and interconnections, ps_{Link}, are collectively satisfied by $attr$, the attributes of the underlying servers and network.

4.4 Cloud User Infrastructure Attestation

In this section, we describe the procedure and algorithms for cloud user infrastructure attestation. First, the cloud service user receives the list of servers in the cloud user infrastructure from the cloud service provider, together with the provider’s attestation that the trusted components exist on the servers. Second, the cloud users leverage the trusted components to verify the architecture of each server. Third, the cloud users leverage the trusted components to measure the topology information of the user infrastructure. Fourth, the cloud users verify whether the connectivity and topology configurations provide the correct properties. Note that the cloud provider attests to the desired properties, not to the specific configurations of its hardware systems; thus, it does not reveal sensitive infrastructure trade secrets to the customers or potential competitors.

4.4.1 Threat Model

We target scenarios in which the cloud users do not trust the cloud provider to fulfill service-level agreements. The cloud provider may maliciously or accidentally fail to implement the cloud user infrastructure that cloud users pay for. The cloud provider has privileged control over the hypervisor, operating systems, and virtual machines. It can also migrate the users' VMs from one server to another. However, we assume that, as long as the cloud provider implements the required cloud user infrastructure, the properties of the infrastructure are enforced, and no third party can hack the security mechanisms to compromise the cloud users' data. Vulnerabilities of the security mechanisms exploited by the cloud providers are outside of the scope of this work.

We assume that there may be hardware components installed on the servers of the cloud provider that are trusted by both the cloud users and the cloud provider. These trusted components are accurate and tamper-resistant as long as they reside on the servers. Existing trusted components include Trusted Platform Modules (TPMs) and the associated Trusted Software Stack (TSS). We also advocate a new trusted component that can be used to measure the network topology; it is tamper-resistant and trusted by both the cloud provider and the users. The hardware components are associated with private signing keys and public verification keys available from the manufacturer; the verification keys can be used to confirm the accuracy of the information sent from these components.

4.4.2 Attestation of Server Architecture

Cloud users can leverage the trusted hardware components to verify the server architecture. The TPM can be used to verify the system architecture. The TPM on each server computes a measurement list, ML, of the configuration of the server; it includes, for example, the BIOS, the bootloader, and the software implementing the platform. The ML is stored securely in the TPM, which is tamper-resistant. A remote cloud user can challenge the server by sending a nonce n_c and the desired properties ps_V to the platform running on the server. The local TPM on the platform checks whether the ML satisfies ps_V, creates a message including the results of these checks and n_c, and signs the message with the her private key. The platform then sends the signed message back to the challenger, who can verify the signature of the message using the public key of the TPM and check whether ps_V is satisfied. Direct Anonymous Attestation (DAA) [BCC04] and Property-based Attestation (PBA) [SS04] can be used so that cloud users are not able to tie the measurement to a specific physical node based on the key.

4.4.3 Attestation of Topology Infrastructure

We focus on topology infrastructure attestation, which has not been fully explored before. The following questions are crucial. 1) How can one obtain tamper-resistant measures of topology information? 2) How can one measure and consolidate the topology information? 3) How can one verify the properties of the topology infrastructure based on the topology information?

We provide a framework to answer these three questions as follows. First, we propose a design of a smart and secure component, which we call a Network TPM (NTPM), to collect topology information. The NTPM can be installed on each server as a network card to collect topology information in a distributed manner. Each NTPM is equipped with some hardware-based tamper-resistant cryptographic functionalities and can sign the topology measurements to guarantee integrity. There is a hardware-based secure channel between the CPU and the NTPM to consolidate the topology information.

Second, we propose a set of protocols that enable NTPMs to collect and analyze the topology information. We describe two kinds of networks in the topology infrastructure, named virtual network and physical network, based on the vitualized or physical nature of the nodes and links in the networks. We propose a Physical Topology Discovery Protocol (PTDP), which enables the NTPMs to communicate with each other to measure the physical topology, and describe how to use well known protocols to collect virtual network information. We also propose a Topology Attribution Protocol (TAP) to consolidate the virtual network and physical network and obtain the topology infrastructure.

After obtaining the topology infrastructure, we leverage the delegation model to do the attestation. There exists a trusted third party that specifies an algorithm $A(CUTI, p)$ that decides whether the cloud user infrastructure $CUTI$ provides p, a set of properties. The trusted third party stays offline after certifying the algorithm $A(.,.)$. Thus, it is not an "active, trusted third party" of the type used in previous work in this area. Finally, we propose a property-based attestation protocol, in which the NTPMs, the cloud user, and the untrusted platform controlled by the cloud provider cooperate to generate a proof that the cloud user can verify. The protocol protects the private information of the cloud provider, i.e., the cloud user learns nothing about the private infrastructure of the cloud provider except whether it satisfies a certain set of properties.

Design of the Network TPM

The design of the proposed Network TPM (NTPM) is shown in Figure 4.4. The NTPM consists of a network module (NM) and a crypto module (CM). The NM is responsible for collecting topology information by communicating with the network modules of the other NTPMs in the cloud user infrastructure. There are two units in the NM of the NTPM. The first is a communication unit, which executes the communication protocols with other NTPMs. The other is a topology-storage unit, which consists of a set of secure memory units (registers) storing the topology information collected by the communication unit in a tamper-resistant manner. The storage is read-only to the untrusted platform; so the hypervisor cannot modify the measures of the topology information.

The CM provides a secure random-number generator, non-volatile tamper-resistant storage, cryptographic functions for encryption, decryption and digital signature, and a hash function. The CM can measure the trusted NTPM driver, check its integrity, and encrypt/decrypt and sign the measures collected by the communication unit of NM. The communication channel between the communication unit of NM and the CM is realized by a bus, which is tamper-resistant.

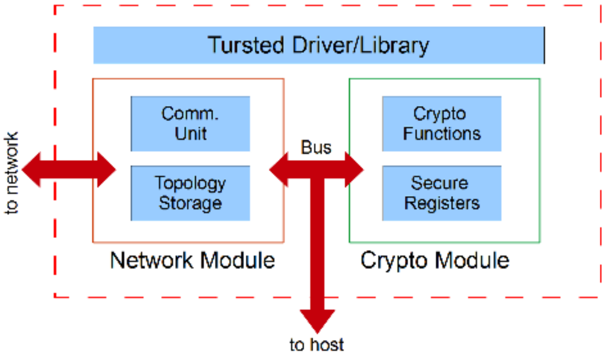

Figure 4.4: Network TPM Design

Measuring the Topology Infrastructure

Virtual Network v.s. Physical Network: The Cloud provider can virtualize the networks for the cloud users. Different VMs can be launched on the same server and connected with each other through a virtual network. Each VM has its own virtual network card and MAC address to connect with virtual routers. The links between the VMs can be virtual, too. The user who controls the VMs does not perceive a difference between the virtualized network and a physical network, because the network interfaces are the same in both cases. When calling the topology-discovery algorithms, such as the Link-Layer Discovery Protocol (LLDP), the cloud user can only measure the link-layer network topology after the virtualization process. The links discovered by LLDP can be either virtual or physical, depending on the implementation details of the cloud provider, which are often not revealed to the cloud users.

For instance, in Figure 4.5, user 1 controls VMs 1 through 4. The cloud provider initiates the VMs in two servers. VM1 and VM2 communicate with each other through a virtual network that is established on server 1. The links connecting VM1 and VM2 are virtual, which leverages the software/hardware of server 1. The links between the two servers are physical. If user 1 only needs to coordinate VM1 and VM2, she only needs the virtual network. However, if she needs to coordinate VM1 and VM3, she needs to go through the physical links as well as the virtual links.

In order to provide concrete and precise measures of the cloud user topology infrastructure, the NTPMs should be able to measure both virtual network topology and physical network topology. Additionally, the NTPMs should be able to evaluate the virtual and physical network topology and attribute the virtual network topology to the physical network topology so that the cloud users can better understand the performance and risks of their cloud services.

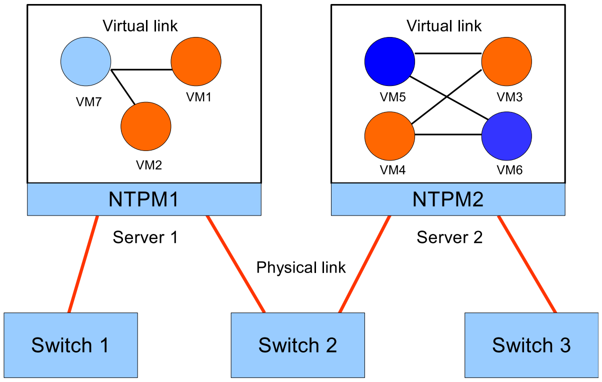

Figure 4.5: Virtual and Physical Networks

Overview of the Topology Measurement Protocol: When a cloud user starts to verify the properties of the topology infrastructure, she first measures the virtual network assigned by the cloud provider to determine the virtual links between the nodes in the link layer. The virtual network is the collection of all the data layer links and nodes that constitute of the cloud user topology infrastructure. The virtual network topology is then passed to the NTPMs, which measure the physical topology, isolate the virtual networks from the physical networks, and attribute the virtual network to the physical network. The virtual network, the physical network, and the attribution relationship from virtual to physical network are stored in the NTPMs and used as the inputs to the property-based attestation of topology infrastructure.

Measuring the Virtual Network: The cloud user can use known link-layer topology discovery protocols, such as LLDP, to measure the virtual topology and discover the links and nodes along the paths between the VMs she controls and from the VMs to the internet. She can also request the virtual topology information from a database, such as the management information database (MIB). The design of the NTPM is independent of the link-layer topology discovery protocols. After obtaining the virtual topology, the cloud user passes the virtual network information to the NTPMs on the servers running the VMs by passing either an adjacency matrix or the queries of the MIB. We assume that the link-layer topology discovery protocols are accurate and tamper-resistant, *i.e.*, that the cloud provider cannot tamper with them.

Measuring the Physical Network: Because the computational power of the hardware is limited, it is hard to implement most of the topology-discovery protocols directly in NTPMs; trusted modules often rely heavily on the computational power of software. A simple but effective protocol is demanded, and thus we proceed as follows. The NTPMs choose one NTPM as the *master NTPM*. The master NTPM communicates with other NTPMs involved in the cloud user infrastructure and collects the physical topology information. It is preferable that the master NTPM be the NTPM in one of the servers that runs one of the VMs of the cloud user, but it can also be selected by an optimization protocol according to the workload of each NTPM and the remaining attestation tasks. In our system, we randomly select a master NTPM from the NTPMs and leave more sophisticated methods as future work.

We design a top-down mechanism for the NTPMs to measure the physical topology. First, the master NTPM examines the virtual topology and verifies whether there are any nodes that use one of the NTPMs as network cards. If not, we call the virtual network a *pure virtual network*, *i.e.*, one in which all the nodes and links are virtual as there are not any nodes with physical network cards. In this case, the physical topology consists of the servers that run and support the virtual network. If the virtual network is not a pure virtual network, then the master NTPM identifies all the nodes that are physical and all the links that connect to the physical nodes in the virtual network. The master NTPM then executes a *Physical Topology Discovery Protocol* (PTDP) to measure the physical topology that connects the NTPMs involved in the cloud user infrastructure.

The design of the PTDP is as follows. First, the master NTPM establishes connections with all the other NTPMs in the virtual network and sends them an initial signal to start the topology discovery process. Then each NTPM sends the acknowledgement to the master NTPM and waits for the signals from other NTPMs. Then the master NTPM initiates a breadth-first traversal to reach all the NTPMs involved. At each round of the traversal, one NTPM is called the "current NTPM"; the first current NTPM is the master NTPM. There are three phases in each round. First is the probing phase, in which the current NTPM sends a probing packet to each of the involved NTPMs. Another NTPM sends an acknowledgement back to the current NTPM only if it is the neighbor of the current NTPM, which is determined by verifying whether the destination MAC address in the packet equals its own MAC address. In the probing phase, the NTPMs never forward the packets for other NTPMs; so, after the probing phase, the current NTPM knows its neighbors. Second is the reporting phase, in which the current NTPM sends the list of its neighbors to the master NTPM, and the master NTPM

Algorithm 4: Physical Topology Discovery Protocol

```
Input: Virtual topology of the user infrastructure $VTP = \{E_{VTP}, V_{VTP}\}$, available set of NTPMs $N$
Output: Physical topology of the user infrastructure $PTP$
Set of Involved NTPMs of the user infrastructure $SubNTPMs = V_{VTP} \cap N$;
master NTPM $m = generateMasterNTPM(SubNTPMs)$;
current NTPM $cur = m$;
visited NTPMs $Visited = \emptyset$;
Edges of PTP $E_{PTP} = \emptyset$;
while $Visited \neq SubNTPMs$ do
    $Edge_{cur} = \emptyset$;
    foreach $v \in SubNTPMs$ and $v \neq cur$ and $v \notin Visited$ do
        $cur.sendSignalTo(v, probing)$;
        if cur.receiveACK(v) then
            cur.neighbor.add(v);
            $Edge_{cur}.add(cur, v)$;
    if cur.sendSignalTo(m, reporting, list(neighbors)) then
        $E_{PTP} = E_{PTP} \cup Edge_{cur}$;
        $Visited.add(v)$;
    nextCur = cur.generateNextCur(list(neighbors));
    while not cur.sendSignalTo(m, selecting, nextCur do
        m.sendNaN(v, selecting);
        nextCur = cur.generateNextCur(list(neighbors));
    cur = nextCur;
return $PTP = \{E_{PTP}, SubNTPMs\}$;
```

marks the current NTPM as a visited node. In the third or "selecting" phase, the current NTPM randomly selects a neighbor as the current NTPM for the next round and reports it to the master NTPM. The master NTPM checks that the new current NTPM has not been visited already; if it has, then the master requests another selecting phase. In the reporting and selecting phases, all the NTPMs help to forward the packets to other NTPMs. After the selecting phase, another round starts, and things proceed in this fashion until all the NTPMs are visited (i.e., become "current" once).

The communication between the NTMPs is done through the PTDP Data Unit (PTDPDU) and proceeds as follows. There are three fields for each PTDPDU. First is the source address field, which records the MAC address of the source node, virtual or physical. Second is the destination field, which is the MAC address of the destination node. Third is the data type field, which is used to record the data type of the PTDPDU and the state of the breadth-first traversal. The data units are signed by the NTPMs on the nodes along the path to guarantee integrity.

Attribution to the Physical Network: After the physical network is obtained, the master NTPM compares the virtual network and the physical network to obtain the topology infrastructure of the cloud user. Basically, the attribution process finds the clusters of the virtual network that are supported by the physical

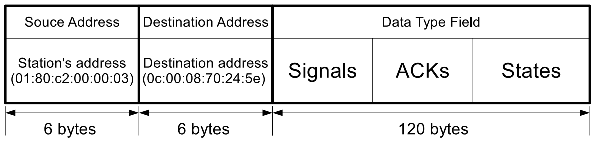

Figure 4.6: Physical Topology Discovery Protocol Data Unit

nodes and links. If the clusters of the virtual network connect to the other parts of the virtual network through only physical nodes and links, we attribute the virtual cluster to the physical cluster. Some properties, such as bandwidth, are more relevant to the virtual clusters, while other properties, such as reliability and failure probability, are more relevant to the physical clusters. Others are determined by both.

There are two different scenarios for topology attribution from virtual network to physical network. For a pure virtual network, the entire virtual network should be attributed to the physical server that supports it. The performance and the reliability of the virtual network are highly related to the performance and reliability of the underlying server. For a *hybrid virtual network*, the attribution should be determined by the connectivity and virtualization of the virtual network.

In order to describe the attribution process, we use a special link called the *attribution link*, *i.e.*, a link that connects a VM and a NTPM that resides on the server that supports the VM. The attribution link is not a real network link but rather a link that indicates the attribution relationship between the VMs and the server that supports the VMs. The attribution links can have properties, such as failure probabilities and can be treated as normal (virtual/physical) links when computing the properties of the topology infrastructure.

The attribution links describe the relationship between the virtual network and the physical network. The properties of the attribution link are determined by the virtualization and configuration of the cloud provider. We use the virtual network, the physical network, and the attribution links to describe the topology infrastructure of the cloud users and compute the properties of the topology infrastructure.

There are three kinds of properties of the attribution link: *dominant properties*, *supportive properties*, and *irrelevant properties*. Dominant properties determine the properties of the virtual networks, because they result directly from the properties of the underlying physical networks. For example, if a virtual network is established on a single server, then the reliability properties of the underlying server and the virtualization process determine the reliability properties of the virtual network. Supportive properties have to be combined with the properties of both the virtual networks and physical networks. For example, the security properties of the virtual networks involve the virtual machines, the virtualization process, and the physical servers. Irrelevant properties exclude the involvement of the attribution links. For example, the bandwidth of the virtual network does not depend on the attribution links, because the virtualization process is not the bottleneck of the transmission bandwidth.

Therefore, for different kinds of properties, we use different methods to assign properties to the attribution links and compute the properties of the topology infrastructure. For dominant properties, we replace the virtual networks with the attribution links, because the properties of the attribution links, combined with the properties of the physical networks, can determine the dominant properties of the whole virtual networks. We then use this new topology without virtual networks to compute the dominant properties of the topology infrastructure. For supportive properties, we do not replace the virtual networks with the attribution links but instead add the attribution links directly into the topology infrastructure that already contains the virtual and physical networks. When computing the properties of the topology infrastructure, we treat the properties of the attribution links in the same way that we treat the properties of the virtual and physical networks. For irrelevant properties, we also directly add the attribution links to the topology infrastructure, but we just ignore the properties of the attribution links when we compute the properties of the topology infrastructure.

The process of topology attribution is as follows. First the master NTMP compares the virtual networks and the physical networks to identify the common parts, i.e., the physical nodes in the virtual networks. Then the master NTMP identifies the virtual clusters that are isolated by the physical nodes in the virtual networks and adds an attribution link from each virtual link to the physical node that isolates it. Finally, the master NTMP classifies the rest of the connected VM clusters as different pure virtual networks, adds the physical nodes that support the pure virtual networks into the topology, and adds the attribution links between each VM and the server that supports it.

Topology Property Attestation

Overview: After obtaining the topology infrastructure, the NTPMs, cloud provider, and cloud user execute an attestation protocol so that the cloud provider can prove that the topology infrastructure provides the properties required by the cloud user. The cloud user and cloud provider first need to reach an agreement on which topology infrastructure provides what properties. A simple solution is to follow the trusted computing literature to create a data structure in which each topology infrastructure is associated with the properties that it provides and store the data structure in a trusted database. However, the topology infrastructure is far more complicated than the system architecture; so this solution requires a lot of storage resources and cannot be realized by hardware based NTPMs. We propose a more suitable solution in which an algorithm is specified to check whether a cloud user topology infrastructure provides a set of properties. For instance, the cloud user

Algorithm 5: Topology Attribution Protocol

```
Input: Virtual topology of the user infrastructure $VTP = \{E_{VTP}, V_{VTP}\}$, Physical topology of the user
        infrastructure $PTP = \{E_{PTP}.V_{PTP}\}$
Output: Topology of the user infrastructure with attribution links $ATP$
the set of physical nodes in virtual nodes $PV = empty()$ ;
$ATP.add(VTP \cup PTP)$ foreach node $e_{VTP} \in E_{VTP}$ do
    foreach node $e_{PTP} \in E_{PTP}$ do
        if $e_{VTP}.NTPM == e_{PTP}.NTPM$ then
            $PV.add(e_{VTP})$;
hybrid = empty();
foreach node $e \in PV$ do
    path.clear();
    foreach node $p \in$ e.neighbors() do
        path = deepFirstSearch(p, hybrid);
        if all nodes in path are virtual then
            foreach node $t \in path$ do
                ATP.attribute(t, e);
                hybrid.add(t);
foreach node $e \in E_{VTP}$ and !hybrid.contains(e) do
    clusters = seachClusters(e);
    foreach cluster $\in$ clusters do
        $e_{PTP} = seachPhysicalNode(cluster)$;
        $ATP.add(e_{PTP})$;
        $ATP.attribute(cluster, e_{PTP})$;
return $ATP$;
```

and cloud provider can specify an algorithm to determine whether there is a pure virtual network to verify that whether the cloud user bears the correlated failure risk.

As mentioned before, we leverage a delegation model in which an offline trusted third party helps to specify the algorithm that determines whether a topology infrastructure provides a certain set of properties. Then the trusted third party stays offline and does not involve itself in the attestation procedure. The trusted third party allows the cloud users to avoid engagement in complex technical details that requires deep technology knowledge. It could be eliminated if one wishes to use ring signatures such as in [CLMS08].

After reaching an agreement, the NTPMs can assist the cloud user to verify the properties. A simple solution is to let the NTPMs to run the protocols, sign the verification results, and send them to the cloud user. However, because the NTPMs' computing power is limited, this solution is not practical. In our system, the cloud provider affords the computation workloads for the NTPMs. The master NTPM signs and sends the topology infrastructure as the inputs to the cloud provider. The cloud provider computes the verification result and the proof and sends them to the cloud user. The following three requirements of the system need to be satisfied: 1) The input of the topology infrastructure is correct, and the cloud provider cannot modify it. 2) The computation of the verification result is correct in the sense that the cloud provider executes the correct protocols and cannot switch to other protocols or provide wrong computation results. 3) The cloud user should be able to verify the result without compromising the private cloud infrastructure information of the cloud provider.

The first requirement is satisfied because of the tamper-resistant property of the NTPMs. For the second and third requirements, we leverage verifiable computation [PHGR13,GGP10,TRMP12,SMBW12,SVP^{+}12]. In particular, we design a system in which the master NTPM executes a verifiable computation protocol with the cloud provider, and the cloud provider returns the verification results and a proof that the result was computed with the correct inputs (topology infrastructure) and the correct function (the verification protocol). The cloud user can verify the computation results and the proof on her own. The proof is zero-knowledge, in the sense that the cloud user can only verify whether the topology infrastructure provides the required properties, *i.e.*, it cannot infer the specific topology infrastructure implemented by the cloud provider.

Attestation Protocol: Suppose the cloud user topology infrastructure obtained is $CUTI$, and the set of properties required is p. There are three parties involved in the attestation protocol, i.e., the master NTPM, the cloud provider and the cloud user. The three parties try to verify that $A(CUTI, p) = 1$, where A is the algorithm certified by the trusted third party.

The master NTPM generates a verifiable computation (VC) key pair (EK_A, VK_A) by running a protocol

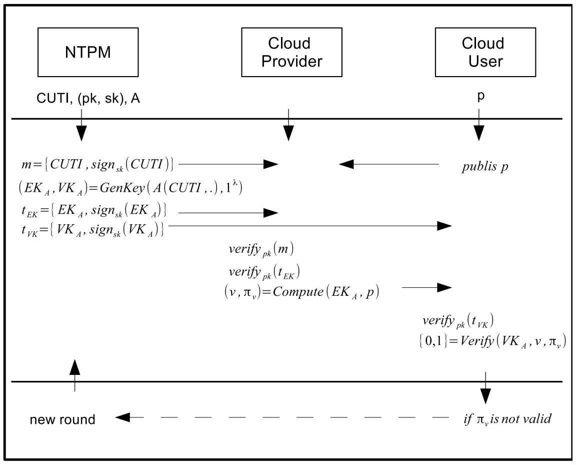
NTPM
Cloud Provider
Cloud User
CUTI, (pk, sk), A
p
$m=\{CUTI, sign_{sk}(CUTI)\}$
publis p
$(EK_A, VK_A)=GenKey(A(CUTI,.),1^{\lambda})$
$t_{EK}=\{EK_A, sign_{sk}(EK_A)\}$
$t_{VK}=\{VK_A, sign_{sk}(VK_A)\}$
$verify_{pk}(m)$
$verify_{pk}(t_{EK})$
$(v,\pi_v)=Compute(EK_A, p)$
$verify_{pk}(t_{VK})$
$\{0,1\}=Verify(VK_A, v, \pi_v)$
new round
if π_v is not valid

Figure 4.7: Property-based Attestation Protocol of Topology Infrastructure

$(EK_A, VK_A) \leftarrow GenKey(A, \lambda)$ which takes $A(.,.)$ and a security parameter λ as inputs. EK_A is an evaluation key that is sent to the cloud provider, and VK_A is a verification key that is sent to the cloud user. To guarantee the integrity of the key pair, the master NTPM signs the key pair with her private key, *sk* before sending the individual keys to the cloud provider and user. Suppose the signing protocol of the master NTPM is $sign_{sk}$, the cloud provider receives $\{EK_A,\ sign_{sk}(EK_A)\}$, and the cloud user receives $\{VK_A, sign_{sk}(VK_A)\}$. The cloud user and provider can use *pk*, the public key of the master NTPM, and the verification protocol of the master NTPM $verify_{pk}$ to verify the integrity of the VC key pair.

After obtaining the evaluation key of the VC key pair, the cloud provider executes a verifiable computation protocol to obtain the result of $v = A(CUTI, p)$ and the proof π_v that proves the correctness of v, i.e., $(v, \pi_v) \leftarrow Compute(EK_A, p)$. The $CUTI$ is sent from the master NTPM, and the set of properties that needs to be verified is sent from the cloud user. In order to preserve the privacy of the cloud provider, the cloud provider implements a randomization method such as in [PHGR13] to add randomness in the proof, so that the cloud user cannot learn $CUTI$ through the verification process.

Then, the result pair v, π_v is sent back to the cloud user. The cloud provider is accountable for any mistakes or manipulations that occur during the VC protocol. The cloud user can determine whether v is correct or not by running a VC-verification protocol $Verify(VK_A, v, \pi_v$,p); the protocol returns “pass” or “fail” to indicate whether or not the proof is valid, and it also assigns a numerical value to the variable v. We use a VC-verification protocol that has a strong correctness property: When the computation is correct, the proof is always valid. Therefore, if the verification fails, the cloud user knows that there were mistakes or manipulations on the part of the cloud provider; the user can reject the verification result and request another attestation process. If the verification passes and $v = 1$, then the cloud user knows that $CUTI$ supports p. If the verification passes but $v \neq 1$, then the cloud user knows that $CUTI$ does not support p.

4.5 Implementation

4.5.1 Prototype of Topology Infrastructure Attestation

We use ComplexNetworkSim [Com] and Pinocchio [PHGR13] to implement a prototype of our topology infrastructure attestation framework and evaluate the prototype by simulation. ComplexNetworkSim is a simulation tool written in Python and used to model and simulate network related problems. ComplexNetworkSim leverages the Python graph package NetworkX [net] to represent complex network information and implements SimPy [Sim], an object-oriented, process-based discrete-event simulation language for Python. ComplexNetworkSim can generate NetworkAgents, which can interact with their neighbors according to a network topology specified as a NetworkX graph object. The NetworkAgents can define different behaviors and protocols and run a clean but sophisticated discrete event simulation. ComplexNetworkSim enables us to implement our topology measurement and infrastructure attestation protocols without involving too many network communication details, thereby providing clean and meaningful simulation results. Pinocchio is a system that provides fast, zero-knowledge, verifiable computation; it was implemented by Microsoft Research.

Our implementation is illustrated as Figure 4.8. Each node in the cloud infrastructure is installed with a NetworkAgent, consisting of two components: NTPM and Network Trusted Software Stack (NTSS). The NTSS is the trusted software that interacts with NTPM to read and write the secure registers and communicate with the untrusted platform controlled by the cloud provider. The communication channel between the NTMPs in different NetworkAgents is modeled as the resource object of ComplexNetworkSim, which is secure and reliable. The untrusted platform of the cloud provider consists of a Verifiable Computation Unit (VC Unit) and a Communication Unit (Comm. Unit). Pinocchio is installed on the VC Unit of the cloud provider, and the VC Unit communicates with the NTSS of the NetworkAgent through the Comm. Unit. In our system, we simulate the NTPM with software, i.e., there is a trusted and tamper-resistent software module with cryptographic functions and limited secure storage. The cryptographic functions are realized by the python cryptographic library PyCrypto [PyC].

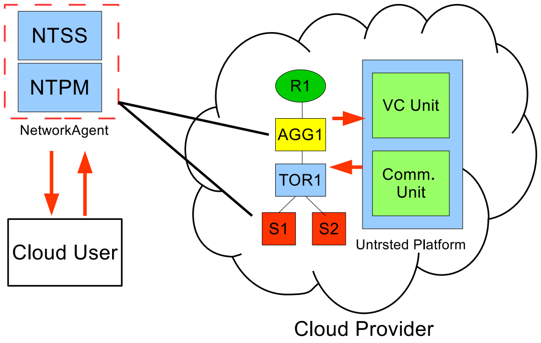

Figure 4.8: Prototype of Topology Infrastructure

4.5.2 Case Study

We illustrate the prototype and simulation with a simple case study. Suppose there is a cloud provider, denoted by CS, and a cloud user, denoted by CU. CU wants to deploy a cloud service on CS. CU requires the following set of properties, denoted by p = p_1, p_2; the failure probability of the service, denoted p_1, is less than 0.01, and the communication bandwidth between the servers controlled by CU, denoted p_2, is at least 1Gbps. There is a trusted third party that determines a reasonable solution to calculate the properties from a given cloud user topology infrastructure. Suppose the trusted third party specifies that the algorithm to calculate the failure probability of the service, denoted A_1, will be the failure sampling algorithm in PSRA [XFF13]. The trusted third party determines a heuristic algorithm, denoted A_2, to calculate the bandwidth between the servers by checking whether the total number of routers and aggregate routers in the cloud user infrastructure is at least 4. Then CU and CS agree on the algorithms of the trusted third party, and the trusted third party combines the two algorithms into a single algorithm $A = A_1 \cup A_2$. After signing and passing algorithm A to CU and CS, the trusted third party stays offline, not involving itself further in the attestation process.

After obtaining the node list of the cloud user infrastructure and verifying the System Architecture with DAA or PBA, the NTPMs select randomly a master NTPM to initiate and execute the topology measurement protocols with the assistance of NTSS. The master NTPM coordinates with the cloud platform to obtain the virtual topology information and communicates with other NTPMs to obtain the physical topology information. Then the master NTPM executes the Topology Attribution Protocol to compute the attribution between physical and virtual topology. In the next step, the master NTPM executes a Verifiable Computation protocol with the untrusted platform controlled by the cloud provider, by generating the evaluation key and verification key, signing them, and sending them to CS and CU, respectively. After receiving the verification key, the VC Unit in the untrusted platform computes the result v and the proof π and returns the result and proof to CU. CU can verify the result and the proof to learn whether the topology infrastructure satisfies the set of properties p.

Consider the following three cases of topology infrastructure. Suppose the failure probability of the servers is 0.01, and the failure probability of other nodes in the cloud user infrastructure is negligible. The topology infrastructure is as in Figure 4.9, where two VMs are launched on the same server; the relationship is indicated by the attribute link. Therefore, as long as server 1 fails, the cloud service fails. Because the failure probability of the server 1 is equal to 0.01, p_1 is not satisfied. If the topology infrastructure is as in Figure 4.10, in which the two VMs are launched on two different servers, but the two servers are connected only by one aggregate router and one router, then p_2 fails to be satisfied. Figure 4.11 satisfies both p_1 and p_2. On the one hand, the two VMs are launched on two different servers; therefore, the failure probability of the server is less than the failure probability of each server, which is 0.01. On the other hand, the number of routers reaches the threshold 4; therefore, p_2 is also satisfied.

If the attestation process does not have any mistakes, CU should be able to verify that the proof is valid. If the proof is valid, and the result $v = 1$, then CU is assured that the topology infrastructure provides the set of properties p. If the proof is valid, but the result $v = 0$, then CU can be sure that p is not satisfied. Recall that the VC protocol we use has a strong correctness property: When the computation is correct, the proof is always valid. Therefore, if the proof is not valid, CU knows that something went wrong during the attestation process, and it can reject the proof and request another attestation; the mishap during attestation might have been a manipulation on the part of the cloud provider or, with negligible probability, some sort of communication error.

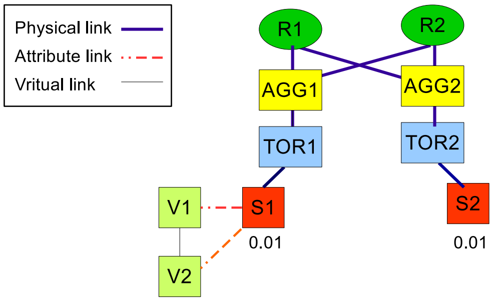

Figure 4.9: Example 1 of Topology Infrastructure

The simulation result for this case study is efficient. The running time of the three cases are 15, 11, and

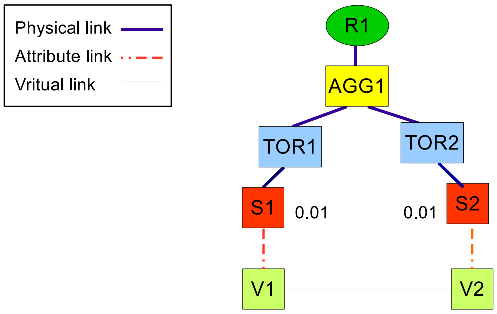

Figure 4.10: Example 2 of Topology Infrastructure

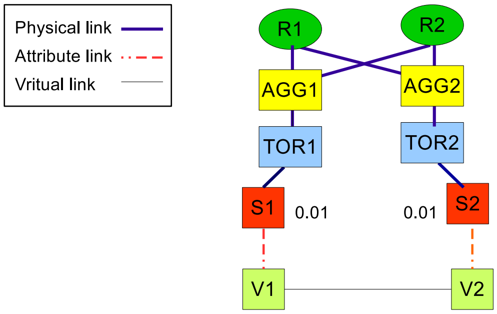

Figure 4.11: Example 3 of Topology Infrastructure

15 seconds, respectively, with the failure sampling algorithm in PSRA as A_1 and the simple loop counting algorithm as A_2. The memory usage of the NTPM is less than 1kb, and the memory usage of NTSS is less than 5kb. The simulation was done on a laptop equipped with a 2.5GHz 2-core Intel i5 CPU and 2.00GB of memory.

4.5.3 Large Scale Simulation

We evaluated our prototype through large scale simulation. We tested five cases summarized in table 4.1. Our cloud infrastructure data is based on the widely accepted three-stage, fat-tree cloud model [Lei85], scaled up to what we expect to find in a real cloud structure. The cloud user topology infrastructures were randomly selected subgraphs of the cloud infrastructure. The size of the cloud user topology infrastructure was also randomly generated, from one router, one aggregate router, one TOR, one server to hundreds of routers, aggregate routers, TORs and servers. In simulations, we generated cloud topology infrastructures as complete infrastructures, containing at least one router, one aggregate router, one TOR and one server, so the cloud user could establish a valid cloud service on them.

We first evaluated our topology measurement protocols, protocol 4 and protocol 5. We leveraged ComplexNetworkSim to establish a TCP/IP channel between each two NTPMs. We set the parameters of the simulation very conservatively. The bandwidth of the channel was only 500kb/s. The cpu frequency supporting the NTSS was set at 1.5GHz, and the data streaming capacity between the NTSS and NTPM was 1kb/s. We argue that the real network channel capacity, hardware processing power, and software processing power are much larger than these settings in real cases. The running times of the two protocols are illustrated in figure 4.12. The PTDP represents the Physical Topology Discovery Protocol, and TAP represents the Topology Attribution Protocol.

Table 4.1: Simulation Cases of Cloud User Topology Infrastructure

Configuration	Case 1	Case 2	Case 3	Case 4	Case 5
# of switch ports	4	8	16	24	48
# of core routers	8	16	32	156	96
# of agg switches	16	32	128	288	1152
# of ToR switches	24	64	512	1024	2484
# of servers	32	128	1024	4812	13824
Total # of components	84	248	1712	6304	17604

From Figure 4.12, we observe that the performance of the topology measurement protocols is in general practical for offline services in the large-scale settings. For personal cloud users, such as case 1 and 2, the topology measurement can be finished within 10 seconds. For enterprise cloud users, such as case 3 to 5, the cloud user infrastructure that consists of hundreds of or even thousands of machines can be measured in 12 minutes to 20 hours. The running time grows polynomially (but superlinearly) with the size of the infrastructure; in a larger infrastructure, the NTPMs need to communicate with more neighbors to measure the connectivity. The majority of the time was spent on PTDP, which takes around 80% to 90% of total time. Because our simulation settings were very conservative, the performance of a real implementation for a cloud platform might be much more better.

We also evaluated the attestation protocol of the cloud user topology infrastructure for the five cases. The machine that ran the untrusted platform with VC unit and Comm. unit was a laptop with 2.5GHz 2-core Intel i5 CPU and 2.00GB of memory. The set of properties attested to was that the probability that no server is accessible by the cloud user was less than a threshold, and the algorithm that the cloud user and provider agree on was the Failure Sampling Algorithm in PSRA [XFF13]. The running time is illustrated in figure 4.13, and the memory usages of the NTPM, the NTSS and the untrusted platform are illustrated in figure 4.14.

In Figure 4.13, we observe that the running time of the attestation for the large-scale cloud settings are practical as an offline service, ranging from 40 seconds to less than 30 hours, with increment of the

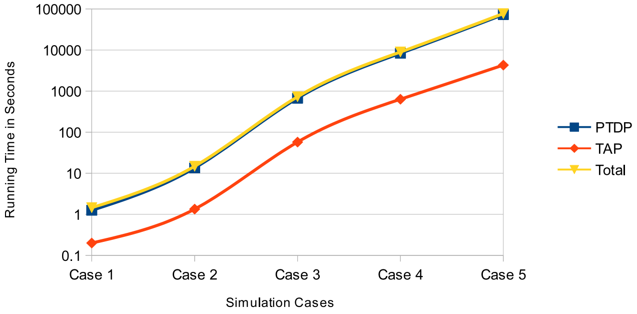

Figure 4.12: Running Time of Topology Measurement Protocols

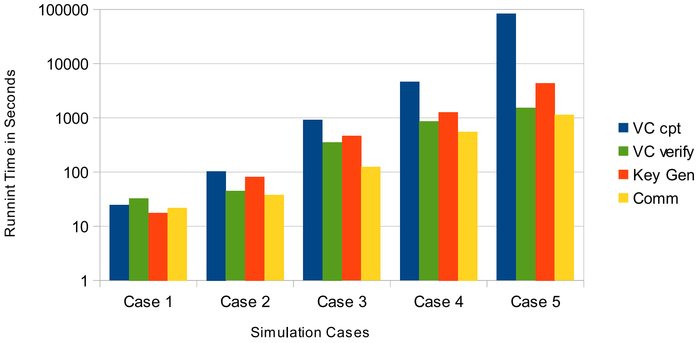

Figure 4.13: Running Time of Topology Attestation Protocols

topology infrastructure size. The VC cpt, VC verify, Key Gen and Comm represent the computation time of the verifiable computation, verification time of the verifiable computation, the key generation time, and the communication overhead, respectively. We can observe that the VC cpt absorbs the majority of the time and grows faster with the size of the cloud user topology infrastructure than other running times.

Figure 4.14 demonstrates that our design effectively transfers the majority of the memory burden to the untrusted platform, thereby reducing the memory usages of the NTPM and NTSS. According to the figure, the NTPM's memory usages were less than 10% of the memory usage of the untrusted platform, and the NTTS's were less than one third of the memory usage of the untrusted platform. With the increment of the size of the cloud user infrastructure, the percentages of memory usage of the NTPM and the NTSS decreases compared to untrusted platforms, which would be beneficial to the hardware based design of NTPM and NTSS in real cloud platforms.

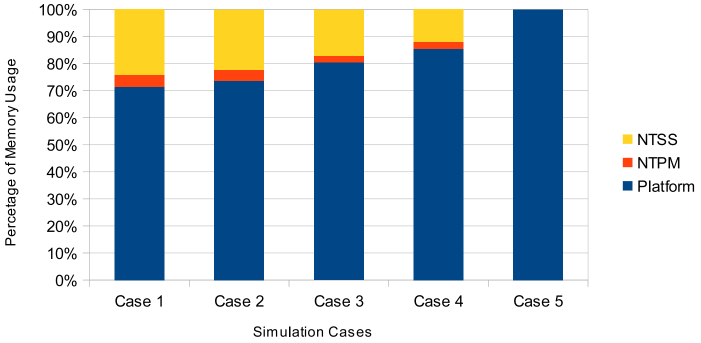

Figure 4.14: Memory Usage Comparison of Topology Attestation Protocols

Chapter 5

On Virtual-Machine Reallocation in Cloud-scale Data Centers

5.1 Introduction

When an unexpected and sudden event occurs, such as a security break-in, computation and data located on the host or hosts that are affected has to be secured. While much of the data is replicated and stored in encrypted form, there is still ongoing computation on the hosts that is not yet backed up or replicated. Moreover, there is plaintext code and data in the memories (DRAM) of the hosts. Ideally, this computation and data must be moved promptly (and local hosts resources scrubbed) to ensure confidentiality, integrity, and availability.

Such prompt movement of computation and data can be achieved through virtual-machine (VM) migration. Many techniques and optimizations have been presented that deal with movement of VMs; see, for example, [CFH$^+$05, NLH05, HG09, DWG11, WSVY07]. Moreover, efficient networking and various resiliency features, such as data replication, can speed up the migration process. All of this earlier work focuses on how to perform movement of code and data from one server to another. It does not, however, consider how to select which servers the VMs should be transferred to or what the minimal cost, i.e. migration time, to migrate these VMs is. We believe that the VM-reallocation problem is at least as important as the VM-migration problem itself, especially in the "sudden and unexpected event scenario. An efficient VM-migration mechanism is of little use if suitable servers with available resources cannot be found efficiently.

We formally define the virtual-machine reallocation problem (VMRAP) as an optimization task. The input is the topology of the data-center network, the resources available in each server, and the data-transmission rates of all channels in the network. The optimal solution is a decision vector that indicates to which server each VM in danger should be migrated. We show that the VMRAP is NP-hard; an optimal solution cannot be found efficiently, given the size of today's cloud data centers.

Consequently, we propose a two-layer heuristic scheme to solve the VMRAP efficiently. We group secure servers[1] into secure resource pools, calculate the decision vector based on secure resource pools first, and compute the final decision vectors within each secure resource pool. We evaluate our scheme using large data sets generated randomly using the fat-tree model [Lei85] of cloud data-center networks. We compare the results obtained using our scheme to optimal solutions (computed by a slow algorithm that could not be used in an emergency) and to those obtained by a naïve randomized scheme. The comparison shows that the accuracy of the two-layer scheme is high; it effectively overcomes the drawbacks of the naïve randomized scheme.

5.1.1 VM Allocation vs. Reallocation

VM-allocation problems have received a lot of attention recently [MPZ10, BB10, JLH$^+$12, SZL$^+$11, MSY12, AL12]. The VMRAP differs in at least two respects from most of the VM-allocation problems that have been studied. First, we consider the VM target-selection problem after the occurrence of unexpected events, while VM placement is done during normal operation of cloud-computing systems. We need to react quickly and calculate the VM reallocation in a very short time before the unexpected event has negative consequences. Second, we consider how to reallocate the VMs from servers in danger to secure servers, instead of launching new VMs and placing them onto available servers. We claim that the reallocation problem is more difficult for at least two reasons. One is that, after the occurrence of unexpected events, the resources of secure servers may be scarce, and reallocation cannot be allowed to affect the normal operations of other secure VMs on secure servers. A second reason is that the number of VMs in the reallocation pipeline is more volatile than VMs in the VM allocation pipeline, because the unexpected events could cause a large number of VMs in several servers or even several racks to be in danger; they all have to be reallocated quickly and simultaneously.

5.1.2 Random Selection and Hot Spares

The VMRAP could be approached through a naïve random-selection scheme in which VMs are reallocated to randomly selected secure servers. This solution has the major drawbacks of unpredictability and of lack of optimality. Randomly selecting servers to which VMs should be migrated means that a server already fully

[1] We use term "secure servers" to denote servers that are not in danger in an emergency situation. The emergency may be equipment failure, physical security breach, etc., depending on the types of threats that the users are worried about.

occupied could be selected, and random selection would potentially have to be done many times before a target can be found. This is especially time-consuming when servers are running at a high utilization rate.

Another alternative is to use hot spares or other under-utilized servers that are kept as backup for emergencies. However, there are two drawbacks of this approach. First, many sudden and unexpected events, such as physical break-ins and attacks, affect both the servers and their hot spares, because the hot spares are often deployed within close range of the servers. Therefore, the hot spares cannot be considered reliable and secure migration targets in these situations. Second, the cost of keeping unutilized servers around may be very high. Even if that is not a concern, such servers are likely to be idle or powered down to save energy. Selecting such a server may be quick, because it is known a priori which server is the hot spare that will have available resources, but the overall migration will not be quick because of the lag time to bring the server back up before VM migration can occur.

5.2 Cloud-scale Data Centers

We explore the migration-target selection problem for cloud-scale data centers. A typical logical data center architecture is shown in Figure 5.1; it is based on the latest (fall 2013) OpenStack architecture. Within an OpenStack system, our work would enhance the scheduler component (*nova-scheduler*) and the data-orchestration component (*nova-conductor*); it would use the *nova database*. Optionally, a new *nova-guard* component could be introduced.

The *nova-scheduler* is responsible for the scheduling of VMs on different compute nodes (or servers we use the terms interchangeably). The compute nodes, via the *nova-conductor*, report their status to the *nova database*. This includes the total resources of the compute node as well as the currently available resources.

Today, the scheduler receives commands from the Dashboard (via the nova-api and the Queue). These commands come from the administrator, who may instruct the scheduler to allocate a new VM, terminate an existing VM, etc. However, there is currently no autonomous mechanism for reallocation of VMs. Reallocation is done manually by the administrators via the Dashboard or the command line.

To enable the reallocation of resources after a sudden and unexpected event, the nova-scheduler has to be enhanced with an algorithm that calculates where (and how fast) VMS can be reallocated. The triggers for reallocation can be obtained from status updates of the servers and read from the database. In particular, a compute node (via nova-conductor) may update status about some internal event. Also, the new nova-guard component may update some information about security events in the data center (e.g., breaches by

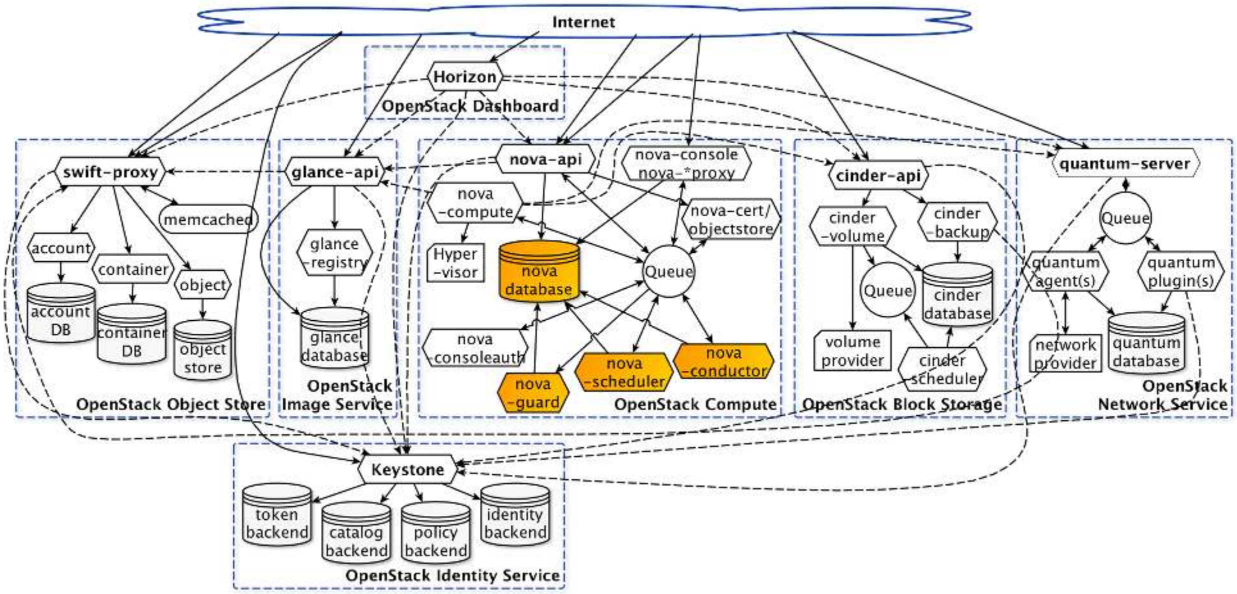

Figure 5.1: Logical architecture of a data center, modeled after OpenStack "Grizzly" logical architecture [Gri]. The highlighted elements would be modified to integrate our reallocation code into OpenStack. The modified parts fall into one of the seven core components; nova-guard is a new, optional part that we propose. The blue dashed boxes logically group parts of each of the seven core components. The solid lines represent API calls from outside of the core components; they are routed on the public network. The dashed lines represent API calls between the core components; they are routed on the management network.

unauthorized parties). These events are logged as new status of the nodes and can be read from the database by the nova-scheduler. Given the current information about the hosts and resources, our work focuses on answering *where to* the migration should happen after an unexpected event; this would be a new part of the scheduler. We show that this is a computationally difficult problem that cannot be solved efficiently using standard optimization techniques because of the considerable input size.

Our work assumes a data center with a networking topology similar to one shown in Figure 5.2; it is based on current, standard design principles [AF12]. The network is organized as a fat tree in which the leaves are the server racks (and the top-of-rack switch at each rack). Within server racks, there are highly interconnected servers. The racks in turn are connected together through a series of aggregation switches and aggregation routers. There are also core routers that connect the data center to the internet. In Section 5.4.4, we leverage the fat-tree model to help design our efficient, heuristic algorithm for the migration-target allocation problem.

Our work is focused on intra-data-center networking - the resources will be reallocated within the data center when the unexpected or sudden event occurs. Within the data center, we assume 1Gbps VM-migration bandwidth between servers in a rack and 0.1Gbps VM-migration bandwidth between racks. We use 125,000

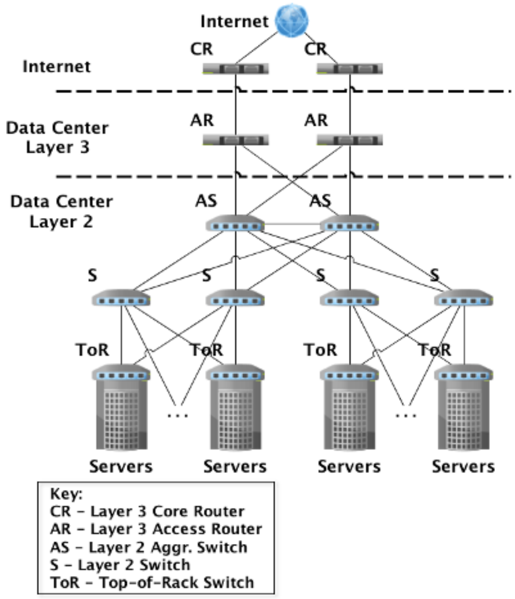

Figure 5.2: Typical data center network, modeled after [GHJ+09].

bytes/s as the real transmission rate, because there may be other traffic flows in the data center network. These are very conservative assumptions. Our algorithm can easily be updated with different values for bandwidths among servers and among server racks. Recent PortLand [NMPF+09] and VL2 [GHJ+09] work presents quite fast network designs that may facilitate much faster migration times. Our focus, however, is on the migration-target selection problem. Interestingly, even with these very conservative assumptions, the calculated migration time for a small number of in-danger servers is less than the time needed to select which hosts the computation and data should be moved to.

5.3 The VM-Reallocation Problem

When a threat is detected (imminent hardware failure, physical security breach, etc.), a reallocation algorithm has to calculate on-the-fly where to move VMs. There is a set of servers that are in danger and a (much larger) set of servers not in danger. The not-in-danger servers, however, are themselves occupied with VMs; so the VMs on the in-danger servers cannot just be moved to other servers arbitrarily. Given the resources taken up by the VMs in danger, the algorithm has to find enough not-in-danger servers *with sufficient available resources*. VMs from the same in-danger servers may be migrated to different not-in-danger servers.

5.3.1 Threat Model

We assume that resources in a data center are protected by number of standard measures. Encryption of persistent storage protects data at rest. Replication of storage ensures availability. Moreover, we assume that servers contain code and local data but that most of the data are stored on network-attached-storage servers. The management infrastructure is trusted, and in particular we rely on correct and secure implementation of the scheduler and VM-migration algorithms. We expect the management infrastructure to be able to detect an imminent threat and trigger our algorithm.

Protections have to be triggered by all imminent threats, because, at any time, there are code and data executing in VMs that have not yet been backed up or replicated. Furthermore, today's server architectures do not use hardware encryption of DRAM, and there are plaintext data and code in memory. As a result, when there is an unexpected or sudden event, the VM has to be quickly relocated to a different host. This involves moving the contents of the memory (DRAM) and scrubbing the source host once all code and data have been moved to a target host.

5.3.2 Threat Examples

Data-center operators today are well prepared for a number of events that may interrupt operation. There are, however, cases of unexpected events or ones that can not be predicted. We list some examples here of events that require on-the-fly reallocation of resources.

Equipment failure is one of the examples. Much of todays hardware includes many mechanisms, such as the SMART (Self-Monitoring, Analysis and Reporting Technology) failure-prediction system currently available in many disk-drives [Sma]. These mechanisms warn of a potential upcoming failure. If there are servers that do not have redundant drives, or if immediate hot swapping of physical drives is not possible, the ongoing computation and data have to be migrated from the server. This can be done remotely, but it currently requires the involvement of an administrator. With our proposed algorithm for reallocation of VMs, once an event is detected, the reallocation is calculated automatically and can be executed without involvement of an administrator. This can reduce costs by reducing human involvement. Also, physical maintenance can be delayed (e.g., until the next business day), which reduces the size of the on-site staff.

Reliability for virtual appliances is another example. While large companies such as Google may design their cloud applications to be resilient and to tolerate failure of individual nodes, there are many casual customers who use “off-the-shelf” virtual appliances that are stand-alone VMs. When a data center is running some portion of VMs that do not have built-in resiliency, there is no other option but to migrate the VMs to avoid loss of data and computation when an unexpected event occurs.

Security events are yet a different example. A hypervisor may detect, through techniques such as VM introspection [GR^{+}03], that one of the VMs poses a security threat. One possible action in such a situation is to move other VMs away. While strong isolation should keep the virtual machines separate, it cannot prevent a malicious party that controls a VM from performing a side-channel attack [ZJRR12]. Thus, in situations when a suspicious VM cannot be terminated immediately, strong isolation may not be able to protect the other VMs, and the best option is to migrate other VMs to a safe system.

Recent work has shown that physical security breaches need to be detected and acted upon [SJCL12]. When an unauthorized individual enters a server room, opens a server rack, etc., the computation on the servers is in danger. Here, the choices are encrypt, delete data, or migrate VMs. Only migration of VMs away from the affected servers (along with scrubbing the servers) will ensure confidentiality, integrity, *and* availability. The proposed *nova-guard* can be used to supply information about physical security events that will trigger reallocation.

5.3.3 Problem Formulation

In our model, the data center is divided into a set R of server racks. Suppose that there are N_R racks, i.e., that $|R| = N_R$. Each rack $r \in R$ is occupied by a set S_r of servers. On each $s \in S_r$, there is a set V_s of VMs running. Each s has a number of total capacities for a variety of resources: microprocessor[2], c_s^{Cap}; memory, m_s^{Cap}; and disk, d_s^{Cap}. Each virtual machine v uses a number of resources: microprocessors, c_v; memory, m_v; and disk, d_v. For each server s, we denote the available resources of microprocessors, memory, and disk by c_s^{ava}, m_s^{ava} and d_s^{ava}, respectively. Consequently, we have $c_s^{ava} = c_s^{Cap} - \sum_{v \in V_s} c_v$, $m_s^{ava} = m_s^{Cap} - \sum_{v \in V_s} m_v$ and $d_s^{ava} = d_s^{Cap} - \sum_{v \in V_s} d_v$

We consider the situation in which an unexpected event occurs. Suppose that there is a set S_D of servers that are in danger after the unexpected event; we need to move the VMs running on these servers to the set of secure servers, denoted by S_S. For each VM v running on one of the in-danger servers $s_D \in S_D$ and any not-in-danger (i.e., secure) server $s \in S_S$, we define an indicator variable $I_{s_D,s}^{v} \in \{0,1\}$ that captures the migration decision of v with respect to s. When $I_{s_D,s}^{v} = 1$, VM v is moved from s_D to s. If $I_{s_D,s}^{v} = 0$, VM v is not moved to s. We denote by $t_{s_D,s}^{v}$ the migration time of VM v running on an in-danger server in s_D to secure server $s \in S_S$ as the total time needed to move the memory and disk storage of v to s. This migration time is $t_{s_D,s}^{v} = \frac{(m_v+d_v)I_{s_D,s}^{v}}{BW_{s_D,s}}$, where $BW_{s_D,s}$ is the bandwidth of the data-center network from server s_D to server s. In

[2] Because microprocessors used to be called “central processing units, we use the variable name c.

real data-center networks, the bandwidth may vary. However, because in VMRAP the reaction time for VM migration is very short, we assume that the bandwidth is constant during that time. We leave consideration of variable bandwidth to future work.

To solve VMRAP, we wish to compute the value of the indicator variables $I^v_{s_D,s}$ for each VM v running on an in-danger server to minimize the total migration time of the VMs.

$$\min_{I^v_{s_D,s}} \sum_{s_D \in S_D} \sum_{v \in V_{s_D}} \sum_{s \in S_S} t^v_{s_D,s} \tag{5.1}$$

s.t. for any $s_D \in S_D$ and any $v \in V_{s_D}$:

$$\sum_{s \in S_S} I^v_{s_D,s} = 1 \tag{5.2}$$

for any $s \in S_S$, any $s_D \in S_D$, and any $v \in V_{s_D}$:

$$I^v_{s_D,s} \in \{0,1\} \tag{5.3}$$

for any $s \in S_S$:

$$\sum_{s_D \in S_D} \sum_{v \in V_{s_D}} m_v I^v_{s_D,s} \leq m^{ava}_s \tag{5.4}$$

for any $s \in S_S$:

$$\sum_{s_D \in S_D} \sum_{v \in V_{s_D}} c_v I^v_{s_D,s} \leq c^{ava}_s \tag{5.5}$$

for any $s \in S_S$:

$$\sum_{s_D \in S_D} \sum_{v \in V_{s_D}} d_v I^v_{s_D,s} \leq d^{ava}_s \tag{5.6}$$

5.3.4 Computational Complexity

The VMRAP can be represented as a binary-programming problem [NW88], the decision version of which was one of the Karp's 21 NP-complete problems [Kar72]. ("Binary programming is also known as "0-1 integer programming.) Moreover, any instance of binary programming can be interpreted as a VMRAP instance; so VMRAP is NP-hard (or, if formulated as a decision problem rather than an optimization problem, NP-complete). If the constraint matrix has the total unimodularity property [NW88], one can relax the binary program to a normal linear program and compute an optimal solution efficiently. Otherwise, the algorithms that solve this problem exactly, including those based on cutting-plane methods [Kel60], are not efficient. In our case, the constraint matrix is derived from the complicated structure of the data-center network and is *not* typically unimodular.

We used linear-programming and binary-programming algorithms to solve large VMRAP instances with commercial and open-source optimization tools [CVX, Gur]. We first implemented a binary-programming solver using CVXOPT [CVX] and the GNU Linear-Programming Kit (GLPK) [GLP] to solve those in instances in a centralized manner. The algorithm is implemented in Python version 2.7.5 and CVXOPT version 1.1.5. We ran it on a workstation equipped with a 6-core 2.4GHz Intel Xeon 2.4GHz CPU and 32GB memory. We generated a data-center network topology using the fat-tree model, setting the network bandwidth as 1Gbps within racks and 0.1Gbps between racks and generating servers with the configuration of Dell PowerEdge R910 Rack Servers [Del] and VMs with resource requirements that are average for Amazon EC2 instances [Amab].

Table 5.1 shows the results of using CVXOPT and Table 5.2 the results of using Gurobi.

Table 5.1: Results of binary programming using CVXOPT

Realloc. Calculation Time (s)	**Calc. Moving Time (s)**	**Racks**	**Serv. / Rack**	**VM / Serv.**	**Vuln. Serv.**
0.1	263	10	40	64	1
7	527	10	40	64	2
70	2636	10	40	64	10
1.5	263	50	40	64	1
3.1	527	50	40	64	2
30	2636	50	40	64	10
3	263	100	40	64	1
7	527	100	40	64	2
65	2636	100	40	64	10
36	263	500	40	64	1
73	527	500	40	64	2
587	2636	500	40	64	10
674	263	2500	40	64	1
1381	527	2500	40	64	2
>7200	n/a	2500	40	64	10

Table 5.2: Results of binary programming using Gurobi

Realloc. Calculation Time (s)	**Calc. Moving Time (s)**	**Racks**	**Serv. / Rack**	**VM / Serv.**	**Vuln. Serv.**
1.7	263	10	40	64	1
3	527	10	40	64	2
15.4	2636	10	40	64	10
8.56	263	50	40	64	1
17.54	527	50	40	64	2
108	2636	50	40	64	10
20	263	100	40	64	1
42	527	100	40	64	2
290	2636	100	40	64	10
251	263	500	40	64	1
589	527	500	40	64	2
>7200	n/a	500	40	64	10
>7200	n/a	2500	40	64	1
>7200	n/a	2500	40	64	2
>7200	n/a	2500	40	64	10

From the tables, we see that neither academic nor commercial solvers can solve the type of binary-programming instances we have in real time when we scale up the input size to realistic data-center sizes. When we scale up the size of the inputs, the running time of the algorithm exceeds the migration time of the VMs, reaching more than 2 hours.

Linear-programming relaxation is a natural and standard approach to explore in this context. The relaxed linear-programming problem is polynomial-time solvable but cannot guarantee that a complete VM is sent to one secure server. Instead, some portion of the VM may be sent to one server and other portions to other servers. How to “split VMs and send them to different servers while making them run normally is another interesting research problem that is beyond the scope of this work. Without splitting technology, the split VMs are not available until they are merged back together; therefore, the relaxation to linear programming may sacrifice the availability of VMs while improving the efficiency of VM reallocation.

To relax the binary-programming problem to a linear-programming problem, we relax constraint 5.3 to continuous constraint 5.7:

$$0 \leq I^{v}_{s_D,s} \leq 1 \tag{5.7}$$

Table 5.3 shows results obtained with linear programming. The algorithm is implemented in Python version 2.7.5 and CVXOPT version 1.1.5. The results using Gurobi are shown in Table 5.4.

Table 5.3: Results of linear programming using CVXOPT

Realloc. Calculation Time (s)	**Calc. Moving Time (s)**	**Racks**	**Serv. / Rack**	**VM / Serv.**	**Vuln. Serv.**
0.1	263	10	40	64	1
7	527	10	40	64	2
71	2636	10	40	64	10
1.4	263	50	40	64	1
3	527	50	40	64	2
33	2636	50	40	64	10
2.5	263	100	40	64	1
6.5	527	100	40	64	2
66	2636	100	40	64	10
33	263	500	40	64	1
70	527	500	40	64	2
580	2636	500	40	64	10
660	263	2500	40	64	1
1344	527	2500	40	64	2
>7200	n/a	2500	40	64	10

As Tables 5.3 and 5.4 show, neither academic nor commercial solvers are fast enough for large problem instances, even when we are willing to tolerate some loss of availability of VMs. When we scale to realistic

Table 5.4: Results of linear programming using Gurobi

Realloc. Calculation Time (s)	**Calc. Moving Time (s)**	**Racks**	**Serv. / Rack**	**VM / Serv.**	**Vuln. Serv.**
1.5	263	10	40	64	1
3	527	10	40	64	2
15	2636	10	40	64	10
7.53	263	50	40	64	1
15.8	527	50	40	64	2
80	2636	50	40	64	10
16	263	100	40	64	1
33	527	100	40	64	2
163	2636	100	40	64	10
84	263	500	40	64	1
168	527	500	40	64	2
973	2636	500	40	64	10
464	263	2500	40	64	1
1375	527	2500	40	64	2
>7200	n/a	2500	40	64	10

data-center sizes, the running times grow to more than 2 hours, i.e., can be as bad the binary-programming problem running time. Therefore, we need a new approach; one is presented in the next section.

5.4 An Efficient, Decomposed, Two-Layer Approach

To solve the VMRAP efficiently, we propose a decomposed, two-layer heuristic approach. Our scheme leverages the special structure of current data-center networks, and experiments demonstrate that the heuristic approach is both accurate and efficient.

5.4.1 Overview of the Two-Layer Approach

We decompose a very large optimization problem into a number of smaller sub-problems. We first partition all the secure servers into a number of secure resource pools and partition all the in-danger VMs into a number of in-danger pools. The available resource capacities of each secure resource pool are the summation of the resource capacities of the individual servers in the pool, while the resources needed by each in-danger pool are the summation of the resources needed by the individual VMs in the pool. We formulate a first-layer optimization problem to split the in-danger pools and fit them into the secure resource pools with minimal migration time, while satisfying the resource constraints of the secure resource pools. We define the migration time of an in-danger pool as the total number of bytes that must be transferred to move the pool, divided by the average bandwidth from the servers running the VMs in the in-danger pool to the servers in the secure resource pools. An overview of the two-layer approach is given in Figure 5.3.

After the first-layer optimization, each secure resource pool gets assigned some of the resource requests

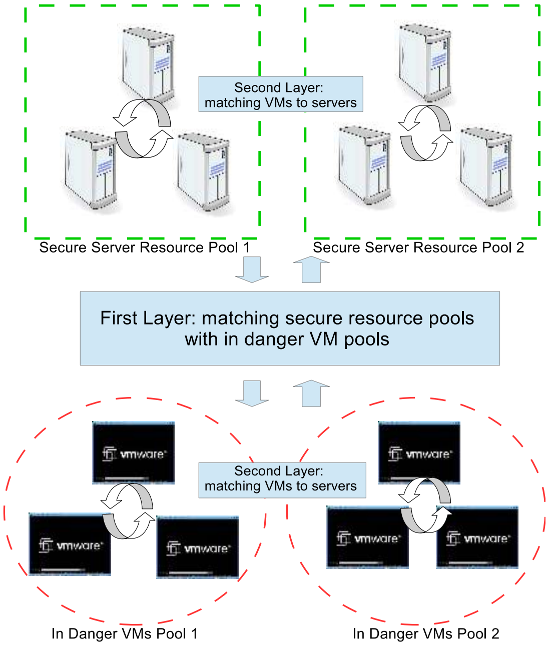
Second Layer: matching VMs to servers
Secure Server Resource Pool 1
Secure Server Resource Pool 2
First Layer: matching secure resource pools with in danger VM pools
vmware
Second Layer: matching VMs to servers
In Danger VMs Pool 1
In Danger VMs Pool 2

Figure 5.3: Overview of the two-layer approach

from the in-danger pools. In the second layer, each secure resource pool decides how to distribute the resource requests from the in danger pools to its servers by solving a small-sized optimization problem to minimize the total expected migration time. The secure resource pools can perform the optimization simultaneously, which improves the efficiency of the global optimization problem. After all the secure resource pools complete the optimization, the results are sent back to the servers in the in-danger pools, which can decide how to send the VMs to the secure servers to which they were assigned.

Recall that, in the binary-programming and linear-programming problems, the number of variables is proportional to the number of VMs in danger and the number of secure servers. In the two-layer approach, the size of the optimization problem is proportional to the number of in-danger pools and secure resource pools, which can be far smaller by design. In addition, the fact that the second-layer optimizations can be executed in parallel further reduces the running time.

5.4.2 First-Layer Optimization Problem

Let $G : S_S \rightarrow P_S$ be a partition of secure servers, S_S, into a set of secure resource pools, P_S. Recall that, by definition, a partition must be complete and mutually exclusive, which means that one secure server must belong to one and only one pool in P_S, i.e., $S_S = \cup_{p \in P_S} p$ and $p_i \cap p_j = \varnothing$ for any $p_i, p_j \in P_S$ and $p_i \neq p_j$. Therefore, there are no in-danger VMs that are sent to the same secure server twice, and no secure servers are omitted as *potential* targets of the in-danger VMs. Let $V_D = \{v : v \in s \text{ and } s \in S_D\}$ be the set of VMs in danger. Let $F : V_D \to P_D$ be a partition of V_D, the in-danger VMs, into a set of in-danger pools, P_D.

The optimization problem of layer 1 can be formulated as follows. The algorithm minimizes the total migration time of the in-danger pools with respect to the resource constraints of each secure resource pool. Let $I_{p_s}^{p_d}$ be a variable in the minimization problem for in-danger pool $p_D \in P_D$ and secure resource pool $p_S \in P_S$; this means that in-danger pool p_D sends percentage of $I_{p_S}^{p_D}$ to secure resource pool p_S. For any in-danger pool or secure resource pool p, the resource needed and the resource available are x_p and x_p^{ava}, respectively, where $x \in \{c,m,d\}$. Denote the migration time of in-danger pool p_D to secure resource pool p_S by $t_{p_S}^{p_D} = \frac{(c_{p_D}+m_{p_D})I_{p_s}^{p_d}}{BW_{p_D,p_S}}$, where $BW_{p_D,p_S} = \frac{\sum_{s_D \in p_D, s \in p_S} BW_{s_D,s}}{|p_D||p_S|}$.

$$\min_{I_{p_s}^{p_d},\ p_d \in P_D,\ p_s \in P_S} \sum_{p_D \in P_D} \sum_{p_S \in P_S} t_{p_S}^{p_D} \tag{5.8}$$

s.t.

for any $p_D \in P_D$:

$$\sum_{p_S \in P_S} I_{p_S}^{p_D} = 1 \tag{5.9}$$

for any $p_D \in P_D$ and $p_S \in P_S$:

$$0 \le I_{p_S}^{p_D} \le 1 \tag{5.10}$$

for any $p_S \in P_S$:

$$\sum_{p_D \in P_D} m_{p_D} I_{p_S}^{p_D} \le m_{p_S}^{ava} \tag{5.11}$$

$$\sum_{p_D \in P_D} c_{p_D} I_{p_S}^{p_D} \le c_{p_S}^{ava} \tag{5.12}$$

$$\sum_{p_D \in P_D} d_{p_D} I_{p_S}^{p_D} \le d_{p_S}^{ava} \tag{5.13}$$

5.4.3 Second-Layer Optimization Problem

The second-layer optimization for each secure resource pool $p_S \in P_S$ can be formulated as follows. Each secure resource pool dispatches the resource requests from the in-danger pools to its servers by minimizing the total migration time. Let $I_s^{p_D}$ be the variable for in-danger pool $p_D \in P_D$ and server s in the secure resource pool p_S, i.e., $s \in p_S$. Let $BW_{p_D,s}$ be the average bandwidth between in-danger pool p_D and secure server s, i.e., $BW_{p_D,s} = (\sum_{s_D \in p_D} BW_{s_D,s})/|p_D|$. The migration time is defined as $t_{p_D,s} = ((m_{p_D} + d_{p_D}) I_{p_S}^{p_D} I_s^{p_D})/BW_{p_D,s}$. Now, the minimization problem becomes:

$$\min_{I_s^{p_D} \ for \ p_D \in P_D} \sum_{p_D \in P_D} \sum_{s \in p_S} t_{p_D,s} \tag{5.14}$$

s.t.

for any $p_D \in P_D$:

$$\sum_{s \in p_S} I_s^{p_D} = 1 \tag{5.15}$$

for any $p_D \in P_D$ and $s \in p_S$:

$$0 \leq I_s^{p_D} \leq 1 \tag{5.16}$$

for any $s \in P_S$:

$$\sum_{p_D \in P_D} m_{p_D} I_{p_S}^{p_D} I_s^{p_D} \leq m_s^{ava} \tag{5.17}$$

$$\sum_{p_D \in P_D} c_{p_D} I_{p_S}^{p_D} I_s^{p_D} \leq c_s^{ava} \tag{5.18}$$

$$\sum_{p_D \in P_D} d_{p_D} I_{p_S}^{p_D} I_s^{p_D} \leq d_s^{ava} \tag{5.19}$$

After each secure resource pool's optimization finishes, each in-danger pool decides which VM sends to which secure server based on the results of first-layer optimization and secure resource pool's optimization, i.e., $I_{p_S}^{p_D}$ and $I_s^{p_D}$. To avoid split VMs as much as possible, each in-danger pool first tries to fit a set of complete VMs to each server. The set $V_{p_D}^C$ of VMs for server s satisfies:

$$\sum_{v \in V_{p_D}^C} m_v \leq m_{p_D} I_{p_S}^{p_D} I_s^{p_D} \tag{5.20}$$

$$\sum_{v \in V_{p_D}^C} c_v \leq c_{p_D} I_{p_S}^{p_D} I_s^{p_D} \tag{5.21}$$

$$\sum_{v \in V_{p_D}^C} d_v \leq d_{p_D} I_{p_S}^{p_D} I_s^{p_D} \tag{5.22}$$

Finally, each in-danger pool splits the remaining VMs and fits them into the resources left on each of the servers in its assigned not-in-danger pool.

5.4.4 How to Partition the Problem

We want to minimize the loss of optimality when we partition the secure servers and in-danger VMs to decompose the optimization problem into two layers. The two-layer approach loses optimality in two ways. In the first layer, average bandwidth of the servers in the secure resource pools and VMs in the in-danger pools is used to replace individual bandwidths; therefore, if the VMs in the same in-danger pool connect to secure servers with different bandwidth, there will be loss of optimality. In the second layer, all VMs in the same in-danger pool tend to be reallocated to the same set of servers according to the resource requests from the first layer. The globally optimal solution, however, may be to reallocate these VMs to different set of servers.

We propose a heuristic that leverages the structure of the data-center network: Group all secure servers in a rack together as a secure resource pool, and group VMs in the same server in danger into an in-danger pool. The benefits are: 1) The bandwidths and average bandwidth between one server in danger and other secure servers in the same rack tend to be good, because all the secure servers in the same rack connect to the same set of ToR switches, leading to homogeneous connectivity to the other parts of the data-center network. Therefore the loss of optimality in the first layer is small. 2) The VMs in the same server tend to share bandwidth toward other servers in the data-center network; therefore they are highly likely to be reallocated to the same set of servers in the globally optimal solution. 3) The number of racks and the number of servers in danger tend to be relatively small (much smaller than the total number of secure servers and VMs in danger), which improves the efficiency of the optimization process significantly.

5.5 Experimental Evaluation of the Two-Layer Approach

In this section, we evaluate our two-layer approach using simulation. The hardware and software configurations used for this simulation are identical to those used for the experiments in Section 5.3.4.

5.5.1 Efficiency of the Two-Layer Approach

Table 5.5 shows results of the proposed two-layer algorithm. Our implementation solves the VMRAP in less than 4 seconds even when we scale the size of the data center to 2500 racks, 40 servers per rack, and 64 VMs per server (that is over 6 million possible VMs total) with 10 servers in danger. This is efficient enough to be used in practice.

We also compare the performance of the two-layer approach with the binary-programming and linear-

Table 5.5: Results of two-layer linear programming using CVXOPT

Realloc. Calculation Time (s)	Calc. Moving Time (s)	Racks	Serv. / Rack	VM / Serv.	Vuln. Serv.
<0.1	263	10	40	64	1
<0.1	527	10	40	64	2
1.1	2636	10	40	64	10
<0.1	263	50	40	64	1
<0.1	527	50	40	64	2
1.1	2636	50	40	64	10
<0.1	263	100	40	64	1
<0.1	527	100	40	64	2
1.1	2636	100	40	64	10
<0.1	263	500	40	64	1
0.1	527	500	40	64	2
1.2	2636	500	40	64	10
0.1	263	2500	40	64	1
0.3	527	2500	40	64	2
3.3	2636	2500	40	64	10

programming relaxation, as shown in Figure 5.4. The running times come from the Tables 5.2, 5.4, and 5.5, in the case of 10 servers in danger. The x-axis is the number of servers in the simulation, while the y-axis is the total computation time for reallocation. Binary programming and linear programming run more than 2 hours in the case of 100000 servers; so we have not plotted these points. From the figure, we can see that the performance of the two-layer approach is much better than that of integer programming or relaxed linear programming.

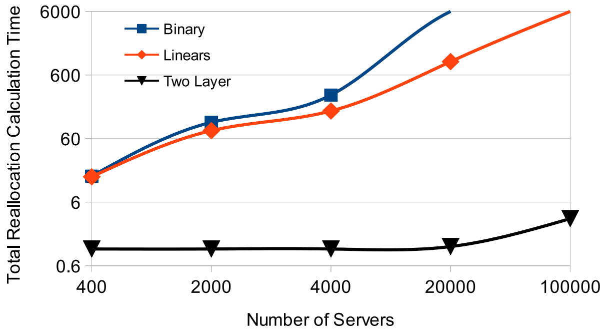

Figure 5.4: Comparison of the performance of different approaches.

5.5.2 Accuracy vs. Improved Performance

The improved performance comes at a cost; so we also test the accuracy of the two-layer approach as follows. We measure the accuracy by how close the sub-optimal migration time computed by the two-layer approach is to the globally optimal migration time and the sub-optimal migration time computed by linear-programming relaxation. We generate simulation cases of data centers with 100 racks, 40 servers per rack, 64 VMs per server, and 10 servers in danger. In addition, we generate simulations with different utilization of servers. We define the utilization ut_s of a server s as the maximal usage rate of each kind of resources, i.e., $ut_s = \max_{x \in \{c,m,d\}} \frac{x^{Cap} - x^{ava}}{x^{Cap}}$. The higher the utilization, the more scarce the available resources are in the secure servers. For each utilization point, we randomly set the resources needed by each VM based on a uniform distribution among available Amazon EC2 instances to generate 1000 samples and take the average of the migration times of the 1000 samples. We plot the accuracy of the results in Figure 5.5. The x-axis is the utilization of the servers. The y-axis is the total migration time computed by the different approaches.

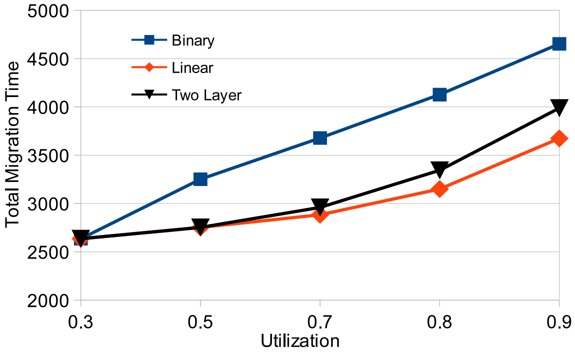

Figure 5.5: Comparison of the accuracy of different approaches.

Note that the optimal migration time computed by linear-programming relaxation should be less than or equal to the optimal time computed by global binary programming, because the linear-programming relaxation sacrifices the availability of the VMs or forces the usage of advanced VM-splitting technology to maintain the availability. Because of splitting of the VMs, the migration time of linear-programming relaxation is smaller than that of binary programming. On the other hand, the two-layer approach's running time is slightly higher than that of linear-programming relaxation in most cases, because of the loss of optimality in the decomposition.

When the utilization of servers is low, which means that the available secure resources are plentiful, the two-layer approach coincides with binary and linear programming. This is because the number of VMs in danger is far less than the capacities of the secure servers; therefore each in-danger VM can always be completely migrated to the secure server with the largest bandwidth toward the in-danger server of the VM. When the utilization increases, binary programming and linear programming diverge to different results, and the result of the two-layer approach is in the middle of the two. On average, the two-layer approach is a good approximation of binary programming and linear-programming relaxation, because the difference between the migration time of the two-layer approach and binary programming is less than 15%, and the difference between the migration time of the two-layer approach and linear programming is less than 10%.

Given this bound on the difference of the solutions, the two-layer approach can be used and its calculated migration-time output increased by 15% to ensure that the scheduler does not underestimate the migration time (leading to false choice of whether the migration is an appropriate action to take given the current threat).

5.6 Related Work

Virtualization was explored by IBM researchers as early as 1970s [MS70]. VM migration is a newer concept but has already been throughly explored. Ideas such as process migration of processes from one instance of an OS to another [MDP$^+$00] or self-migration of a whole OS instance, done by the OS itself, have been presented [HJ04]. Cold, warm, and live [CFH$^+$05] migration of whole VMs performed by the hypervisor has been proposed and explored extensively. Live migration minimizes downtime and ensures responsiveness of services; so it is often most desired. There are pre-copy algorithms, e.g. [NLH05], that copy memory pages of a VM ahead of time, and once most pages are copied execution transfers from the source to the target machine. Post-copy algorithms, e.g. [HG09], on the other hand defer the transfer of memory pages until such a page is actually accessed. Further optimizations, such as gang migration of many VMs at once, have been proposed as well [DWG11]. Using some information about content of the VM (i.e., black-box vs. gray-box) to aid in migration has been investigated as well [WSVY07]. Security aspects of migration have also been explored, e.g., use of a virtual trusted-platform module (vTPM), e.g. [DMKC11]. None of these works presents algorithms to determine which servers the VMs should be transferred to.

There are many algorithms for scheduling of resources within a single server. Just as there are schedulers inside an operating system, hypervisors have schedulers that decide when certain VMs should runs and which resources within the server they will use. Borrowed Virtual Time (BVT), Simplest Earliest Deadline First (SEDF), and fair-share proportional scheduling (Credit scheduler) are examples of schedulers in production hypervisors [CGV07]. Our work similarly could be considered a scheduler, but at the data-center level rather than at the server level. The significant difference, however, is the scale of the problem. Rather than dealing with tens of processors, we are dealing with tens of thousands, or more, of servers.

Cloud-management software such as the OpenStack contains a “scheduler. The default OpenStack scheduler, however, applies very simple policies, such as choosing the least loaded server, randomly selecting an available server, etc. [CFF12] The management software does not deal with reallocation of resources on the fly.

Chapter 6

Conclusion and Open Problems

In this work, we investigate accountability in cloud-computing and distributed computer systems. We first systematize much of the research on “accountability” in computer science and then proceed to design, implement, and evaluate new accountability mechanisms for cloud-computing and distributed computer systems. We have designed and developed a P-SRA (for “private structural-reliability auditing”) system, which can audit the cloud infrastructure to assess the reliability of cloud services without revealing the private information of the cloud providers. The P-SRA system is not only a useful accountability mechanism for cloud-computing systems but also an interesting application of secure multi-party computation (SMPC), which has not often be used for computations on complex, linked data structures. We have also developed the notion of “cloud user infrastructure attestation, and designed a framework that enables cloud-service providers to attest to cloud-service users that the resources they have provided satisfy the users’ requirements. The privacy of the cloud-service providers is protected, while the accountability of the cloud-service providers to the cloud-service users is achieved. Finally, we investigate accountability mechanisms in cloud-scale data centers. We formulate the virtual-machine reallocation as an optimization problem and explain how it differs from the general virtual-machine allocation problem. Virtual-machine reallocation is NP-hard, but we provide an efficient, two-layered, heuristic algorithm to solve it reasonably efficiently.

6.1 Systematizing Accountability in Computer Science

Our systematic consideration of many major works on “accountability” in Chapter 2 demonstrates that computer scientists have used the term to mean different things. We have organized prior work on accountability along the axes of time, information, and actions and highlighted both existing results and open questions. Interestingly, our decision to define accountability mechanisms as those that allow actions to be tied to consequences (and, in particular, allow violations to be tied to punishment) dispels the mistaken notions that accountability precludes anonymity and that it requires centralized authority.

Our systematization effort has revealed the need for more sparing use of the word "accountability" and, more generally, for more precise and consistent terminology. In particular, destroying the anonymity of the violator of a security policy is more accurately described as "identification" or "exposure" than as "accountability." Consistent and more focused use of the term "accountability" should promote the formation of a coherent research area and the adoption of the technology that it develops.

As discussed elsewhere [FJW11], one challenge in addressing punishment is separating punishment for a violation from other, unrelated events that might occur between the violation and the punishment. Other challenges (especially in implementing systems for accountability) include calibrating the severity of the punishment so that it is an effective deterrent (despite the fact that different participants may view the cost of a particular punishment very differently) and determining how often punishment should be meted out.[1] In addition to these punishment-related issues, our work in Chapter 2 highlights and distinguishes differing approaches to the detection–evidence–judgment–punishment spectrum and to questions of information and action. These different approaches will inform further analysis of accountability, including the study of fundamental tradeoffs related to accountability and the design of new accountability systems.

While we have focused on accountability in Computer Science, the aspects of accountability that we use in our analysis might also be applied to accountability in other disciplines (*e.g.*, the notion of "calling to account" within a particular legal or political framework). The work in Chapter 2 might thus facilitate comparisons and interactions between notions of accountability in different disciplines.

Finally, we remark that the work we have presented in Chapter 2 is about accountability with respect to established policies. Yet, there are forms of online life, including search and social networking, in which expectations, laws, and policies are still developing. Despite the fact that their obligations have not yet been fully formalized and are not yet fully agreed upon, it would be highly desirable to be able to hold the companies that provide search, social networking, and other online services accountable if, at some point in the future, they are seen to have acted egregiously. As work on accountability in computer science continues, this issue should receive more attention.

[1] There are certainly occasions on which punishment might be withheld in order to promote some larger goal, but, if punishment were *always* withheld, the system would not provide accountability.

6.2 Structural Cloud Audits that Protect Private Information

In Chapter 3, we study mechanisms that enable cloud users to hold cloud providers accountable for providing reliable services. We have designed P-SRA (a private, structural-reliability auditor for cloud services based on secure, multi-party computation) and prototyped it using the Sharemind SecreC platform. In addition, we have explored the use of data partitioning and subgraph abstraction in secure, multi-party computations on large graphs, with promising results. We believe that data partitioning and subgraph abstraction can be applied to other cloud-computing or SMPC-related problems. Our preliminary experiments and simulations indicate that P-SRA could be a practical, off-line service, at least for small-scale cloud services or for ones that permit significant subgraph abstraction.

There are many interesting directions for future work, including: (1) Although our preliminary experiments indicate that the cost of privacy in structural reliability auditing (*i.e.*, the additional cost of using P-SRA instead of SRA) is not prohibitive, it would be useful to measure this cost more precisely with more exhaustive experiments. (2) It will be interesting to seek more efficient algorithms for fault-tree analysis and/or a more efficient P-SRA implementation; both would enable us to test P-SRA on larger cloud architectures. (3) Note that we assumed, following Zhai *et al.* [ZWX$^+$], that dependency graphs of cloud services are acyclic, but they need not be. Tunneling-within-tunneling of the type already in use in MPLS and corporate VPNs could (perhaps unintentionally) create cyclic dependencies if used in clouds. Thus, it will be worthwhile to develop structural-reliability auditing techniques that apply to cyclic dependency graphs. (4) P-SRA partitions components based on the fact that some physical equipment is used by exactly one service provider and hence cannot cause the failure of another provider's service, but this type of partitioning has limitations. If, for example, two cloud-service providers purchase large numbers of hard drives of the same make and model from the same batch, and that batch is discovered to be faulty, then the two services have a common dependency on this faulty batch of drives. P-SRA's data partitioning could hide this common dependency, because the hard drives could be considered "private" equipment by both services. It will be worthwhile to extend P-SRA so that it can discover this type of common dependency while retaining the efficiency provided by data partitioning and subgraph abstraction.

6.3 Cloud User Infrastructure Attestation

In Chapter 4, we continue our study of techniques that allow cloud-service users to hold cloud-service providers accountable. Specifically, we develop the notion of cloud user infrastructure attestation. The goal of the work is to enable cloud providers to attest to cloud users that the users have received the resources (VMs and their interconnection) that they requested. Furthermore, the provider should not have to reveal to the users the actual configuration of its servers or network, only their properties; this helps protect the providers' trade secrets and should motivate more providers to adopt such attestation protocols.

Our solution focuses on hardware security anchors, or roots of trust, installed in the individual servers of the cloud provider. We leverage well established TPM technology and also propose a novel component called a *Network TPM*. Both of these hardware components are used to collect information about the cloud infrastructure and attest to properties requested by users. TPMs on the servers where the users VMs run are used to attest to the properties of server infrastructure and to give users assurance about the properties of the servers. Our new proposed Network TPMs installed in these servers are used to attest to the properties of the network infrastructure and to give users assurance about the properties of the interconnection of their VMs. Meanwhile, our attestation protocol uses digitally signed data and verifiable-computation mechanisms to ensure that the attestation is correct and trustworthy. We use ComplexNetworkSim and Pinocchio to implement a prototype of our topology infrastructure attestation framework. Through large-scale simulations, we demonstrate that our cloud user infrastructure attestation framework is a practical offline service.

Directions for future work on cloud user infrastructure attestation include 1) It would be interesting to build a prototype of a network TPMs, along with a trusted software driver. These could be used to measure the performance of a real hardware implementation and to assess extra costs attributable to this special hardware. 2) It would be more convenient for cloud providers to adopt our framework if our attestation protocols were integrated into existing cloud-management software, such as OpenStack. 3) Further work is needed on the design of attestation protocols and on improved efficiency of attestation (with respect to both running time and memory utilization).

6.4 On Virtual-Machine Reallocation in Cloud-scale Data Centers

In Chapter 5, we investigate accountability in the operation of cloud-scale data centers — specifically, we ask what actions should be taken when violations of system policies or other abnormal events are detected. We have formulated the virtual-machine reallocation problem (VMRAP) as an optimization problem, which is NP-hard. We propose a heuristic, two-layer approach and use large-scale simulations to demonstrate that it is efficient and highly accurate. Our solution may be applicable to other aspects of data-center and cloud security.

Interesting directions for future work on VMRAP include 1) As usual, our algorithms would be more likely to be adopted if they were integrated into OpenStack. They could be integrated as an independent module or as a part of the existing VM-scheduling module. 2) In the formulation of the VMRAP, we follow a deterministic but intuitive network model of data-center networks. However, in real data centers, there are many random events that affect the network conditions, such as random failures of links and congestions caused by abrupt arrivals of new tasks. Therefore, it would be interesting to use a probabilistic network model in a formulation the VMRAP, analyze the effects of the randomness, and design random-event-driven solutions to the problem.

Bibliography

[ABF$^+$08] David G. Andersen, Hari Balakrishnan, Nick Feamster, Teemu Koponen, Daekyeong Moon, and Scott Shenker. Accountable internet protocol (AIP). *SIGCOMM Comput. Commun. Rev.*, 38:339–350, August 2008.

[AF12] Dennis Abts and Bob Felderman. A guided tour through data-center networking. *Queue*, 10(5):10, 2012.

[AGHP02] Martín Abadi, Neal Glew, Bill Horne, and Benny Pinkas. Certified email with a light on-line trusted third party: design and implementation. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 387–395, New York, NY, USA, 2002. ACM.

[AL12] Mansoor Alicherry and TV Lakshman. Network aware resource allocation in distributed clouds. In *INFOCOM, 2012 Proceedings IEEE*, pages 963–971. IEEE, 2012.

[Amaa] Amazon web services global infrastructure. `http://aws.amazon.com/en/about-aws/globalinfrastructure/`.

[Amab] Amazon EC2 Instance Types. `http://aws.amazon.com/ec2/instance-types/#instance-details`, accessed September 2013.

[AS98] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of np. *Journal of the ACM (JACM)*, 45(1):70–122, 1998.

[Azu] Windows azure. `http://en.wikipedia.org/wiki/Windows_Azure`.

[BB10] Anton Beloglazov and Rajkumar Buyya. Energy efficient resource management in virtualized cloud data centers. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 826–831, Washington, DC, USA, 2010. IEEE Computer Society.

[BCC04] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 132–145. ACM, 2004.

[BDD$^+$06] Michael Backes, Anupam Datta, Ante Derek, John C. Mitchell, and Mathieu Turuani. Compositional analysis of contract-signing protocols. *Theor. Comput. Sci.*, 367:33–56, November 2006.

[BDMS07] Adam Barth, Anupam Datta, John Mitchell, and Sharada Sundaram. Privacy and utility in business processes. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 279–294, Washington, DC, USA, 2007. IEEE Computer Society.

[BDNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *ACM Symposium on Computer and Communication Security*, pages 257–266, 2008.

[BK13] Dan Bogdanov and Aivo Kalu. Pushing back the rain – how to create trustworthy services in the cloud. *ISACA Journal*, 3:49–51, 2013. Available at `http://www.isaca.org/Journal/Past-Issues/2013/Volume-3/Pages/default.aspx`.

[BLB03] Sonja Buchegger and Jean-Yves Le Boudec. A robust reputation system for mobile ad-hoc networks. Technical report, EPFL, 2003.

[BP06] Giampaolo Bella and Lawrence C. Paulson. Accountability protocols: Formalized and verified. *ACM Trans. Inf. Syst. Secur.*, 9:138–161, May 2006.

[BSMD10] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, 2010.

[BSP$^+$10] Soren Bleikertz, Matthias Schunter, Christian W. Probst, Dimitrios Pendarakis, and Konrad Eriksson. Security audits of multi-tier virtual infrastructures in public infrastructure clouds. In *ACM Cloud Computing Security Workshop*, pages 93–102, 2010.

[But13a] Brandon Butler. Cloud storage viable option, but proceed carefully, 2013. Available at `http://www.networkworld.com/news/2013/010313-gartner-storage-265460.html`.

[But13b] Brandon Butler. Top 10 cloud storage providers, 2013. Available at `http://www.networkworld.com/news/2013/010313-gartner-cloud-storage-265459.html`.

[CFF12] Antonio Corradi, Mario Fanelli, and Luca Foschini. Vm consolidation: a real case based on openstack cloud. *Future Generation Computer Systems*, 2012.

[CFH$^+$05] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[CFN90] D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In *Proceedings on Advances in cryptology*, CRYPTO '88, pages 319–327, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[CGV07] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Performance Evaluation Review*, 35(2):42–51, 2007. `http://www-archive.xenproject.org/files/xensummit_4/3schedulers-xen-summit_Cherkosova.pdf`, accessed September 2013.

[CH04] Hana Chockler and Joseph Y. Halpern. Responsibility and blame: A structural-model approach. *Journal of Artificial Intelligence Research*, 22:93–115, 2004.

[Cha82] David Chaum. Blind signatures for untraceable payments. In *CRYPTO*, pages 199–203, 1982.

[Cha83] David Chaum. Blind signature system. In *CRYPTO*, page 153, 1983.

[CHL06] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Balancing accountability and privacy using e-cash (extended abstract). In *SCN*, pages 141–155, 2006.

[CKV10] Kai-Min Chung, Yael Kalai, and Salil Vadhan. Improved delegation of computation using fully homomorphic encryption. In *Advances in Cryptology–CRYPTO 2010*, pages 483–501. Springer, 2010.

[CLMS08] Liqun Chen, Hans Löhr, Mark Manulis, and Ahmad-Reza Sadeghi. Property-based attestation without a trusted third party. In *Information Security*, pages 31–46. Springer, 2008.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 90–112. ACM, 2012.

[Com] ComplexNetworkSim. `http://pythonhosted.org//ComplexNetworkSim/index.html`. Accessed: 2014-07-28.

[CVX] CVXOPT. Python Software for Convex Optimization. `http://cvxopt.org/`, accessed September 2013.

[Del] Dell R910 Rack Server. `https://www.dell.com/us/business/p/poweredge-r910/pd?~ck=anav`, accessed September 2013.

[DG05] Wenliang Du and Michael T Goodrich. Searching for high-value rare events with uncheatable grid computing. In *Applied Cryptography and Network Security*, pages 122–137. Springer, 2005.

[DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigård, and JesperBuus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography – PKC 2009*, pages 160–179. Springer Verlag, LNCS 5443, 2009.

[DMKC11] Boris Danev, Ramya Jayaram Masti, Ghassan O. Karame, and Srdjan Capkun. Enabling secure vm-vtpm migration in private clouds. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 187–196, New York, NY, USA, 2011. ACM.

[DWG11] Umesh Deshpande, Xiaoshuang Wang, and Kartik Gopalan. Live gang migration of virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 135–146, New York, NY, USA, 2011. ACM.

[EI00] Clifton A. Ericson II. *Hazard analysis techniques for system safety*. John Wiley and Sons, 2000.

[FHJ$^+$11] Joan Feigenbaum, James A. Hendler, Aaron D. Jaggard, Daniel J. Weitzner, and Rebecca N. Wright. Accountability and deterrence in online life (extended abstract). In *ACM Web Science*, 2011. `http://www.websci11.org/fileadmin/websci/Papers/35\_paper.pdf`.

[FJW11] Joan Feigenbaum, Aaron D. Jaggard, and Rebecca N. Wright. Towards a formal model of accountability. In *Proceedings of the 2011 workshop on New security paradigms workshop*, NSPW '11, pages 45–56, New York, NY, USA, 2011. ACM.

[FZML02] Csilla Farkas, Gábor Ziegler, Attila Meretei, and András Lörincz. Anonymity and accountability in self-organizing electronic communities. In *Proceedings of the 2002 ACM workshop on Privacy in the Electronic Society*, WPES '02, pages 81–90, New York, NY, USA, 2002. ACM.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Advances in Cryptology–CRYPTO 2010*, pages 465–482. Springer, 2010.

[GHJ$^+$09] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[GK05] Ruth W. Grant and Robert O. Keohane. Accountability and abuses of power in world politics. *American Political Science Review*, 99(01):29–43, 2005.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for muggles. In *Proceedings of the annual ACM Symposium on Theory of Computing*, pages 113–122. ACM, 2008.

[GLP] GLPK (GNU Linear Programming Kit). `https://www.gnu.org/software/glpk/`, accessed September 2013.

[GM01] Philippe Golle and Ilya Mironov. Uncheatable distributed computations. In *Topics in Cryptology, CT-RSA 2001*, pages 425–440. Springer, 2001.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[GPC$^+$03] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 193–206. ACM, 2003.

[GR$^+$03] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, 2003.

[Gri] Openstack grizzly architecture. `http://www.solinea.com/2013/06/15/openstack-grizzly-architecture-revisited/`, accessed September 2013.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Advances in Cryptology-ASIACRYPT 2010*, pages 321–340. Springer, 2010.

[GSP$^+$12] Debayan Gupta, Aaron Segal, Aurojit Panda, Gil Segev, Michael Schapira, Joan Feigenbaum, Jennifer Rexford, and Scott Shenker. A New Approach to Interdomain Routing Based on Secure Multi-Party Computation. In *ACM SIGCOMM Workshop on Hot Topics in Networks*, 2012.

[Gur] Gurobi Optimizer, State of the Art Mathematical Programming Solver. `http://www.gurobi.com/products/gurobi-optimizer/gurobi-overview`, accessed September 2013.

[Hae10] Andreas Haeberlen. A case for the accountable cloud. *ACM SIGOPS Operating Systems Review*, 44(2):52–57, 2010.

[HARD10] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *OSDI*, pages 119–134, 2010.

[HG09] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 51–60, New York, NY, USA, 2009. ACM.

[HG11] Ryan Henry and Ian Goldberg. Formalizing anonymous blacklisting systems. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 81–95, Washington, DC, USA, 2011. IEEE Computer Society.

[HJ04] Jacob Gorm Hansen and Eric Jul. Self-migration of operating systems. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, EW 11, New York, NY, USA, 2004. ACM.

[HKD07] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: practical accountability for distributed systems. *SIGOPS Oper. Syst. Rev.*, 41:175–188, October 2007.

[HKS$^+$10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: tool for automating secure two-party computations. In *ACM Conference on Computer and Communications Security*, pages 451–462, 2010.

[HP05] Joseph Y. Halpern and Judea Pearl. Causes and explanations: A structural-model approach–part I: Causes. *British J. Philos. Sci.*, 56:843–887, 2005.

[JJPR09] Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. Towards a theory of accountability and audit. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, pages 152–167, Berlin, Heidelberg, 2009. Springer-Verlag.

[JLH$^+$12] Joe Wenjie Jiang, Tian Lan, Sangtae Ha, Minghua Chen, and Mung Chiang. Joint vm placement and routing for data center traffic engineering. In *INFOCOM, 2012 Proceedings IEEE*, pages 2876–2880. IEEE, 2012.

[Kai96] Rajashekar Kailar. Accountability in electronic commerce protocols. *IEEE Trans. Softw. Eng.*, 22:313–328, May 1996.

[Kar72] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[KDSR11] Eric Keller, Dmitry Drutskoy, Jakub Szefer, and Jennifer Rexford. Cloud resident data center, 2011. Technical report, Princeton University.

[Kel60] James E Kelley, Jr. The cutting-plane method for solving convex programs. *Journal of the Society for Industrial & Applied Mathematics*, 8(4):703–712, 1960.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the annual ACM Symposium on Theory of Computing*, pages 723–732. ACM, 1992.

[KJM$^+$11] Ryan KL Ko, Peter Jagadpramana, Miranda Mowbray, Siani Pearson, Markus Kirchberg, Qianhui Liang, and Bu Sung Lee. Trustcloud: A framework for accountability and trust in cloud computing. In *Proceedings of the IEEE World Congress on Services*, pages 584–588. IEEE, 2011.

[KTV10] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Accountability: definition and relationship to verifiability. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 526–535, New York, NY, USA, 2010. ACM.

[Lam09] Butler Lampson. Privacy and security: Usable security: how to get it. *Commun. ACM*, 52:25–27, November 2009.

[Lei85] C.E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, Oct 1985.

[LSL$^+$11] Dave Levin, Aaron Schulman, Katrina Lacurts, Neil Spring, and Bobby Bhattacharjee. Making currency inexpensive with iOwe. In *Proceedings of NetEcon'11*, 2011.

[MDP$^+$00] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, September 2000.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay – a secure two-party computation system. In *USENIX Security Symposium*, pages 298–302, 2004.

[MOR01] Silvio Micali, Kazuo Ohta, and Leonid Reyzin. Accountable-subgroup multisignatures: extended abstract. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, CCS '01, pages 245–254, New York, NY, USA, 2001. ACM.

[MPZ10] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[MS70] R. A. Meyer and L. H. Seawright. A virtual machine time-sharing system. *IBM Systems Journal*, 9(3):199–218, 1970.

[MSY12] Siva Theja Maguluri, R Srikant, and Lei Ying. Stochastic models of load balancing and scheduling in cloud computing clusters. In *INFOCOM, 2012 Proceedings IEEE*, pages 702–710. IEEE, 2012.

[Mul00] Richard Mulgan. 'Accountability': An ever-expanding concept? *Public Administration*, 78(3):555–573, 2000.

[MWR99] Fabian Monrose, Peter Wyckoff, and Aviel D Rubin. Distributed execution with remote audit. In *NDSS*, volume 99, pages 3–5, 1999.

[Nak] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`.

[net] NetworkX. `http://networkx.github.com/`.

[Nis97] Helen Nissenbaum. Accountability in a computerized society. In Batya Friedman, editor, *Human values and the design of computer technology*, pages 41–64. Center for the Study of Language and Information, Stanford, CA, USA, 1997.

[NLH05] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[NMPF$^+$09] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 39–50, New York, NY, USA, 2009. ACM.

[NW88] George L Nemhauser and Laurence A Wolsey. *Integer and combinatorial optimization*, volume 18. Wiley New York, 1988.

[Ore12] Will Oremus. Internet outages highlight problem for cloud computing: Actual clouds, 2012. Available at `http://www.slate.com/blogs/future_tense/2012/07/02/amazon_ec2_outage_netflix_pinterest_instagram_down_after_aws_cloud_loses_power.html`.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the EEE Symposium on Security and Privacy*, pages 238–252. IEEE, 2013.

[PSvD06] Ronald Perez, Reiner Sailer, and Leendert van Doorn. vtpm: virtualizing the trusted platform module. In *Proceedings of the 15th Conference on USENIX Security Symposium*, pages 305–320, 2006.

[PyC] PyCrypto. `https://www.dlitz.net/software/pycrypto/`. Accessed: 2014-07-28.

[RS97] Ronald L. Rivest and Adi Shamir. Payword and micromint: Two simple micropayment schemes. In *Proceedings of the International Workshop on Security Protocols*, pages 69–87, London, UK, 1997. Springer-Verlag.

[SBMS07] Mehul A. Shah, Mary Baker, Jeffrey C. Mogul, and Ram Swaminathan. Auditing to keep online storage services honest. In *USENIX Workshop on Hot Topics in Operating Systems*, 2007.

[SGR09] Nuno Santos, Krishna P Gummadi, and Rodrigo Rodrigues. Towards trusted cloud computing. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, pages 3–3, 2009.

[Sim] SimPy. `https://simpy.readthedocs.org/en/latest/`. Accessed: 2014-07-28.

[SJCL12] Jakub Szefer, Pramod Jamkhedkar, Yu-Yuan Chen, and Ruby B. Lee. Physical Attack Protection with Human-Secure Virtualization in Data Centers. In *Workshop on Open Resilient human-aware Cyber-physical Systems*, WORCS, pages 1–6, June 2012.

[SJV$^+$05] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramon Caceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a mac-based security architecture for the xen open-source hypervisor. In *Computer Security Applications Conference*, pages 10–pp. IEEE, 2005.

[Sma] Hard disk smart drives. `http://www.pctechguide.com/hard-disks/hard-disk-smart-drives`, accessed September 2013.

[SMBW12] Srinath TV Setty, Richard McPherson, Andrew J Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, 2012.

[SRGS12] Nuno Santos, Rodrigo Rodrigues, Krishna P Gummadi, and Stefan Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Usenix Security*, 2012.

[SS04] Ahmad-Reza Sadeghi and Christian 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.

[SSB08] Mehul A. Shah, Ram Swaminathan, and Mary Baker. Privacy-preserving audit and extraction of digital contents. Cryptology ePrint Archive, Report 2008/186, 2008. Available at `http://eprint.iacr.org/2008/186/`.

[SVP$^+$12] Srinath TV Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security Symposium*, pages 253–268, 2012.

[SZL$^+$11] Vivek Shrivastava, Petros Zerfos, Kang-Won Lee, Hani Jamjoom, Yew-Huey Liu, and Suman Banerjee. Application-aware virtual machine migration in data centers. In *INFOCOM, 2011 Proceedings IEEE*, pages 66–70. IEEE, 2011.

[TAKS08] Patrick P. Tsang, Man Ho Au, Apu Kapadia, and Sean W. Smith. PEREA: towards practical TTP-free revocation in anonymous authentication. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 333–344, New York, NY, USA, 2008. ACM.

[TCG] Trusted Computing Group. `http://www.trustedcomputinggroup.org/`. Accessed: 2014-02-07.

[TRMP12] Justin Thaler, Mike Roberts, Michael Mitzenmacher, and Hanspeter Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud Workshop*, 2012.

[VGRH81] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. US Nuclear Regulatory Commission, 1981.

[WABL$^+$08] Daniel J. Weitzner, Harold Abelson, Tim Berners-Lee, Joan Feigenbaum, James Hendler, and Gerald Jay Sussman. Information accountability. *Commun. ACM*, 51:82–87, June 2008.

[WCW$^+$13] Cong Wang, Sherman S. M. Chow, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for secure cloud storage. *IEEE Transactions on Computers*, 62(2):362–375, 2013.

[WRLL10] Cong Wang, Kui Ren, Wenjing Lou, and Jin Li. Toward publicly auditable secure cloud data storage services. *IEEE Network*, 24(4):19–24, 2010.

[WSVY07] Timothy Wood, Prashant J Shenoy, Arun Venkataramani, and Mazin S Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, volume 7, pages 229–242, 2007.

[WWR$^+$11] Qian Wang, Cong Wang, Kui Ren, Wenjing Lou, and Jin Li. Enabling public auditability and data dynamics for storage security in cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(5):847–859, 2011.

[WWRL10] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *IEEE INFOCOM*, pages 525–533, 2010.

[XFF13] Hongda Xiao, Bryan Ford, and Joan Feigenbaum. Structural cloud audits that protect private information. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop*, pages 101–112. ACM, 2013.

[Yao82] Andrew C. Yao. Protocols for secure computation. In *IEEE Symposium on Foundations of Computer Science*, pages 160–164, 1982.

[Yao86] Andrew C. Yao. How to generate and exchange secrets. In *IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.

[YC04] Aydan R. Yumerefendi and Jeffrey S. Chase. Trust but verify: accountability for network services. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, EW 11, New York, NY, USA, 2004. ACM.

[YC05] Aydan R. Yumerefendi and Jeffrey S. Chase. The role of accountability in dependable distributed systems. In *Proceedings of the First conference on Hot topics in system dependability*, HotDep'05, pages 3–3, Berkeley, CA, USA, 2005. USENIX Association.

[YC07] Aydan R. Yumerefendi and Jeffrey S. Chase. Strong accountability for network storage. *Trans. Storage*, 3, October 2007.

[YJ12] Kan Yang and Xiaohua Jia. Data storage auditing service in cloud computing: challenges, methods and opportunities. *World Wide Web*, 15(4):409–428, 2012.

[ZCRY03] Sheng Zhong, Jiang Chen, and Yang Richard Richard Yang. Sprite: A simple, cheat-proof, credit-based system for mobile ad-hoc networks. In *INFOCOM*, 2003.

[ZCWF13] Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. An untold story of redundant clouds: Making your service deployment truly reliable. In *ACM Workshop on Hot Topics in Dependable Systems*, 2013.

[ZG96] Jianying Zhou and D. Gollman. A fair non-repudiation protocol. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, SP '96, pages 55–, Washington, DC, USA, 1996. IEEE Computer Society.

[ZJRR12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM.

[ZWX$^+$] Ennan Zhai, David Isaac Wolinsky, Hongda Xiao, Hongqiang Liu, Xueyuan Su, and Bryan Ford. Auditing the structural reliability of the clouds. Technical Report YALEU/DCS/TR-1479, July 2013. Available at `http://www.cs.yale.edu/publications/techreports/tr1479.pdf`.