Università degli Studi "Roma Tre"

Dipartimento di Matematica e Fisica

Scuola Dottorale in Scienze Matematiche e Fisiche

Sezione di Matematica

XXVI Ciclo

Ph.D. Thesis

Secure Distributed Computing on a Manycore Cloud

Candidate:

Flavio Lombardi

Advisor:

Dr. Roberto Di Pietro

Coordinator:

Prof. Luigi Chierchia

A.A. 2013-2014

Abstract

Computation outsourcing is an increasingly successful paradigm today. Private and public organizations, as well as common users, can access a large number of economically viable resources to perform the desired computations or access data. The cloud approach allows outsourcers to offer on-demand scalable services to third parties or to perform large computations without high server farm maintenance costs. Scientific computing is one of the beneficiaries of such novel scenarios. Stemming from the first in-house built clusters, distributed computing has flourished over the years also thanks to the Internet. Grids before, and later clouds, have enhanced the storage and computation potentiality. At the same time, to overcome Moore's law real-world limitations, multicore and manycore hardware has emerged thus allowing parallel computing on a chip. Again, scientific computing in particular has benefited from the speed-up of such novel technology, especially when combined with outsourced cloud computing. As a further opportunity, increasingly powerful mobile devices are now widespread and feature multiple execution cores. They are pervasively immersed in the cloud and can host a large number of applications. These handheld devices can provide additional data acquisition interfaces and pervasive computational resources for heterogeneous workloads.

In such a novel scenario it is of primary importance to provide adequate security and privacy guarantees to both providers and users of outsourced services. The problem is complex, as present cloud landscape actors (cloud providers, service providers, and service users) have different requirements and objectives. The first category aims at offering cost-efficient computation while avoiding misuse, the second category aims at reducing costs while at the same time maximizing revenue, the last category aims at reducing the expenses for the accessed services while being guaranteed that the computation is secure, reliable and correct. As a common threat to all these categories lies malicious software, increasingly smart and stealth, affecting a wide range of devices from servers to desktops and even mobile nodes. Such malware can access distributed resources, alter computation outcome and data, leak information. As such, it is of paramount importance to prevent, detect and react to such threats in the smartest possible way.

This thesis addresses the security issues involved with computation outsourcing over distributed heterogeneous cloud nodes. It investigates different directions and proposes possible approaches to their solutions in a variety of scenarios. We expect the approaches and achievements presented here to pave the way to further research in the field.

Acknowledgements

Writing a Ph.D. Thesis is a fashinating and complex task that requires dedication but also great support from friends and colleagues. I feel the need to thank many people, who were close to me both in scientific and personal matters. I am glad to take this opportunity to mention, in the following unordered list, all those people I owe a lot.

First of all, I would like to thank my Thesis advisor Dr. Roberto Di Pietro and his patient wife "vi abbiamo nel cuore" Chiara. With Roberto, research is a pervasive overly-interesting wonderful collaborative and stimulating effort performed 24h/day 365day/year. Roberto gave me the chance to take part to the young and dynamic Springer Research Group at Roma Tre. As my thesis advisor, Roberto enjoyed with me the issues of information security and gave me the motivation to start a Ph.D. programme.

A special thank and a kiss goes once again to my lovely lifemate Sonia. Special thanks and kisses also go to my father Antonio, my mother Paola, my grandmother Giovanna, my brother Fabio, my two wonderful nephews Arianna and Federico and "mi cugna' " Stefania.

I would also like to thank all the co-authours of my published, submitted, and ongoing work and the director of the Ph.D. programme prof. Chierchia. In particular, I would like to thank Arturo, Benjamin, Guillermo, Juan, Jose, Javi, Sergio, Jorge, Jose, Lorena, Anabel, Agustin and in general all the people of COSEC Lab at University Carlos III de Madrid for their hospitality and fruitful collaboration and support. I would like to add that I was pleased and impressed after working with Agusti and Pau from Tarragona.

Last, but not least, I would like to mention those who shared these sweet (see below) Ph.D. years with me, among others Stefano "keep calm" Guarino, Giulio "daie!" Meleleo, Cihan "pysmanye" Pehlivan, Antonio "the Boss" Cigliola, Fabio "Mozart" Felici, Giulio "helpful" Aliberti, Antonio Villani, Daniele Piras, Lorenzo "bombo" Menici, Elena "doc" Pulvirenti, Giovanni "Mr.ciccioli" Mongardi, and many many more...


ROMA
TRE

Contents

List of Figures

List of Tables

1 Introduction

The availability of a large number of computing and storage facilities for computationally-intensive tasks indicates the trend towards outsourced and distributed services. Larger and larger amounts of data are being stored, exchanged and processed by a large number of heterogeneous nodes. In fact, the amount of computing performed by both private companies and public organizations is rapidly increasing. As an example, Scientific Computing at large can take advantage of the large computing and storage capability made available by the outsourced computing paradigm.

In particular, cloud computing is increasingly successful and related technologies are rapidly evolving. As a matter of fact, the use of cloud services reduces maintenance costs and increases performance and service availability, but may introduce security and privacy concerns as computing nodes may misbehave or fail due to a number of reasons.

Security concerns over cloud computing nodes are often due to virtual machine integrity and privacy issues (possibly due to malware infections). In addition, nodes can deliberately choose to misbehave in order to save resources and thus reduce maintenance costs. As a consequence, availability and correctness of stored data and computed results from remote services can be an issue.

Furthermore, given that cloud nodes are heterogeneous and span from manycore cluster nodes (hosting powerful manycore GPUs) to mobile devices (also featuring manycore GPUs), care must be taken in evaluating and addressing specific security issues of each platform while at the

same time keeping in mind that these pervasively-available and constantly-connected resources have a common substrate and common global security issues.

To ensure an adequate level of availability and security, traditional approaches consist in introducing redundancy in the computation over multiple distributed nodes, thus increasing service reliability and cheating detection. Such traditional solutions are not very efficient and have the disadvantage of reducing cloud outsourcing convenience. This introduces the need to develop new models and methods for the definition and enforcement of reliable outsourced computation while ensuring efficiency and cost-effectiveness.

Motivation

The need to solve increasingly more complex problems, such as those requiring to build and simulate large mathematical and scientific models, calls for a large number of powerful on-demand computing resources. A solution to the low performance provided by a single processing unit (i.e., a single processor) was introduced more than 30 years ago, and goes under the name of "parallel and distributed computing". Furthermore, as demonstrated by a large number of attempts such as (among others) Seti@home, MapReduce and cloud computing offerings, the computation can be outsourced and distributed over a large number of (possibly heterogeneous) nodes.

The main idea is to leverage the potentiality offered by the simultaneous processing of arbitrarily many locally available cores. Anyway, a further development is the use of distributed heterogeneous computing processors, interconnected by fast high-bandwidth networks. Such a system presents advantages, such as economy and ease of deployment of additional (on-demand) computing resources. However, it suffers from a number of problems, in particular related to reliability, availability, privacy and security in general.

Distributing the workload can scale up in the small, i.e. leveraging the multicore and manycore processing units available in most present servers, desktops and even mobile platforms. In addition, the workload can also scale up over the network, thus leveraging cluster, Hadoop and cloud nodes to spread data and computation over distributed computing resources. Both approaches are often used at the same time, allowing an unprecedented amount of work to be performed in a short time.

In particular, GPGPU computing i.e., the possibility of leveraging the large number (+1000s) of simple fast computing cores of Graphics Processing Units for general purpose computations, is an increasingly successful relevant new area of research. GPGPU computing introduces novel computing paradigms as well as important security concerns.

Allowing secure transparent usage of both locally-parallel and large-scale distributed cloud computing resources is the objective of present work. The possibility of leveraging smart monitoring and of analyzing the behavior and the outcomes of local and remote resources enables a new level of security that allows to transparently and confidently use such resources. Once reliability, availability, integrity and privacy guarantees will be given over both cloud data and computation, the cloud paradigm will finally emerge as one of the most relevant approaches to computing of the last decade.

Main contributions of this thesis

In this thesis, we present a comprehensive approach for monitoring and protecting computations hosted on heterogeneous computing nodes out of the client control. There are mainly three security aspects that need to be considered when ensuring reliable secure computation over remote heterogeneous cloud nodes.

The first aspect is given by the integrity enforcement of cloud nodes (in particular of VMs). The problem here is to guarantee that virtual machines behave as expected and do not perform malicious activity. An advanced comprehensive execution monitoring system has been introduced and evaluated in [1]. Further, a novel effective real-time execution control approach based on Execution Path Analysis has been produced in [2]. Such a system allows detecting and reacting to anomalous code behavior before it can produce damage to the cloud. In addition, a differential analysis approach based on tainting [3] has been devised that helps evaluating applications in order to asses their behavior and to help classify them as goodware, greyware or malware.

The second aspect is computation reliability aimed at limiting the probability of erroneous or fake data or computed results. With respect to this relevant problem, we have introduced effective task distribution [4] and result evaluation and node-ranking approaches [5] that minimize the chances that cheating nodes can affect the global computation outcome. In particular [5] won the Best Paper Award at the 10th IEEE International Conference on Autonomic and

Trusted Computing (ATC). Furthermore, we have investigated and discussed interesting results on time/action cheating that can help forensic investigations in the cloud [6].

The third aspect is related to cloud privacy issues. In particular we have shown that present manycore cloud computing is inherently insecure and we have proposed both new execution models and practical remedies [7] that prevent one user from accessing or affecting data and computation of another user on a multitenant GPU cloud.

Organization of present work

In this first section we have introduced the main problems and the motivation of our work. In addition, we have summarized the main contributions of this Thesis. The remaining chapters are organized as follows:

Chapter 2 Surveys cloud security issues and suggests integrity-guaranteeing approaches.

Chapter 3 Introduces execution monitoring and proactive remedies to malware and misconfiguration in cloud nodes/VMs.

Chapter 4 Introduces and discusses resilience approaches to misbehaving cloud nodes.

Chapter 5 Depicts novel multitenant service sharing issues over GPU clouds.

Chapter 6 Finally draws conclusions and highlights future work directions.

List of contributions

Works accepted for publication during the Ph.D. and relevant to the topic of present thesis:

1. R. Di Pietro, F. Lombardi. Secure Virtualization for Cloud Computing. Elsevier Journal of Network and Computer Applications (2011). ISSN 1084-8045 [1].

2. R. Di Pietro, F. Lombardi, and M. Signorini. CloRexPa: Cloud resilience via execution path analysis. Elsevier Future Generation Computer Systems (2014). ISSN 0167-739X [2].

3. R. Di Pietro, F. Lombardi, F. Martinelli and D. Sgandurra. CheR: Cheating Resilience in the Cloud via Smart Resource Allocation. 6th Intl. Symposium on Foundations & Practice of Security (FPS 2013) [4].

4. R. Di Pietro, F. Lombardi, F. Martinelli and D. Sgandurra. AntiCheetah: an Autonomic Multi-round Approach for Reliable Computing. 10th IEEE International Conference on Autonomic and Trusted Computing (ATC 2013 **Best Paper Award**) [5].

Other accepted works related but less relevant with respect to the topic of present thesis:

1. G. Suarez de Tangil, F. Lombardi, J. E. Tapiador and R. Di Pietro. Thwarting Obfuscated Malware via Differential Fault Analysis. IEEE Computer Magazine. ISSN 0018-9162 [3].

2. F. Lombardi, R. Spigler. The Evolution of the approach to Scientific Computing: a Survey. Parallel & Cloud Computing ISSN: 2304-9456 [8].

3. F. Lombardi and R. Di Pietro. (Book Chapter) Title: "Towards a GPU Cloud: Benefits and Security Issues" Book title: "Continued Rise of the Cloud: Advances and Trends in Cloud Computing". ISBN 978-1-4471-6451-7 [9].

4. F. Lombardi and R. Di Pietro. (Book Chapter) Title: "Virtualization and Cloud Security: Benefits, Caveats and Future Developments" Book title: "Cloud Computing: Challenges, Limitations and R&D Solutions" [10].

2 Secure Virtualization and Cloud Computing

Cloud computing adoption and diffusion are threatened by unresolved security issues that affect both the cloud provider and the cloud user. In this chapter [1], we show how virtualization can increase the security of cloud computing, by protecting both the integrity of guest virtual machines and the cloud infrastructure components. In particular, we propose a novel architecture, Advanced Cloud Protection System (ACPS), aimed at guaranteeing increased security to cloud resources. ACPS can be deployed on several cloud solutions and can effectively monitor the integrity of guest and infrastructure components while remaining fully transparent to virtual machines and to cloud users. ACPS can locally react to security breaches as well as notify a further security management layer of such events. A prototype of our ACPS proposal is fully implemented on two current open source solutions: Eucalyptus and OpenECP. The prototype is tested against effectiveness and performance. In particular: (a) effectiveness is shown testing our prototype against attacks known in the literature; (b) performance evaluation of the ACPS prototype is carried out under different types of workload. Results show that our proposal is resilient against attacks and that the introduced overhead is small when compared to the provided features.

[1] Part of this chapter appeared in [1]

2.1 Introduction

Internet is on the edge of another revolution, where resources are globally networked and can be easily shared. *Cloud computing* is the main component of this paradigm, that renders the Internet a large repository where resources are available to everyone as services. In particular, cloud nodes are increasingly popular even though unresolved security and privacy issues are slowing down their adoption and success. Indeed, integrity, confidentiality, and availability concerns are still open problems that call for effective and efficient solutions. Cloud nodes are inherently more vulnerable to cyber attacks than traditional solutions, given their size and underlying service-related complexity—that brings an unprecedented exposure to third parties of services and interfaces. In fact, the cloud "is" the Internet, with all the pros and cons of this pervasive system. As a consequence, increased protection of cloud internetworked nodes is a challenging task. It becomes then crucial to recognize the possible threats and to establish security processes to protect services and hosting platforms from attacks.

Cloud Computing already leverages virtualization for load balancing via dynamic provisioning and migration of virtual machines (VM or *guest* in the following) among physical nodes. VMs on the Internet are exposed to many kinds of interactions that virtualization technology can help filtering while assuring a higher degree of security. In particular, virtualization can also be used as a security component; for instance, to provide monitoring of VMs, allowing easier management of the security of complex cluster, server farms, and cloud computing infrastructures to cite a few. However, virtualization technologies also create new potential concerns with respect to security, as we will see in Section 2.4.

2.1.1 Contributions

The goal of this chapter is twofold: (a) to investigate the security issues of cloud computing; (b) to provide a solution to the above issues.

We analyzed cloud security issues and model, examined threats and identified the main requirements of a protection system. In particular, we developed an architecture framework, Advanced Cloud Protection System (ACPS), to increase the security of cloud nodes. ACPS is a complete protection system for clouds that transparently monitors cloud components and interacts with local and remote parties to protect and to recover from attacks.

In the following we show how ACPS can leverage full virtualization to provide increased protection to actually deployed cloud systems such as Eucalyptus [11] and OpenECP [12] (also referred to as Enomalism [13] in the following). In fact, OpenECP is a fully open source code fork of the previously open source Enomalism offer; as such, it shares the same architecture and codebase. A prototype implementation is presented. Its effectiveness and performance are tested. Results indicate that our proposal is resilient against attacks and that the introduced overhead is small—especially when compared to the features provided.
One main outcome of our research is a framework that allows virtualization-supported cloud protection across physical hosts over the Internet.

2.1.2 Roadmap

The remainder of this chapter is organized as follows: next section surveys related work. Section 2.3 provides background information, while Section 2.4 classifies cloud security issues. Section 2.5 describes ACPS requirements and architecture. In Section 2.6 implementation details are provided, while effectiveness and performance are discussed in Section 2.7. Finally, Section 2.8 draws some conclusions.

2.2 Related Work

While privacy issues in clouds have been described in depth by Pearson [14], cloud security is less discussed in the literature [15]. Some interesting security issues are discussed in [16], while an almost complete survey of security in the context of cloud storage services is provided by Cachin [17]. An exhaustive cloud security risk assessment has been recently presented by ENISA [18]. Also worth reading is the survey on cloud computing presented in [19]. These papers have been the starting points of our work and we refer to them in terms of problems and terms definition.

A fundamental reference for our research is the work on co-location [20] by Ristenpart. This work shows that it is possible to instantiate an increasing number of guest VMs until one is placed co-resident with the target VM. Once successfully achieved co-residence, attacks can theoretically extract information from a target VM on the same machine. An attacker might also actively trigger new victim instances exploiting cloud auto-scaling systems. Ristenpart shows

that it practical to hire additional VMs whose launch can produce a high chance of co-residence with the target VM. He also shows that determining co-residence is quite simple.

Most current integrity monitoring and intrusion detection solutions can be successfully applied to cloud computing. Filesystem Integrity Tools and Intrusion Detection Systems such as *Tripwire* [21] and (*AIDE*)[22] can also be deployed in virtual machines, but are exposed to attacks possibly coming from a malicious guest machine user. Furthermore, when an attacker detects that the target machine is in a virtual environment, it may attempt to break out of the virtual environment through vulnerabilities (very rare at the time of writing [23]) in the Virtual Machine Monitor (VMM). Most approaches leverage VMM isolation properties to secure VMs by leveraging various levels of virtual introspection. Virtual introspection [24] is a process that allows to observe the state of a VM from the VMM. *SecVisor* [25] *Lares* [26] and *KVM-L4* [27], to name a few, leverage virtualization to observe and monitor guest kernel code integrity from a privileged VM or from the VMM. *Nickle* [28] aims at detecting kernel rootkits by monitoring the integrity of kernel code. However, *Nickle* does not protect against kernel data attacks [29], whereas our solution does.

In an effort to make nodes resilient against long-lasting attacks, *Self-Cleansing Intrusion Tolerance* (SCIT) [30] treats all servers as potentially compromised (since undetected attacks are extremely dangerous over time). SCIT restores servers from secure images on a regular basis. The drawback of such a system is that it does not support long-lasting sessions required by most cloud applications. Similarly, *VM-FIT* [31] creates redundant server copies which can periodically be refreshed to increase the resilience of the server. Finally, Sousa's [32] approach combines proactive recovery with services that allow correct replicas to react and be recovered when there is a sufficient probability that they have been compromised. Along with the many advantages brought by virtualization, there are additional technological challenges that virtualization presents, which include an increase in the complexity of digital forensics [33] investigations as well as questions regarding the forensics boundaries of a system.

ACPS enjoys unique features, such as the SWADR approach, the increased decoupling of action and reaction, the increased immunity and integrity of the platform—as well as the integration with real-world architecture—and the support for accountability.

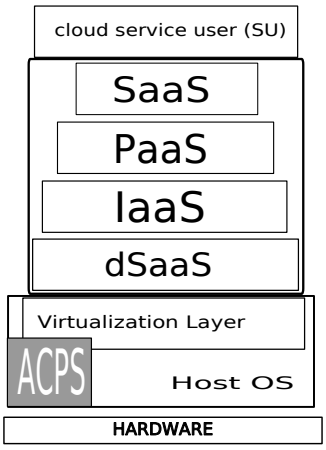

FIGURE 2.1: Cloud layers and the Advanced Cloud Protection System.

2.3 Background

A cloud [34] is a pool of virtualized resources across the Internet that follows a pay-per-use model and can be dynamically reconfigured to satisfy user requests via on-the-fly provisioning/deprovisioning of virtual machines. Cloud computing is a service model for IT provisioning, often based on virtualization and distributed computing technologies. Within the cloud paradigm, concepts such as virtualization, distributed computing and utility computing are applied [35]. Cloud computing approach to distributed computing shares many ideas with grid computing, but these two differ in target, in focus, and in the implementation technologies [36]. On the one hand, with respect to a grid, the cloud user has less control over the location of data and computation. On the other hand, cloud computing management costs are usually much lower and management is less cumbersome. In the following we will also refer to the cloud infrastructure components as middleware.

Cloud services are available at different layers (see the *-as-a-Service or *aaS layers in Figure 2.1): **dSaaS** The data Storage as a Service delivering basic storage capability over the network; **IaaS** The Infrastructure as a Service layer providing bare virtual hardware with no software stack; **PaaS** The Platform as a Service layer providing a virtualized servers, OS, and applications; **SaaS** The Software as a Service layer providing access to software over the Internet as a service.

In this work, efforts have been focused on the "lowest" computational layer (i.e. **IaaS**) since we can more effectively provide a security foundation on top of which more secure services can be offered. Most existing cloud computing systems are proprietary (even though APIs are

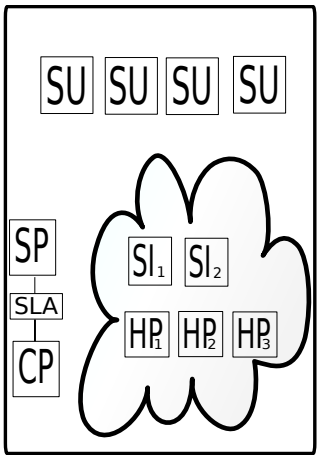

FIGURE 2.2: Cloud service model components: Cloud Provider (CP), Hosting Platform (HP), Service Level Agreement (SLA), Service Provider (SP), Service Instance (SI), Service User (SU).

open and well-known) and as such do not allow modifications, enhancements or integration with other systems for research purposes. This is the reason why we have chosen Eucalyptus and OpenECP, both open source cloud implementations, for integration with our architecture. In the following, even though we will focus on the security issues of those two platforms, most considerations will be general enough to be valid for other platforms as well.

2.4 Cloud Security Issues

One of the key issues of cloud computing (see Figure 2.1) is loss of control. As a first example, the service user (SU) does not know where exactly its data is stored and processed in the cloud. Computation and data are mobile and can be migrated to systems the SU cannot directly control. Over the Internet, data is free to cross international borders and this can expose to further security threats. A second example of loss of control is that the cloud provider (CP) gets paid for running a service he does not know the details of. This is the dark side of the "Infrastructure as a Service" model, but also of other "as a Service" approaches. To date, misuse problems tend to be regulated by a service contract, where such an agreement should be enforced and controlled by monitoring tools [37].

Some of the security issues of a cloud are [36]:

SEI1 Privileged user access: access to sensitive outsourced data has to be limited to a subset of privileged users (to mitigate the risk of abuse of high privilege roles);

SEI2 Data segregation: one instance of customer data has to be fully segregated from other customer data;

SEI3 Privacy: exposure of sensitive information stored on the platforms implies legal liability and loss of reputation;

SEI4 Bug Exploitation: an attacker can exploit a software bug to steal valuable data or to take over resources and allow for further attacks;

SEI5 Recovery: the cloud provider has to provide an efficient replication and recovery mechanism to restore services, should a disaster occur;

SEI6 Accountability: even though cloud services are difficult to trace for accountability purposes, in some cases this is a mandatory application requirement.

With respect to the latter point, accountability can increase security and reduce risks for both the service user and the service provider. A trade-off between privacy and accountability exists, since the latter produces a record of actions that can be examined by a third party when something goes wrong. Such an investigation might show faulty components or internal cloud resource configuration details. This way, a cloud customer might be able to learn information about the internal structure of the cloud that could be used to perform an attack. A possible solution could be the use of obfuscation and privacy-preserving techniques to limit the information the VM exposes to the cloud [38]. Anyway, current technology cannot prevent a VMM from accessing guest raw memory. This leaves open confidentiality issues with respect to the service provider (or with respect to an attacker if he compromises the hosting platform).

2.4.1 Cloud Security Model

Figure 2.2 illustrates the scenario we are concerned with in this chapter. A service provider (SP) runs one or more service instances (SI) on the cloud, which can be accessed by a group of final service users (SU). For this purpose, the SP hires resources from the cloud provider (CP). It is worth noticing that the SU and the SP do not have any physical control over cloud machines, whose status cannot be observed. The SU and the CP enter into a Service Level Agreement that describes how the cloud is going to run service SI.

Possible attacks against cloud systems can be classified as follows (see also [39]):

CAT1 Resource attacks against CPs;

CAT2 Resource attacks against SPs;

CAT3 Data attacks against CPs;

CAT4 Data attacks against SPs;

CAT5 Data attacks against SUs.

Resource attacks (CAT1-CAT2) regard the misuse of resources, such as stealing virtual resources to mount a large scale botnet attack. Data attacks (CAT3-CAT4) steal or modify service or node configuration data (that can be used later to perform an attack). Data attacks against service users (CAT5) can lead to leakage of sensitive data. CAT1 and CAT3 attack classes involve an attack to cloud infrastructure components. Virtualization technologies underlying cloud computing infrastructure can pose security challenges themselves [23]. In addition, cloud computing middleware potentially allows some specific attacks that have not been identified yet. We will later see how ACPS deals with such threats.

2.5 Advanced Cloud Protection System

The proposed Advanced Cloud Protection System (ACPS) is intended to actively protect the integrity of the guest VMs and of the distributed computing middleware by allowing the host to monitor guest virtual machines and infrastructure components.

ACPS is a purely host side architecture leveraging virtual introspection [40]. This allows: to deploy any guest virtual appliance "as it is"; to enforce some form of accountability on guest activity without being noticed by an attacker located on the guest. This latter feature is provided being the protection system hard to detect, as it is immune to timing analysis attacks—it is completely asynchronous. In the following we describe the ACPS threat model and requirements for different distributed computing platforms. We then give implementation details as well as an evaluation of the ACPS effectiveness and performance, having provided an implementation of the designed cloud computing protection architecture.

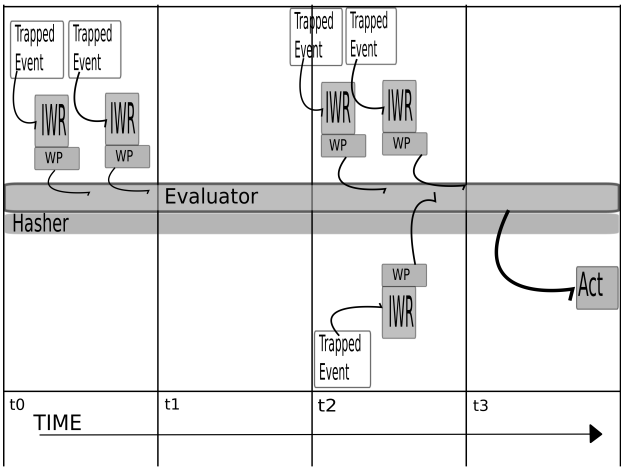

FIGURE 2.3: SWADR monitoring workflow: Interceptor and Warning Recorder (IWR), Warning Pool (WP), Actuator (Act) interactions over time.

2.5.1 Threat Model

In our model we can rely on host integrity, since we assume the host to be part of the Trusted Computing Base (TCB) [41]. When the VM image is provided by a trusted entity, guest integrity is assumed at setup time but it is subject to threats as soon as the VM is deployed and exposed to the network. Indeed, guests can be the target of possible kinds of cyber attacks and intrusions such as viruses, code injection, and buffer overflow to cite a few. In case the guest image is provided by the user, VM trustfulness cannot be guaranteed and guest actions have to be monitored to trace possibly malicious activities. In our model, attackers can be cloud users (SP) or cloud applications users (SU), whereas victims can be the providers running services in the cloud (CAT2-CAT4), the cloud infrastructure itself (CAT1-CAT3) or other users (CAT5). A traditional threat is when an attacker attempts to perform remote exploitation of software vulnerabilities in the guest system (CAT2). Some attacks are made possible by exploiting cloud services (CAT1-CAT2), since a malicious party can legally hire other instances within the cloud and, as previously highlighted, it can manage to learn confidential information (CAT5). Other attacks are also possible such as Denial of Service (CAT1-CAT2), estimating traffic rates, and keystroke timing (CAT2-CAT5) (see [20]).

2.5.2 Requirements

We identified the core set of requirements to be met by a security monitoring system for clouds are the following:

REQ1 Effectiveness: the system should be able to detect most kinds of attacks and integrity violations.

REQ2 Precision: the system should be able to (ideally) avoid false-positives; that is, mistakenly detecting malware attacks where authorized activities are taking place.

REQ3 Transparency: the system should minimize visibility from VMs; that is: SP, SU, and potential intruders should not be able to detect the presence of the monitoring system.

REQ4 Nonsubvertability: the host system, cloud infrastructure and the sibling VMs should be protected from attacks proceeding from a compromised guest and it should not be possible to disable or alter the monitoring system itself.

REQ5 Deployability: the system should be deployable on the vast majority of available cloud middleware and HW/SW configurations.

REQ6 Dynamic Reaction: the system should detect an intrusion attempt over a cloud component and, if required by the security policy, it should take appropriate actions against the attempt and against the compromised guest and/or notify remote middleware security-management components.

REQ7 Accountability: the system should not interfere with cloud and cloud application actions, but collect data and snapshots to enforce accountability policies.

There is a trade-off between transparency and dynamic reaction; we solved this problem by letting the set of possible ACPS reactions be a subset of regular guest maintenance capabilities, e.g. halting the guest, restarting it from a fresh image, and migrating the VM instance. The above actions are, from the point of view of the SU or SP, virtually indistinguishable from regular load-balance based VM operations.

2.5.3 Proposed Approach

We monitor key components that would be targeted or affected by attacks in order to protect the VMs and the cloud infrastructure. By either actively or passively monitoring key kernel and middleware components we are able to detect any possible modification to kernel data and code, thus guaranteeing that kernel and middleware integrity have not been compromised. Furthermore, in order to monitor cloud entry points, we check behavior and integrity of cloud

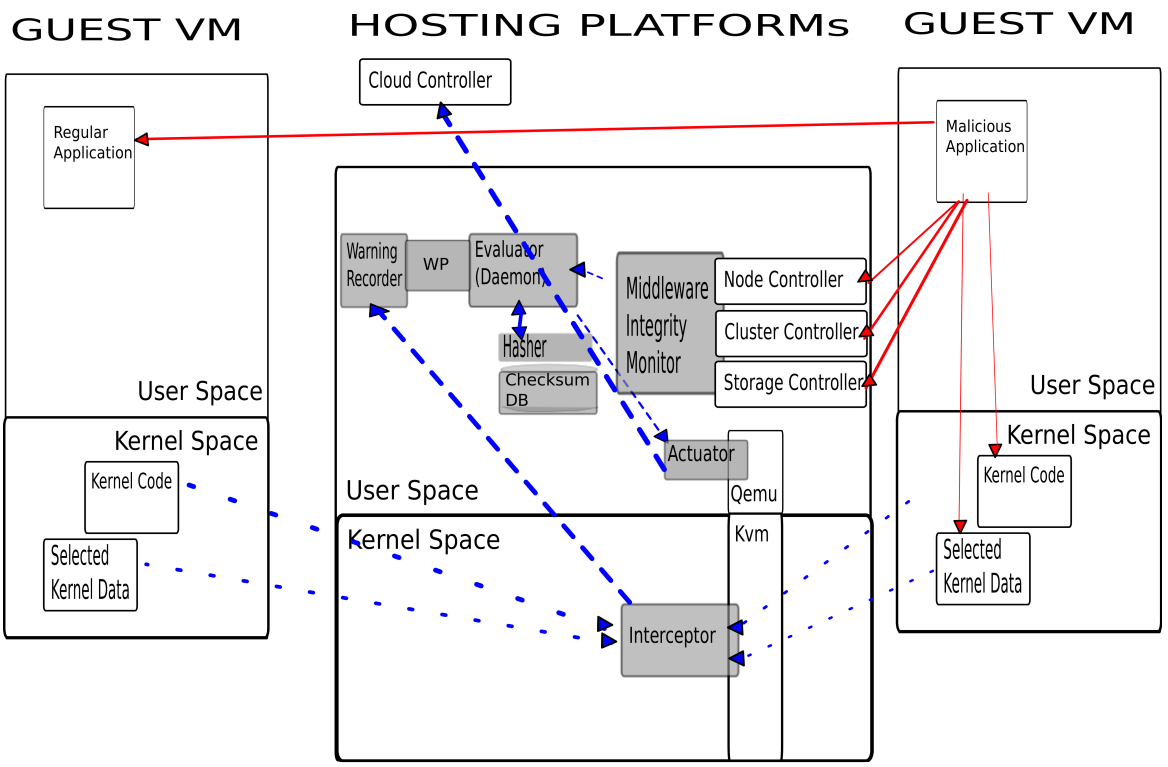

FIGURE 2.4: ACPS (components in gray) combined with Eucalyptus - Architecture.

components via logging and periodic checksum verification of executable files and libraries. A further objective we want to achieve, especially when the guest image is not trusted by the cloud provider, is ensuring that an attacker-run application cannot detect that an external intrusion detection system is in place. Note that, as for introspection techniques, it is still not clear to what extent they can be detected by the target virtual machine. In fact, the presence of a monitoring system can potentially be detected through measurement of the time it takes for certain function calls to execute. Leveraging this observation, our monitoring system acts in a way that we define *SWADR*—synchronous warning - asynchronous detection and response. In particular, ACPS can provide protection:

PRT1 from attacks coming from outside the cloud;

PRT2 from attacks coming from sibling VMs;

PRT3 from attacks coming from VMs.

The high level description of ACPS combined with Eucalyptus, and ACPS combined with OpenECP architectures, are shown in Figures 2.4 and 2.5 respectively, where potentially dangerous data flows are depicted in continuous lines and monitoring data flows are depicted in dashed lines. All ACPS modules are located on the Host. ACPS makes use of *Qemu* [42] to access the guest. Suspicious guest activities (e.g. system_call invocation) can be noticed by the

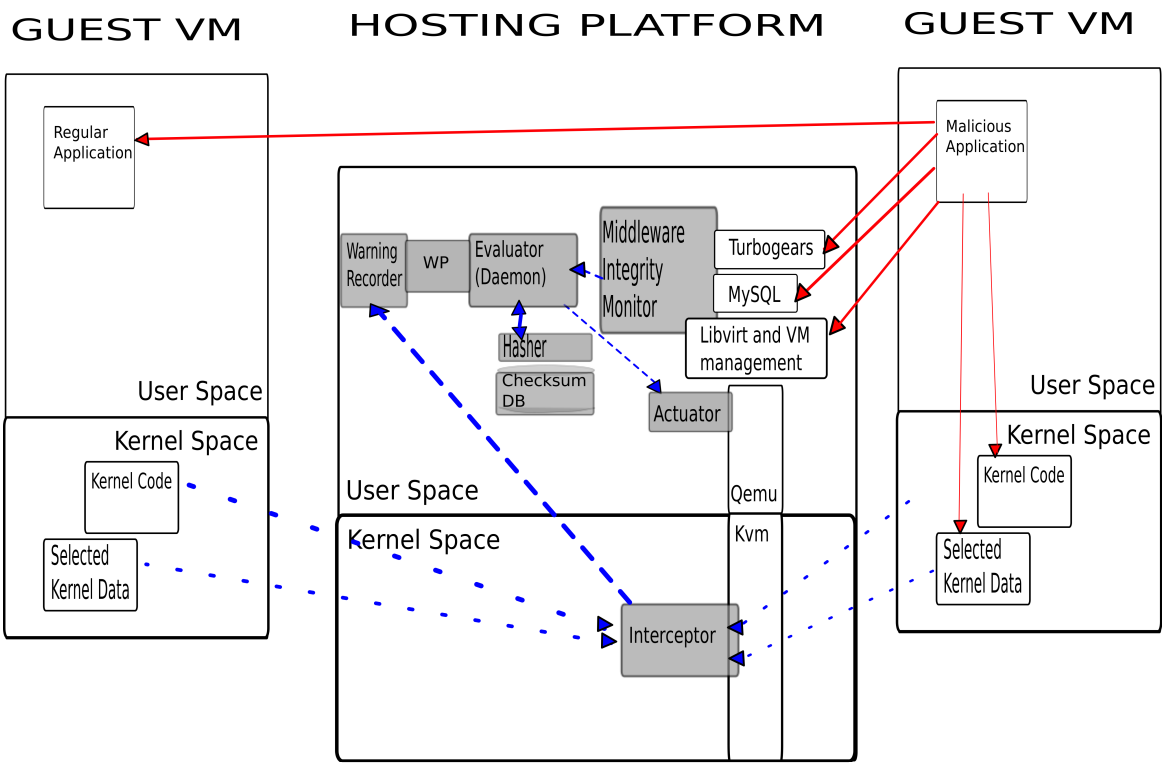

FIGURE 2.5: ACPS (components in gray) combined with OpenECP - Architecture.

Interceptor and recorded by the *Warning Recorder* into the *Warning Pool*, where the potential threat will be evaluated by the *Evaluator* component. The *Interceptor* has been conceived not to block or deny any system call, in order to prevent the monitoring system from being detected: in *SWADR* mode, the timing attack is neutralized. Indeed, the evaluation components (*Evaluator* and *Hasher*) are always active—see Figure 2.3—running and continuously performing security checks. In fact, the *Evaluator* and the *Hasher* are active and running even when the *Warning Pool* is empty. In this case, the purpose of the *Warning Pool* is mainly to cache warnings in order not to choke the evaluation component. The *Warning Pool* also allows setting priorities with respect to the order of evaluation. This guarantees increased invisibility, even though a large number of warnings might potentially delay decision and reaction by the *Actuator*. With respect to such an issue, an increasing rate of incoming warnings could be treated as a security threat on its own. It is true that, the *SWADR* asynchronous, non-blocking approach can potentially allow the attacker to perform some—limited in time—tampering with the target system. It is also true that in order to perform modifications to the guest system, the attacker must have already taken control of such system. Furthermore, the undetectability of the monitoring system allows malware behavior to be observed in a honeypot-fashion.

ACPS enjoys the following features: it is transparent to guest machines (even malicious or

untrusted ones); it supports full virtualization [43], which renders the system less detectable on guest side; and, it can be deployed on most x86 and x86_64-based distributed cloud computing platforms.

Most important, ACPS is completely transparent to guest machines, features SWADR mode (see REQ3), Warning Pools and enables hot recovery by replacement of a compromised service as well as resuming execution from the lates secure snapshot (see REQ6). ACPS is difficult to compromise even from an (already) compromised or untrusted virtual machine (see CAT1 and CAT3), while it can transparently inspect and analyze data inside guests. ACPS supports accountability (see REQ7), as discussed later in this section, and allows tracing and recording of guest status and data via snapshots, thus supporting forensics analysis. Furthermore, it has been fully integrated within existing cloud middleware. ACPS, like TCPS, is entirely located on the host machine (see REQ3). In ACPS each Virtual Machine uses its own private memory area, so it is totally independent from other VMs (see REQ4 and CAT4).

In ACPS, the host-side database *Checksum DB* contains computed checksums for selected critical host infrastructure and guest kernel code, data, and files. The runtime *Warning Recorder* daemon can asynchronously recompute hash values for such monitored objects and can file warnings towards the *Evaluator*. The *Evaluator* daemon examines such warnings and evaluates (see REQ1-REQ2) whether the security of the system has been endangered. In such a case the *Actuator* daemon is invoked to act according to a specified security policy (REQ6). Consequently, ACPS can locally react to security breaches or notify the security management layer for such components of the occurred events. ACPS can also replace a compromised server on-the-fly by restoring that VM from a clean backup image (see [31]). To avoid false positives as much as possible (REQ2), an administrator or the Cloud Controller component can notify ACPS of the new components' checksums. ACPS is integrated in the virtualization software and leverages hardware virtualization support to monitor the integrity of the guest and middleware components by performing a checksum of such objects. It is worth noting that no system_call is ever blocked or delayed by ACPS to check for permission violation and the kind of reaction our monitoring system can perform (freezing/halting/restarting the guest VM) is virtually indistinguishable from normal system maintenance tasks. Furthermore, the *Interceptor* and *Warning Recorder* can trace events, actions and the actors who performed them. These data, combined with *Checksum DB* data, can be used for accountability purposes (REQ7) [37]. This provides the proposed architecture with the necessary support to implement external secure event logging and accountability tools.

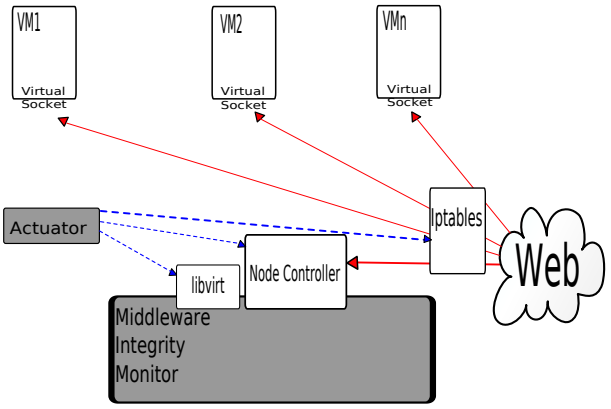

FIGURE 2.6: ACPS (components in gray) combined with Eucalyptus - detail.

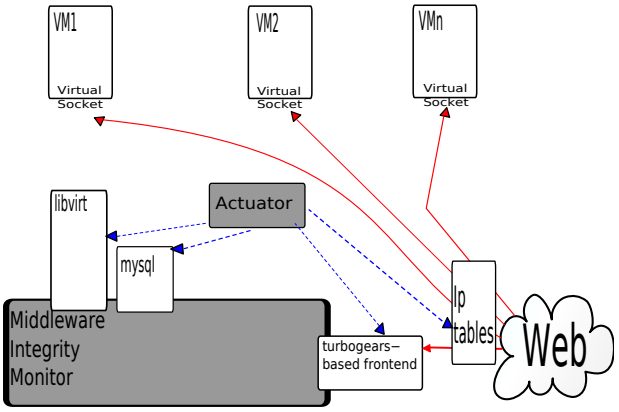

FIGURE 2.7: ACPS (components in gray) combined with OpenECP - detail.

2.6 Implementation

We implemented ACPS over Eucalyptus and OpenECP (REQ5). Eucalyptus high-level system components are implemented as webservices. Eucalyptus [11] is composed of: a Node Controller (NC) that controls the execution, inspection, and termination of VM instances on the host where it runs; a Cluster Controller (CC) that gathers information about VM and schedules VM execution on specific node controllers; further, it manages virtual instance networks; a Storage Controller (SC)—Walrus—that is, a storage service providing a mechanism for storing and accessing VM images and user data; a Cloud Controller (CLC), the webservices entry point for users and administrators that makes high level scheduling decisions.

A more detailed description of how ACPS integrates with the Eucalyptus component is reported in Figure 2.6. On Eucalyptus, ACPS can be deployed with the Cloud Controller, the Cluster

Controller and, most importantly, the Node Controller. The NC runs on every node hosting VM instances. We especially monitor NC activity and integrity, since this is the key component for this cloud implementation [44]. In fact, as shown in Figure 2.6, in case an attack or a potentially dangerous alteration is detected, the *actuator* can change the NC, Libvirt, and Iptables configuration in order to prevent further damages. The possible reactions include migrating the guests that did not raise any warning (clean guests) to other hosts, while disabling the suspicious host node itself.

OpenECP, like its proprietary Enomalism [13] sibling, provisions and manages resources by leveraging Turbogears, Python, and the Libvirt library [45]. These are the additional infrastructure resources we need to monitor the integrity of. The components that have to be monitored are Python, Libvirt and Mysql processes, executable files and libraries, as well as configuration files. Turbogears front-end components need to be especially monitored, since they are particularly exposed to the network. Such monitoring provides integrity protection for both front-end and back-end systems (against CAT1). Enomalism integration details are shown in Figure 2.7. In particular, in case an attack or a potentially dangerous alteration is detected, the *actuator* can change the Mysql, Turbogears, Libvirt and Iptables configuration in order to prevent further damages. The possible reactions include filtering out selected web requests, migrating clean guests to other hosts and disabling the suspicious host node itself.

2.7 Effectiveness - ACPS under attack

In this section we show how our proposal copes with attacks the cloud can be subject to in real environments. In particular, we report on the practical experiments performed to assess the resilience of the proposed architecture and also provide discussion on how the key requirements set in previous sections are met by our proposal.

The detection capabilities (see Table 2.2) of our system are assessed against known attack techniques (see Table 2.1). However, since source code for many attacks is not publicly available, we performed our test by simulating the attack steps.

As shown earlier, we can partition attacks into 5 categories, ranging from CAT1 to CAT5. ACPS has been proven to detect and to react to attacks belonging to the above mentioned categories, in a way that is summarized in Table 2.2. In particular, we took from the current literature some relevant attacks that actual networked architectures can be subject to ([46],[20],[47]) and we

Category	Implemented Attack
CAT1	Apache vuln. (Eucalyptus)/ssh Python vuln. (OpenECP)
CAT2	Sebek rootkit
CAT3	network probing
CAT4	colocation, detection
CAT5	colocation, keystroke timing

TABLE 2.1: Attacks instantiation.

showed the degree of added protection provided by ACPS to guests and VMs when the system is exposed to such attacks.

In particular, we simulated an attack of type CAT1 by exploiting host service vulnerabilities (see Debian ssh [48] and *Apache* vulnerabilities [49]). In this case ACPS monitors the Apache process behavior and memory footprint and notices the abnormal memory usage and connection attempts. Once the attack is detected, ACPS restarts the compromised service from a verified executable and re-establishes its configuration files.

We implemented an attack of type CAT2 by inserting a Sebek rootkit [50] in a guest VM. Sebek is a kernel module that hides its presence and intercepts filesystem and network activity. It does so by altering the syscall table in order to change the execution flow and to execute malicious code. Here ACPS detects both the alteration of the syscall table and the change in the checksum of kernel files on virtual storage.

We also implemented an attack of type CAT4 using a kernel data attack as described in [29]. In particular, given that network cards are emulated by the underlying *Qemu* software, guests are protected by ACPS from the Lying Network Card attack approach. In this context, we implemented the process hiding approach that allows the attacker to run tasks without having them appearing in the list of processes. This attack has been accomplished using a dynamic data attack leveraging /dev/kmem to manipulate the task list structure. ACPS detects the alteration when, by navigating the kernel scheduler task list, it discovers that additional hidden structures are present. As a reaction, ACPS restarts the guest from a clean VM disk image.

Finally, we implemented attacks of type CAT3 and CAT5 by using the techniques cited by Ristenpart in [20]. First of all, both external (outside the cloud) and internal (from sibling VMs) network probing via port scanning is intercepted by Iptables rules that raise an alarm to the Warning Recorder (WR). As regards keystroke timing [20], given that the attacker resorts to

Attack Technique	**Detection reason**	**Implemented Reaction**
Apache/Python/ssh	process footprint	service migration/restart
Sebek	altered sys_call table	clean VM restart
Process Hiding	tasklist navigation	clean VM restart
colocation, network probing	Iptables monitoring	Silently filter/drop network packets

TABLE 2.2: ACPS Detection/Reaction capabilities

co-residence load measurements to analyze the time between keystrokes and collect sensitive information, ACPS renders such attack less feasible. Indeed, having ACPS running on the CPU under attack makes times measurement much harder for the attacker.

2.7.1 Anatomy of Attack and Reaction

In the following, we describe the details of a sample attack we performed and the reaction we obtained from ACPS (host and guest systems' integrity is assumed granted at time t_0):

1. the attacker (ATT) exploits a ssh vulnerability ([48]) or a weak password to get access to an account;
2. ACPS *Iptables* logs to the Warning Recorder (WR), the number and targets of ssh connection attempts.
3. ATT then performs a symbolic link privilege escalation attack ([51]) to gain root privileges;
4. ATT patches critical kernel syscall code;
5. ACPS *Interceptor* notices the operation on the kernel object and files and records such potentially dangerous operations in the WR;
6. ACPS *Evaluator* fetches warnings issued by the *Warning Recorder* and checks for the integrity of affected parts by comparing checksums;
7. when the alteration is detected an alert is issued to the remote security-management component; furthermore the VM is stopped and re-initiated (see REQ6).

The ACPS *Evaluator* can be configured to react (e.g. launching a service restart) when the desired number of attack clues have been collected. The increased security provided by this

Feature	**host A**	**host B**
CPU Model	Athlon 64 4400+	Turion 64 RM-72
Cores	2	2
Ram	4096	4096
Host OS	Ubuntu 8.10 (O.ECP) Ubuntu 9.10 (Eucal.)	Ubuntu 8.10 (O.ECP) Ubuntu 9.10 (Eucal.)
Kernel	Linux 2.6.30	Linux 2.6.30
VMM	Kvm 88	Kvm 88

TABLE 2.3: Host test environment.

approach must be balanced by the possible increase of both service downtime (DoS) and computing resources usage—that arise in case of false positives.

2.7.2 Performance

In this section we present the results of the experiments aimed at evaluating ACPS performance when implemented over current cloud solutions. The hardware/software configuration adopted for such tests is depicted in Table 2.3. The guest operating systems were x86 Centos 5.2 leveraging 1 virtual CPU and 1 GB RAM. Hardware virtualization was enabled on the hosts. Guests run 32-bit OSes whereas hosts run 64-bit OSes. Guest virtual disk made use of an image file on the hosts.

We tested the performance of our solution under three different types of workload:

1. CPU-intensive;
2. Mixed workload;
3. I/O intensive.

In detail, we provisioned an Eucalyptus and an OpenECP guest and measured the time it takes such Eucalyptus and OpenECP guests to perform three different kinds of operations: mp3 encoding of a wav file (CPU-intensive); vanilla 2.6.30 Linux kernel compilation (mixed workload); and, *dd* of a large file (1GB) to a disk partition (I/O intensive).

Results are reported in Figure 2.8 where bars represent execution times normalized with respect to the same test executed on a regular Kvm guest machine on the same hosts. Values are averaged over the tested CPUs and show that the overhead introduced by ACPS is quite small. There

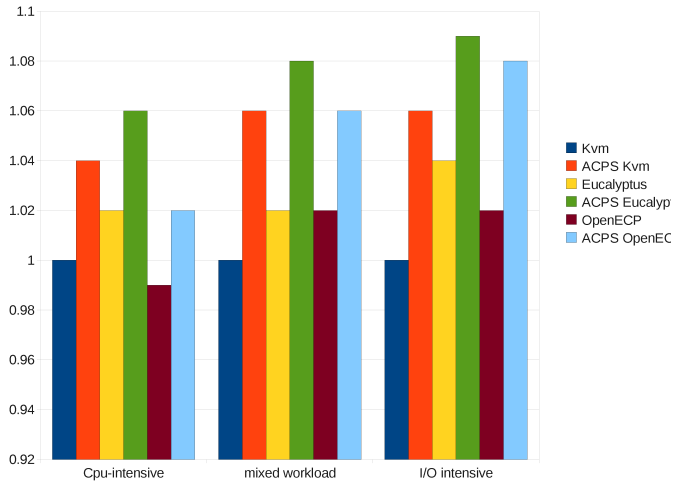

FIGURE 2.8: ACPS execution times (normalized w.r.t. Kvm) - first test round.

is a small performance loss due to the additional integrity checks ACPS performs on cloud middleware. The differences between Enomalism and Eucalyptus results can be explained by the difference in the number and complexity of the components of the two. This benchmarks helped us to quantify the actual real-world application overhead introduced by the additional asynchronous monitoring components. For this purpose, bars have to be compared pairwise, the left bar representing performance without ACPS, whereas the right one represents performance with ACPS active. Indeed, the impact of ACPS on the performance of current cloud solutions is quite limited, given that the maximum performance loss is under 6%. In particular, for the CPU-intensive test it can be as low as 3%. This result is not surprising since in the *SWADR* approach the evaluation is run as a low priority process and it is spread over time, thus leaving the CPU resources free for the most part. A slightly more complex result is obtained when looking at the mixed workload and the I/O intensive workload results. This is probably due to the increased number of active ACPS interactions with filesystem activities. However, the impact of ACPS on the performance of these types of workload never exceeds 6% and, on average, provided results are quite interesting.

We then performed a further series of tests to collect more detailed performance measurements, with a special interest in I/O subsystems. In particular, the following selected tests from the well-known Unixbench [52] test suite were executed:

1. Execl: this test measures the number of *execl()* function calls that can be performed in one second. Execl is aimed at replacing the current process image with the new one; hence, this operation stresses memory I/O performance.

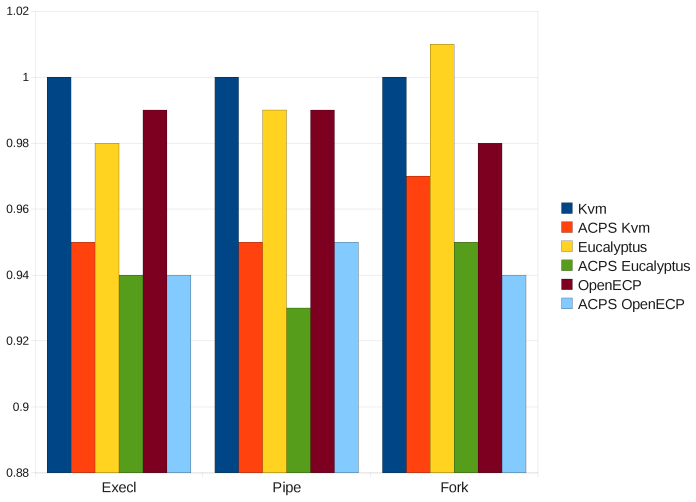

FIGURE 2.9: ACPS performance comparison (normalized w.r.t. Kvm) - second test round.

2. Pipe: this test measures the number of pipe-writes (512 bytes) a process successfully performs in one second. It is an indication of how fast the process is in performing I/O activities.

3. Fork: this test measures the number of times a *fork()* call can be invoked per unit of time. This test is an important indicator of overall performance.

Results are reported in Figure 2.9. As above, the comparison has to be carried out pairwise with respect to columns. Bars represent the number of executed operations, hence a higher bar means a better performance. This benchmarks helped us to quantify the specific-operation performance overhead due to ACPS. The good news is that performance loss due to the additional integrity checks ACPS performs is less than 6% in any test. More in detail, in this case the performance for the two cloud computing environments is very similar, as expected, to the plain KVM guest. The reason is that the cloud computing infrastructure only indirectly affects the execution of such low-level operations, whose execution depends on operating system configuration and security-related checks (even though these latter ones cannot be distinguished from normal workload). ACPS activity mostly affects I/O performance (see Pipe throughput results affected by up to 6% performance loss) whereas the fork experiment performance loss is less than 4%. Such a difference can be due to the interaction with the ACPS interception components.

Results show that there is a margin of improvement for the I/O monitoring operations. This margin of improvement can be explained by the fact that the implemented I/O monitoring is not fully mature. Indeed, it requires extra interaction with the I/O subsystem that could be reduced

in future implementations of the ACPS framework. Overall, these first results are interesting—due to the generally low overhead introduced—, and encourage us to further investigation aimed at leveraging the improvement margin previously highlighted. Finally, it is worth noticing that even though overall performance is degraded by the monitoring system itself, such performance penalty cannot be distinguished by the attacker from regular CPU load, since the system_call timing difference between protected and unprotected configurations is constantly within the 3%-6% range, which is virtually indistinguishable from the performance loss due to regular task operations.

2.8 Conclusion

In this chapter, we have provided several contributions to secure clouds via virtualization. First, we have proposed a novel advanced architecture (ACPS) for cloud protection that can monitor both guest and middleware integrity and protect them from most kinds of attack while remaining fully transparent to the service user and to the service provider; ACPS has been tailored and deployed onto different cloud implementations and has been proven able to locally react to security breaches and capable of notifying the security management layer of such an occurrence. Second, the proposed architecture has been implemented entirely on current open source solutions and both protection effectiveness and performance results have been collected and analyzed. Results show that the proposed approach is effective and introduces just a small performance penalty.

3 Monitoring Service behavior via Execution Path Analysis

Despite the increasing interest around cloud concepts, current cloud technologies and services related to security are not mature enough to enable a more widespread industrial acceptance of cloud systems. Providing an adequate level of resilience to cloud services is a challenging problem due to the complexity of the environment as well as the need for efficient solutions that could preserve cloud benefits over other solutions. In this chapter[1] we provide architectural design, implementation details, and performance results of a customizable resilience service solution for cloud guests. This solution leverages execution path analysis. In particular, we propose an architecture that can trace, analyze and control live virtual machine activity as well as intervened code and data modifications—possibly due to either malicious attacks or software faults. Execution path analysis allows the Virtual Machine Manager (VMM) to trace VM state and to prevent such guest from reaching faulty states. We evaluated effectiveness and performance trade-off of our prototype on a real cloud test bed. Experimental results support the viability of the proposed solution.

3.1 Introduction

Cloud computing allows companies to greatly reduce costs by outsourcing computation and data. However, many companies have not yet leveraged the cloud opportunity because of the

[1] Part of this chapter appeared in [2]

relevant concerns of the cloud technology when analysing related security issues [53]. As such, many potential cloud providers, customers and users still do not trust the security model and actual service resilience of the cloud. Cloud management aspects such as reliability and security have to be addressed while guaranteeing a high level of performance. A satisfactory trade-off has to be found that could allow to provide robustness to services hosted on cloud Virtual Machines. Improving the resilience of complex systems such as clouds is a difficult task due to the number and complexity of events and changes that take place at runtime [54]. Cloud services, even if not offered at IaaS level, are provided trough the interaction of a large number of real and virtual resources. A large part of present cloud resource hosting and management technology is Linux-based. Its kernel, cloud platform and services are exposed to an increasing number of threats. Malware and in particular rootkits exploit system vulnerabilities and install themselves on largely deployed hardware and virtual machines [55]. Rootkits hide their presence from spyware blockers, antiviruses, and system utilities, while at the same time allowing hidden remote access to valuable data in the compromised machine. The fact that cloud providers manage a large number of identically configured virtual machines and services, dramatically worsens the consequences of a successful attack.

Further, software upgrades and deployment of new components can potentially reduce or block cloud service functionality. Software faults can also open the way to misconfigurations that can lead to massive service disruption in the cloud [56]. Effective and efficient new techniques are needed to improve cloud resilience with respect to such issues.

3.1.1 Contribution

In this chapter we propose an advanced approach that makes use of execution path analysis to allow the cloud VM manager to react to anomalies in the VMs and contained services. The proposed system (CloRExPa for Cloud Resilience via Execution Path analysis) traces, models and analyzes system behavior via introspection techniques [57]. CloRExPa leverages virtualization to allow cloud management to faithfully build a model of running guest VMs and services in order to protect them from attacks and software faults.

Our solution adopts a novel approach to execution path analysis (EPA). EPA is applied to scenario and action graphs and it is used to prevent the VMs from reaching faulty states. The CloRExPa architecture is described as well as how it can be used to provide protection to cloud

resources and services. Clouds natively match the requirements for CloRExPa as cloud middleware supports hardware virtualization extensions. Moreover, clouds greatly benefit from the proposed approach as service disruption/misconfiguration and malware can cause extensive damage an thus important consequences on large deployments of identically configured services such as clouds. In particular, cloud service management and migration facilities can potentially work in conjunction with CloRExPa in order to prevent and block possibly massive replicated service attack and disruption. A first CloRExPa prototype implemented on a standard Linux cloud host platform is shown.

Effectiveness and performance trade-offs of the prototype have been evaluated: Obtained results show the viability and the achievable benefits of our proposal.

3.1.2 Roadmap

The remainder of this chapter is organized as follows: Section 3.2 summarizes state of the art solutions applied to cloud resilience; Section 3.3 describes the proposed approach; Section 3.4 gives further implementation details; Section 3.5 presents and discusses effectiveness and performance figures; finally, in Section 3.6 conclusions are drawn.

3.2 Related Work

Virtualization has been largely leveraged to protect VM integrity on the cloud [58][59][60]. A remote integrity attestation system that considers current behavior of a system has been proposed in [61] where two virtual machines are executed, an actual one and a shadow one where integrity checks are performed. This is more costly than our solution as it halves host efficiency since per each VM one spare VM has to be run and updated. Further, no guarantee is given that the environments of the two VMs are identical, and as a consequence, their behavior.

3.2.1 VM Monitoring and Security

3.2.1.1 Hidden Object Detection

Three main approaches have recently been proposed to detect hidden objects within a guest operating system: the first one is based on operating system modifications, the second one

leverages additional hardware, and the last one is based on mechanisms residing in the virtual machine monitor (VMM). GhostBuster [62] is an example of hidden object detection mechanism designed at the operating system level. Because of that, it can potentially be tampered with, bypassed, and disabled. There are also methods based on additional hardware. Copilot [63] is a PCI card hardware-based solution with a monitor software running on the Card. This software will periodically scan the memory, obtain a memory backup and then access the system objects list. However, this requires specific additional costly hardware. In a virtual computing environment, both methods above have some shortcomings. Whereas, the virtual machine monitor natively allows privileged access to VM state. The main advantages of a solution based on a "virtualized" environment is that the VMM can access VM state and data.

However, it is not easy to cross the semantic gap and to interpret the raw data inferred from the VM. Antfarm [64] uses the CR3 register to distinguish process-related operations such creation, context switches and termination but it cannot get full guest semantic information. VMwatcher [65] obtains a more reliable view of the guest operating system from the VMM layer, but its passive approach allows some events to escape tracing. Ether [66] is an active VMM-based monitoring platform that "traps" guest events to detect cpu, memory and system calls but the inferred information is coarse-grained. Lycosid [67] can get process information from the VMM and analyze hidden processes. However it does not allow to bridge the semantic gap and it only works for hidden processes. SIM [68] adopts a hybrid approach as the detection module is placed in the VM, but in a separate address space. However its detection scope is limited as it is effective only for process creation and system calls. The most recent work in this area is VMDetector[69]. VMDetector is transparent to guests and more accurate than previous solutions as it monitors kernel-level process and also (hidden) network activity.

3.2.1.2 Intrusion Detection Systems

Much work has been done in the area of host-based IDSs [70]. We can classify IDS technologies as:

- Program-level IDS (An IDS that uses information available at the application abstraction level). Wagner et al.[71] show how static analysis can be used to thwart attacks that change the run-time behavior of a program. They build a static model of the expected behavior and compare it to the run-time program behavior. Also Kirda et al.[72] were

able to achieve the same result but they leveraged both static and dynamic analysis to detect malicious behavior.

- OS-level IDS (An IDS that makes use of information available at the OS level such as system calls and system state). Many intrusion detection systems have used system call profiling to detect malicious code [73–75] since system call tracing can be performed efficiently and can provide useful insight into program activities. In Bezoar[76], a VM is used to provide recovery from zero-day control-flow attacks due to additional "integrity bit", injected into memory addresses and cpu registers.

- VMM-level IDS (An IDS that uses semantics and information available at the VMM-level). This set of IDS can additionally be subdivided into two subsets, "Hybrid" and "Pure" VMM IDSes. Hybrid VMM/OS intrusion detection systems leverage the VMM to isolate and protect the IDS. However, they rely on OS-level information and therefore they are not pure VMM IDSs. Pure VMM intrusion detection systems try and interpret from outside (VMM) the semantics of the VM. This limits the amount of information available to the IDS and poses a greater challenge. Laureano et al.[77] and VNIDA[78] are based on an hybrid solution while Azmandian [79] uses a pure solution.

The works presented above have the following drawbacks:

- Monitored Targets: both IDSes and detection tools are built to protect the guest environment against "malicious" objects or events;

- Privacy: they leverage semantic introspection [80] to reconstruct guest structure like process list or module list and so they have access to guest data.

Our approach addresses the above drawbacks as follows:

- Monitored Targets: since our solution is about "fault resilience", it is not focused on "malicious" events but more generically on changes that could lead to a system fault or a service unavailability. This is important because not only a malware can lead to a fault, also a software update or the installation of additional software can lead to the same effect;

- Privacy: If the cloud provider is malicious, he/she can infer information by intercepting guest events. However, the approach proposed in the present chapter does not maintain

user data but rather application state. VMM instrumentation can actually intercept and dump guest data. However this is a general issue of all introspection techniques. In fact, in our proposal, privacy is guaranteed because we do not implement any semantic introspection over customer data. Further, to preserve guest privacy we do not rebuild guest system objects but maintain checksums of raw data.

3.2.2 Modeling Complex Systems

Various approaches have been proposed for modeling complex systems. As regards system state representation two main approaches exist: Data Set [81] and Finite State Machines [82]. Hidden Markov Models (HMMs) are also widely used to model and analyze the state of a system [83].

Further, Execution Path Analysis [84] has been leveraged to explore multiple execution paths, enabling the semi-automatic monitoring of running programs. In particular Moser [85] analyzed malware actions by examining VM actions and snapshots.

However, these approaches have drawbacks:

- **Unobservability:** HMM state represents some unobservable condition of the system being modeled. This is a drawback in that it makes it harder to detect anomalies.
- **Unfaithful Representation:** HMM transitions and symbol probabilities are initialized randomly and then possibly adjusted at runtime. This renders the system model less faithful.
- **Memory Occupancy:** State transition graphs can be fully connected. For each state, transition probabilities and probabilities associated with producing each system call have to be stored. This requires large amounts of memory and reduces scalability.

Compared to previous solutions, our approach addresses the above drawbacks as follows:

Unobservability: Present virtualization technology allows to transparently (i.e. during a VMEXIT operation) take snapshots of a running VM without affecting VM execution or service quality. Such snapshots comprehend the overall state of all VM structures.

Unfaithful Representation: CloRExPa graphs are not randomly initialized. Every vertex (also node in the following) or transition corresponds to a system state or a system action actually

happened in the past. By updating graphs at every system call, CloRExPa can trace VM transitions for a real scenario. As described in Section 3.4, tracing is performed using virtualization technology.

Memory Occupancy: CloRExPa graphs represent states and actions on actual paths. Such graphs are not necessarily fully connected. In fact, the number of nodes and paths depends on the actual executed paths of every single VM, given that graph nodes and links are created at runtime. This does not require a large amount of memory and as such improves scalability. As described in Section 3.3.1, a "Garbage Collector" technique that could further reduce the spatial complexity is under development and will be available in next version of CloRExPa.

3.3 CloRExPa

CloRExPa is a monitoring system for guest resilience via VM-state-modeling scenario graphs [86], [87]. CloRExPa addresses previously-mentioned drawbacks by leveraging hardware supported virtualization to faithfully trace execution while modeling guest state with a multi-layer data structure. CloRExPa makes use of a novel scenario graph approach to model guest system state, and leverages both kernel-level and user-level execution tracing. This latter can be tuned and limited to specific applications (e.g. web server, application server) in order to reduce the performance overhead.

CloRExPa makes use of two different kinds of scenario graphs, namely the **state graph** and the **action graph**. The first one models VM state, where a VM **state** represents the snapshot of any VM data structures. The second graph type models all actions performed by the VM at runtime, where an **action** can be any assembly instruction (e.g. the invocation of any system call). It is important to stress that these two graph types differ both as regards node semantics and as regards edge semantics. A directed edge in a state graph represents an action leading from a state to another. The edges of an action graph establish an "happened before" relation between two actions on the same object. Both these two graph types are directed and disconnected. As later described (see Section 3.3.2) any connected component represents a single VM structure. By combining (overlapping) such components useful information on global VM state can be gathered.

CloRExPa graphs are automatically generated and do not require manual intervention or correction. This is an advantage w.r. to scalability issues, given that a CloRExPa system can increase the stability of its guests over time.

Every path in a scenario graph is an ordered sequence of actually performed actions. Any connected component is composed of different nodes that are classified as follows (see Figure 3.1):

- **Start Node**: represents VM state at launch time. In a scenario graph such node represents the root of any connected component.
- **Faulty Node**: node representing a state where the VM or service experienced a severe fault (e.g. a kernel panic) or has been victim of a successful attack. Each path from an initial state to an unsafe state represents a system execution that violates a safety property.
- **Healthy Node**: node representing a state from which no path leads to a faulty node.
- **Sick Node**: node representing a state from where at least one path leads to a faulty node.

It is important to stress that a sick node can turn into an healthy node over time (and vice-versa) due to the graph update tasks that can modify paths at runtime. In fact, CloRExPa learns from VM execution and updates its internal data at every system execution. So even if the current path execution surely leads to a faulty node in the graph, we cannot ensure that it will lead to a VM fault in practice.

The proposed graph system does not actually require an end node, even if such node could be considered as representing an operating system crash event (like a kernel oops). In fact, it is always possible to change such end node in order to execute at least one more instruction (in this example we could add some extra print in the kernel oops handler). In other words, if such end node would exist in our model, than we would have a “final” system state. From that state it would not be possible to go in any other state. This is not correct in our model as it is always possible to make changes to some system object and thus generate a state change. As such, there is no end node in our “dynamic” system. As an example, even a shutdown cannot be considered an “end” point. In fact, supposing that a generic node, representing a system shutdown, is an end node, we can start the system and again execute the same steps as before until the end node is reached. Then, we could execute a new step from the same node that was

previously considered an end node. The frequency of state and action graph updates depend on how CloRExPa monitoring is set up:

- Asynchronous: CloRExPa checks the system state at a given time interval. This interval is in the order of a few seconds and can be tuned in order to provide the desired security-performance trade-off.
- Synchronous: CloRExPa checks the system state before every *potentially dangerous* action. The overhead of this solution is greater that the former because the control frequency cannot be decided by CloRExPa but it is still possible to tune it by adjusting the number of actions being monitored.

When the VM ends-up in a sick node from where all paths lead to faulty nodes, CloRExPa can take different approaches:

1. Enter the sick node and walk through a path of sick nodes. This would most probably lead to a faulty node. Advantage: normal VM behavior does not get altered. Drawback: possibly high number of VM (snapshot-based) rollbacks (depicted in Section 3.4) and consequently high overhead.
2. Prevent the VM from entering that path (if possible) e.g. by blocking that specific activity while allowing all the other ones. Blocking the specific activity allows the VM to continue executing. Advantage: small VM overhead. Drawback: altered execution of the VM. e.g. if the requested action was a legal software update, that update will not be performed.

As regards the scalability of the model when facing large-scale programs and long-running executions, these are managed in a similar way. A large-scale program will likely lead to the creation of large graph structures. Despite that, CloRExPa runs in a different thread in the host environment, so the overhead due to the control system can be limited (see Section 3.5). This implies that, the larger the program is, the larger and more complicated the graphs are. This will require a large amount of memory. So the problem has shifted to the host machine, where it can be handled with load-balancing techniques. As regards long-running programs they are, from the CloRExPa point of view, the same as multiple short-running programs, due to the fact that CloRExPa stores system execution information in the filesystem. The crucial aspect is not how long a program will execute, but how many system structures it will alter and how often.

Stress executables (that rapidly perform changes over system structures) have been deployed on CloRExPa and results are shown in Section 3.5.

We would like to point out that our solution does not currently ensure that the whole guest environment never reaches an unsafe state, i.e. it is in a Faulty Node. CloRExPa ensures that a subset of guest objects are kept in a safe state. This could actually lead to unpredictable process behavior, depending on the complexity of interactions among monitored objects in the guest. As an example, a customer hires some "cloud resources" to host a service on the web and this service requires web server, application server and database server. CloRExPa ensures that those servers will remain in a safe state. If some event (for example a software update) causes some component of the system to enter an unsafe state, our solution makes it possible to rollback (or prevent) such update and restore the servers to the previous safe state. Even though some other objects of the system could remain in a sick state, the proposed solution ensures that the state of the monitored servers is safe. However, this depends on the choice of objects to be monitored. If the whole system has to remain in a safe state, then all objects/structures of the operating system should be monitored. However, a reasonable objective is to control a significant subset of guest objects. In general, therefore, is important to pay attention when selecting the subset of guest system objects that have to be controlled.

3.3.1 Scenario Graph Management

Every node in a scenario graph is identified by a triple (**H**,**D**,**S**) where: **H** is a uniquely defined node ID, **D** is the distance between the current node and the closest faulty node, and **S** is the node type (one of the four node types listed above). An example of a scenario graph is given in Figure 3.1.

The first node of each connected component created by CloRExPa represents either the initial state of the monitored structure (in a state graph) or the first operation executed on such structure at VM launch time (in an action graph). As an example, with respect to the IDTR (Interrupt Descriptor Table Register), the first action is the *lidt* operation that writes an address value inside the IDTR. So the start node of the the IDTR state graph will be created when the first *lidt* occurs. All other actions involving IDTR changes possibly create other nodes in the IDTR state graph. The distance value is used by CloRExPa to choose the best avoidance/recovery approach. Let T be the number of past snapshot we could jump back into. This value is used in

conjunction with the distance value D in order to decide if a rollback is convenient/useful or not. As an example, if $D > T$ then even rolling back T steps would not lead us to an healthy system.

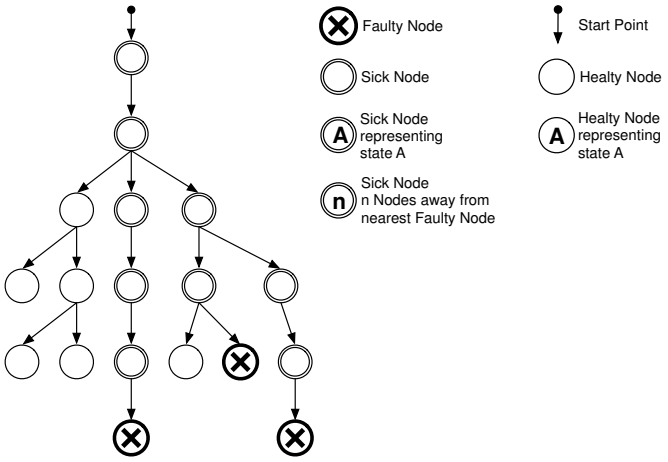

FIGURE 3.1: Scenario Graph for a given VM, last n nodes on paths leading to faults are sick nodes

Given that a running VM can have a very high number of possible states, node (number) explosion has to be prevented. At present a "Garbage Collector" mechanism has been devised to limit the number of nodes. The Garbage Collector collapses "equivalent paths" into a single node. A path is CloRExPa-equivalent to another one when both paths are composed of the same nodes but with different order. Such paths are collapsed into a single node, and such node labeled as a safe node or faulty node depending on the node they lead to.

As stated before, our scenario graphs dynamically adapts to guest evolution over time. The size of the state and action graphs is constantly monitored given that graph updates are always performed on both types of graph.

3.3.2 Multi-layer Scenario Graph

An attack can affect a single or multiple system objects [88]. In order to guarantee that the VM is in a healthy state we need to monitor a variety of system objects at the same time. This is the reason why we adopted a *multi-layer graph* [89] approach that allows to join all connected components of a scenario graph in order to gather useful information for the whole VM. So a single graph is created for every system object. As an example, depicted in Figure 3.2, we built three graphs for respectively, the System Call Table, the Interrupt Descriptor Table (IDT) and the Interrupt Descriptor Table Register (IDTR). In Figure 3.3 a side view of such "graph stack" is depicted where each component represents a layer of the final graph.

Analyzing all layers together, gives a far more complete view of the VM state and allows to analyze a much more accurate model. In Figure 3.3 an example of VM state is depicted. In this example nodes representing a particular VM state are filled with color. The dashed area represents a specific VM state. It is worth noting that, for a given layer, only one node can belong to a dashed area since for any VM state at any time, every system structure can be in only one state (represented by the snapshot) and so in only one node of the graph. Using multi-layer scenario graphs and collecting information from a set of kernel objects has the advantage of information reuse and reduces the size of each scenario graph. Such an approach reduces the number of edges that we would have if we built a single connected component graph for all these structures.

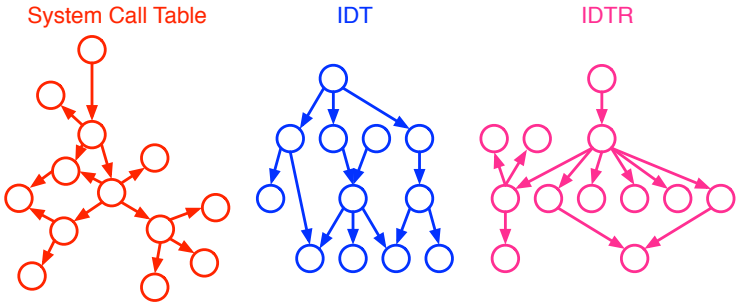

FIGURE 3.2: Multi-layer Graph Components

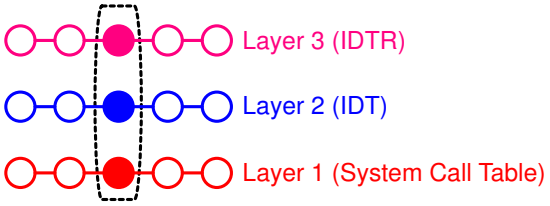

FIGURE 3.3: Multi-layer Graph Side View Layout

3.3.3 Action Graph

State Graphs represent VM system state. The main drawback of this approach is that authorized updates can change a substantial part of the guest system, possibly leading to drop a large part of the graphs that have been built.

This is the reason why we also introduce a new type of graph called *action graph*, where sequences of performed actions that lead to changes are kept.

Figure 3.4 depicts an example of the initialization phase of an action graph. Starting from "S" we see four paths leading to nodes A,B,C and D respectively. Node A represents some changes

performed starting from S. If such graph represents the System Call Table, this means that an operation A involving the System Call Table has been performed (e.g. a new system call has been added). We can then follow the path S → A → X obtaining a sequence of operations performed on the same system structure.

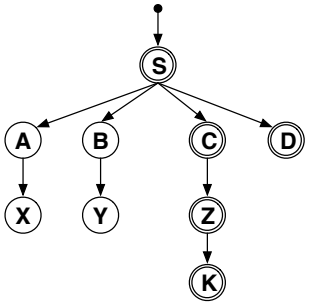

FIGURE 3.4: Generic Action Graph

Combining information from different scenario graphs is beneficial to the analysis of the VM execution state. The main advantages of this new combined approach are:

- Focus on attacks: it allows to focus on the sequence of actions performed by a malicious party.
- Reuse: even if different VMs have different data structures, the *modus operandi* i.e. the sequence of actions of the same malicious software is very similar. Thus, keeping track of the actions performed by a rootkit in a VM will help in detecting and preventing the same rootkit on different VMs.

In Figure 3.5 two action graphs are depicted, one for a Fedora-based VM and one for an Ubuntu-based VM. Suppose that S → T are the changes performed by a malware (i.e. rootkit R1) and that after change T the VM will end up in a faulty state, whereas the path S → A → F → U represents trusted changes performed by the OS. Then, after reaching the faulty node T, the Ubuntu VM will know that it has to be avoided.

Suppose now that the Fedora VM has never been attacked before by rootkit R1 so there is a good probability that R1 could also succeed in infecting this machine. By "merging" the two action graphs, knowledge learned by the Ubuntu VM will also be useful to the Fedora VM.

However, action graphs have some drawbacks. In fact, no assumption on the integrity of a system can be done only based on its action graph given that system integrity depends on system structures, data and code content.

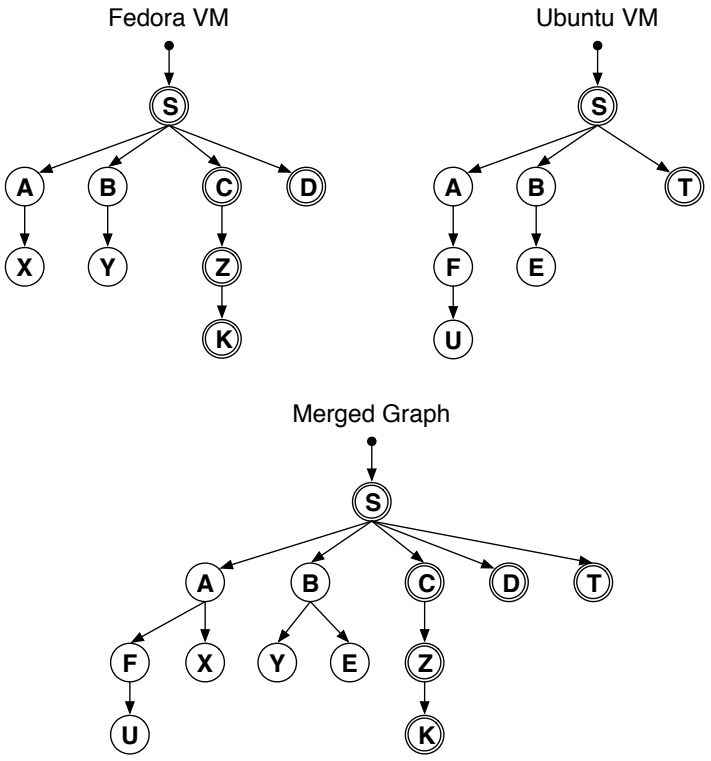

FIGURE 3.5: Merged Action Graphs

So both action and state graphs are required in order to check that the system is in a faulty state.

3.3.4 Graph Cooperation

As depicted, both state graphs and action graphs have limitations. Combining these approaches allows to leverage corresponding benefits, while reducing related limitations.

In particular we can divide the cooperation into two actions:

- **State Graph→Action Graph**: as we can see in Figure 3.6 when we fall into a faulty state inside the state graph, we already know the state of system data structures that allowed such an attack (they are faulty nodes in the state graph) and so we can label the actions that led to that faulty node as sick. That is, the two graphs (action and state) complement each other, as the state graph helps building the action graph. The state graph detects a fault and then labels, in the action graph, the sequence of actions leading to that state.
- **Action Graph→State Graph**: in Figure 3.7 we provide an example of how the action graph can help the state graph. Given that similar attacks perform similar sequences of actions, even though they cause different effects on different targets, information collected

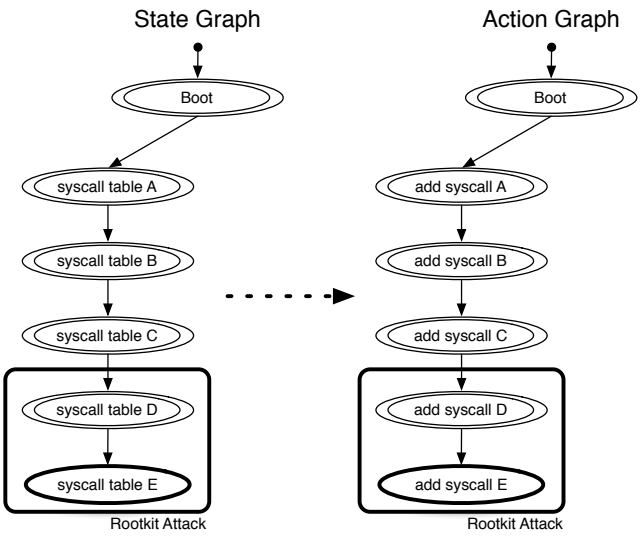

FIGURE 3.6: Creation of an action graph path based on a crash found by the scenario graph

on similar attacks can be reused for different targets. Suppose that we have performed a legal update, so our monitored object has radically changed and we have lost a lot of information. As an example, after a kernel update, data structures might be initialized in a different way w.r. to previous kernels.

In the example of Figure 3.7, system call table A is the initialization state of the system call table for some kernel version. After a kernel update (or some other large update) the initialization state of the system call table, say R, could be different from the first one. Thus, even if subsequent operations executed over the system call table will be the same, the new system call table could be different from the first one and so even the paths in our graphs will be different. This would mean using a new node in the graph and as a consequence the loss of previously-collected information around the old node. Re-launching a well known attack on such a different structure will generate a different path on our scenario graph such that we can not state if such path will lead to a faulty or to an healthy state. However the action graph is based on actions performed by the attacker, so independently from the state of the structure, this attack behavior can be detected in the action graph.

In the example in Figure 3.7 we performed action sequence A-B-C-D-E and then we checked that D-E lead to a fault. At a later time, suppose that the system call table has radically changed (e.g. due to a kernel update) and that A-B-C-D-E is being executed again. Now we already know that D-E could lead to a fault (due to the previous execution) and so we can decide to not execute it.

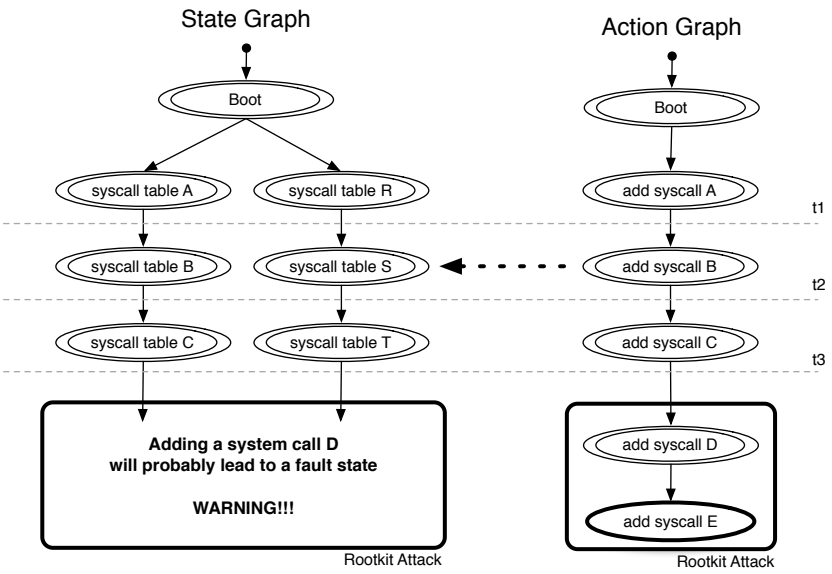

FIGURE 3.7: Check of a new unidentified path on the scenario graph based on other systems action graphs at times t1...t3

3.3.5 Node Labeling/Relabeling

Node labeling is a delicate operation in CloRExPa. It updates the node type and the distance from the closer faulty node. Usually, this kind of operation on a node is executed at the moment the node is created. Our solution takes a different approach since we start the labeling operation when the VM falls into a faulty state. At that moment we start (re)labeling the nodes of the new path in reverse order, as depicted in Figure 3.8.

It is important to highlight that executing the "labeling routine" when the VM is in fault allows us to perform any kind of operation over the graphs without adding overhead.

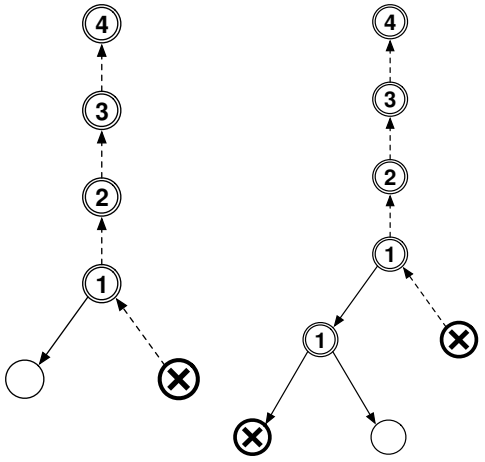

FIGURE 3.8: Graph labeling (node values represent the distance *D*)

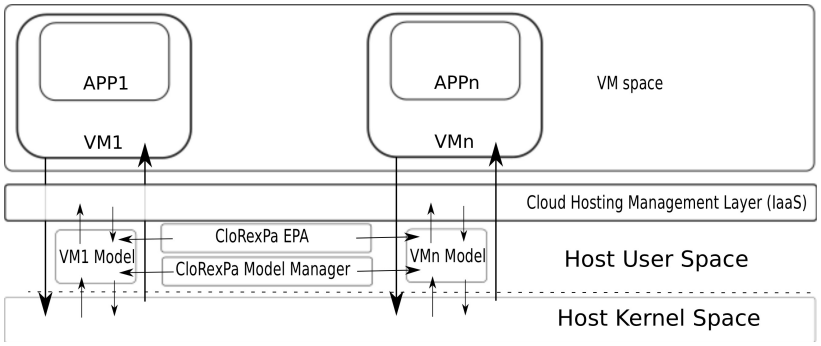

FIGURE 3.9: CloRExPa Architecture

3.4 CloRExPa Implementation

The architecture of CloRExPa is depicted in Figure 3.9. VM state graphs, model update and management components (CloRExPa Model Manager), as well as the Execution Path Analyzer (EPA) module, are all contained in the host user space. Note that no CloRExPa component runs inside the guest VM.

3.4.1 CloRExPa Model Manager

The Model Manager component of the CloRExPa architecture is in charge of managing the graphs of each VM. It loads graph data at startup and stores graph data at shutdown. Also, the Model Manager deals with the node *relabeling* operation.

3.4.2 CloRExPa Execution Path Analyzer

The EPA module is important since it deals with handling execution requests of any operation by the VM. It is the EPA module that chooses between *preventing* or a posteriori checking for execution errors. Those decisions are based on the action graph as detailed in Section 3.3.3. CloRExPa is inspired by the ACPS [1] work on VM protection and leverages semantic introspection [90] in order to update its model.

It is worth noting that in this work we assume the physical host to be secure. As such, given that CloRExPa is entirely host-contained, we assume CloRExPa is secure. CloRExPa has been implemented on both kernel and user space. In kernel space it was necessary to modify the hypervisor source code in order to trap any VM action that could hide a malicious action. It is

important to highlight that it is possible to use the CloRExPa core engine with any hypervisor. The only requirement is that the hypervisor could trap some desired guest instructions and notify the user space component of the virtual environment (in our case *qemu*) in order to wake up CloRExPa. The trapping granularity can be tuned in order to monitor a specific operation (e.g. a specific system call invocation) or a group of system calls. In this first version of CloRExPa we decided to trap the "sysenter" instruction in order to be able to monitor every system call invoked by the VM operating system.

Our prototype can intercept the sysenter/sysexit instructions by the GPE (General Protection Error) caused by the overwriting of the code segment register (e.g MSR_SYSENTER_CS for AMD) [66][91]. We chose to use GPE [91] in place of the PFE, given that PF is frequently used during the normal execution of an OS. In contrast, the GPF is rarely invoked and so it can be leveraged to handle only the sysenter/sysexit instructions, speeding up the whole guest execution. In order to do that we had first to setup KVM to trap the GPE error with a function named gp_interception.

```
SVM_EXIT_EXCP_BASE + GP_VECTOR = gp_interception;
set_exception_intercept(svm, GP_VECTOR);
```

Then, in the function that takes care of setting the MSR register we modified the value of the MSR_SYSENTER_CS as follows:

```
svm->vmcb->save.sysenter_cs = faulty_sysenter_cs;
```

Now we only had to emulate the requested instruction when a GPE occurs. However, in the gp_interception function we first recover the original MSR_SYSENTER_CS value, than we invoke the emulation, and finally we rewrite the original value again in order to be able again to trap other system calls.

```
svm->vmcb->save.sysenter_cs = original_sysenter_cs;
er = emulate_instruction(&svm->vcpu, 0);
...
svm->vmcb->save.sysenter_cs = faulty_sysenter_cs;
```

It is important to stress that even if the above code represents less then 1% of the whole hypervisor, such code is injected in the core system of the hypervisor and so all operations must be performed carefully. Above all it is important to take care of:

- MSR overwrite: whatever MSR register is used to trap into the hypervisor, it is important to overwrite the correct value with a canary value;
- Single step emulation: it is important to rewrite the correct value to the above MSR for only one step of emulation and before going back to the guest otherwise it will not be possible to trap the GPE again.

The described modification to the hypervisor can also be ported on any kind of hypervisor other than KVM and on any platform supporting hardware virtualization. In order to do that is important to take care of:

- Exit Register: depending on the platform (AMD/Intel) we have to check that the architecture exposes interception configuration registers that allow selecting the subsets of guest instructions that have to be trapped.
- Intercept Handler: we have to select the event that will cause the VMExit and create an handler for it.

CloRExPa analyzes the requested operation (i.e checks if the corresponding node is already in the graph) once the operation gets trapped by the guest. It checks whether such action leads to a safe node or not (we know that if node is safe there are no paths leading to a fault, if node is sick such path exists). In case this information is not known (there are no paths from the current node with an edge corresponding to the requested action) the operation is then executed and a new snapshot computed representing the new state, that will be a new node in the state graph.

CloRExPa is required to monitor complex data structures. Techniques to protect the IDT and the syscall table are well known, but other structures such as the open connections table and some user space application config files (e.g. Apache config or /proc virtual filesystem) have to be monitored with a different approach. In particular, the semantics of each data structure and its evolution over time are completely different. In order to be effective, the CloRExPa monitoring system represents the meaning of the monitored structure by using semantic introspection [80].

As regards rollbacks, given a rollback of *X* operations, all system operations and all data (both user and system) computed in such interval would be lost. This could seem as a drawback at the first glance; however, without rollback, the system would fall into a crash state that would render all performed operations useless. Of course, CloRExPa rollbacks both malicious and non malicious changes.

CloRExPa knows the minimum number of steps that will (with an high probability) restore the monitored objects in a safe state because the last safe state in which a "dangerous" choice was made is known. It is important to note that the first rollback leads to the nearest "fork" in which a dangerous choice was made. In the unlikely event that such rollback fails, then a new one with an higher number of backward steps has to be executed.

The rollback was implemented using the "savevm" qemu monitor (i.e. the user console used to communicate with qemu) functionality. As many other virtualization software, qemu allows the creation of a complete snapshot of the entire virtual machine environment into a file. This "dump" of the virtual machine can be used to restore a previous state of the virtual machine from a file to a running vm. It is important to highlight that the larger the number of rollbacks, the larger the total amount of required disk space. This is another performance security trade-off that can be tuned depending on the level of security/performance we want to obtain. When rolling back to a snapshot, the system can be in an incorrect state to receive its next input. As such, transactions have to be replayed. However, this trade-off allows to avoid crashes that would leave the system in an incorrect and unstable state, where an unstable state is a state of the system that could lead to a failure. This kind of system states are represented in CloRExPa by "sick" nodes as depicted in Figure 3.1.

3.5 Evaluation

This section discusses the results of experiments aimed at evaluating CloRExPa effectiveness and performance in real-world scenarios. Tests have been conducted on quad core AMD CPUs equipped with 4GB RAM; Eucalyptus [92] version 2.0.2 was deployed on Fedora 14 (64 bit) hosts, whereas guest OSes were Ubuntu Server 10.04LTS and Fedora 14. Virtual Machines were given 1GB Ram and 1 virtual CPU each.

3.5.1 Effectiveness

CloRExPa can react to different kinds of malicious software but only if the “modus operandi” of such software is known. For example, if CloRExPa has been successfully attacked by a malicious software that makes use of “/dev/mem” to inject code, starting from that event, malicious software that makes use of such a kind of attack will be detected. This is due to the fact that, as depicted in section 3.3.3, using the action graph we can focus on actions executed by an attacker. For example, any attack that uses the /dev/mem to gain access to the kernel, adopts the same technique (sequence of actions) to gain root privileges. Collecting the actions of the first "/dev/mem" attack, CloRExPa will be able to recognize any other "/dev/mem" attack.

In order to evaluate CloRExPa effectiveness in detecting and preventing attacks, we exposed cloud VMs to the rootkits in Table 3.1. During an initial training phase, a set of known rootkits we built out of rootkit common knowledge [93] [94] were fed to the system. As such they were blocked before compromission could take place. Note that training rootkits were different from test rootkits. This shows the ability of CloRExPa to detect rootkits that behave similarly to other rootkits. This is generally due to the fact that CloRExPa focuses on single instruction instead of rootkit side effects.

We tested the rollback operation on a web application. We built a specific malware that had the effect of injecting malicious code inside a web application in order to make service unavailable and raise an error. When run under CloRExPa, the effect of the malware was reverted in a few seconds as the system was rolled back to the safe working state.

Some false positives have been encountered during the effectiveness tests. CloRExPa suffers from both false positives and false negatives in a real world scenario. Given definitions of Section 3.3, false positives and false negatives could be described as follows:

- False Positive: CloRExPa states that an action will lead to a faulty state even if the fault would be never reached.
- False Negative: CloRExPa states that an action is safe whereas that operation leads to a fault.

CloRExPa dynamically changes its graphs, as such false positive and false negative could happen as follows:

- False Positive: after a crash state prediction, new actions could lead to different paths and so to a safe state despite the prediction, as depicted in Figure 3.10.
- False Negative: after a safe state prediction, new actions could lead to a crash despite the prediction, as depicted in Figure 3.11.

It is important to highlight that false positive and false negative numbers decrease with time because the larger the graph is, the smaller the probability to create new nodes. CloRExPa is capable of handling both user-level and kernel-level execution tracing. As for resilience against software faults, kernel oopses and hard freezes were artificially triggered by fake module insertion and by inserting bugs inside the kernel source (e.g. altering the IDTR with invalid values). After the usual initial training on a VM, a warning at module insertion was issued and rollbacks to the closer safe state were automatically performed.

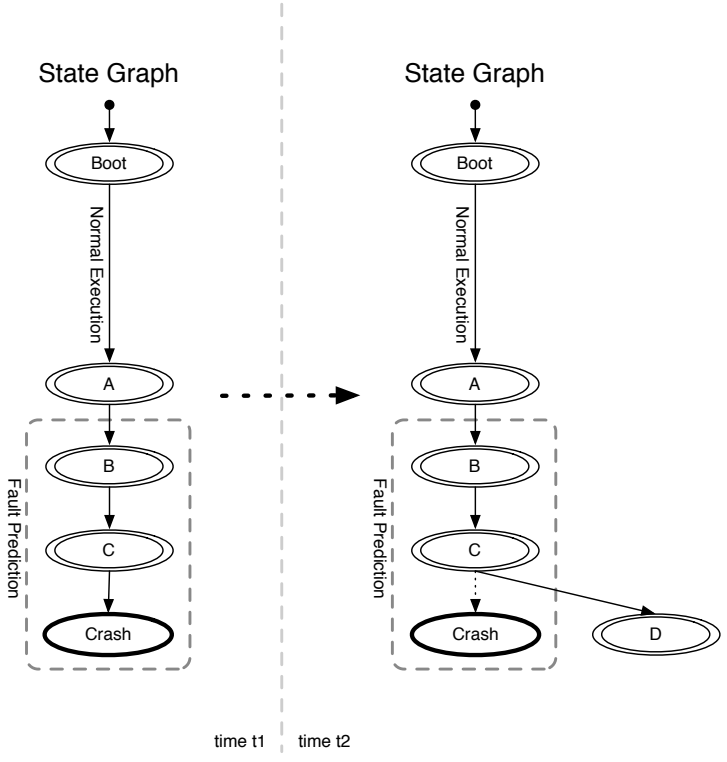

FIGURE 3.10: CloRExPa False Positive Detection

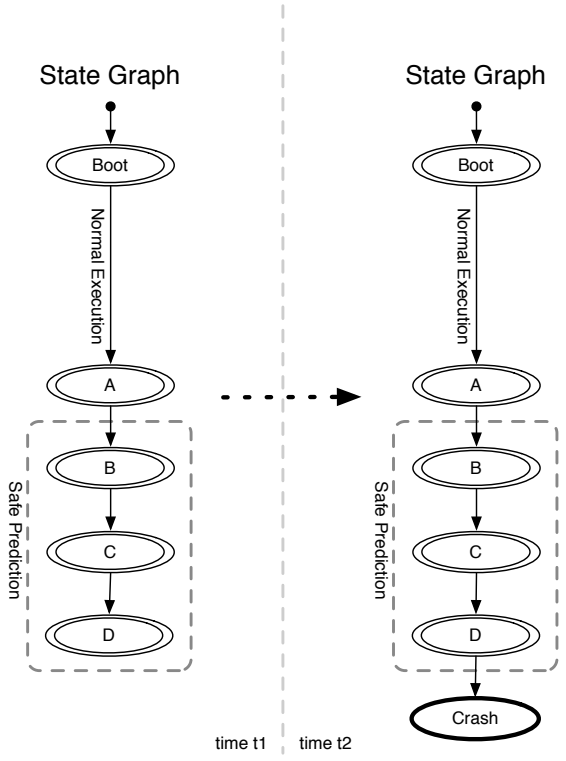

FIGURE 3.11: CloRExPa False Negative Detection

RootKit name	RootKit features	How detected
Adore-ng	kernel module	code checksum/inode
Enye	kernel module	kernel code checksum
Hideme	kernel code modification	code checksum
Phalanx	/dev/mem, syscall_table	syscall_table chk.
Sebek	kernel module, syscall_table	syscall_table chk

TABLE 3.1: Rootkit behavior and detection techniques

3.5.2 Performance

In order to evaluate the trade-off between the level of safety and security that is enforced, and the level of system performance, four different tracing approaches have been adopted for each benchmark: (1) *single layer*, where the model takes care of managing a single state graph; (2) *two layers*, where the model takes care of managing both a state and an action graphs; (3) *four layers*, where the model takes care of managing two state and two action graphs; (4) *six layers*, where the model takes care of managing three state and three action graphs.

Results always represent performance (higher is always better) and are shown in groups composed of four bars each. Columns represent performance relative to the same test executed on a single VM without CloRExPa, which is always 1. In case of time results the value in the graph

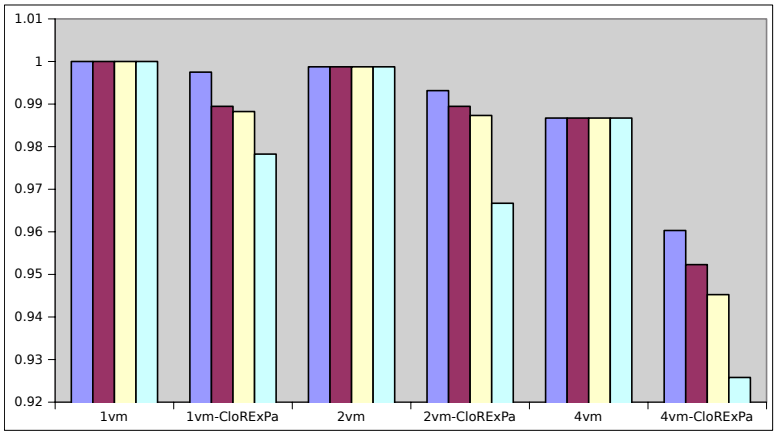

FIGURE 3.12: CloRExPa security performance trade-off on Lame on VM execution times in seconds - T1

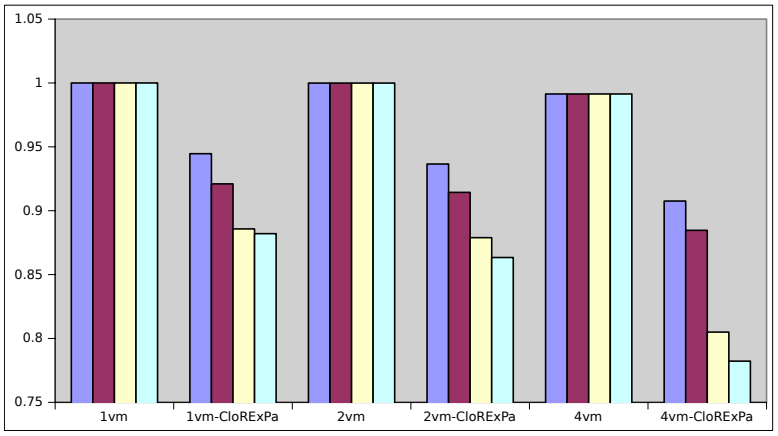

FIGURE 3.13: CloRExPa security performance trade-off of kernel compiling on VMs - T2

was obtained by dividing the reference value by the collected value. In case of number of events the value in the graph was obtained by dividing the collected value by the reference value.

Among the same group of four bars the only varying parameter is the tracing approach (from the single layer to the six layers—left to right) Among the six bar groups the varying parameter is the scenario (1 VM running with CloRExPa, 1 VM running without CloRExPa, 2 VM running with CloRExPa, 2 VM running without CloRExPa, etc). Any columns can be compared to any other.

T1 CPU stress tests: the Lame audio software is used to test CPU performance when encoding WAV files to the mp3 format.

T2 Mixed stress tests: the standard kernel compilation test is used to measure both CPU and filesystem performance.

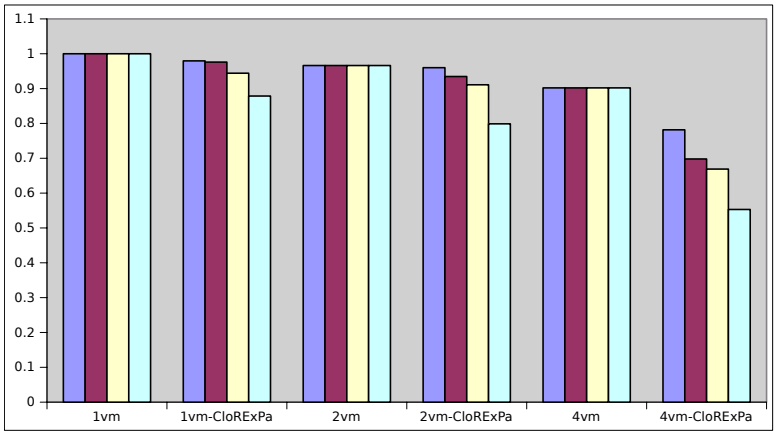

FIGURE 3.14: CloRExPa security performance trade-off on VM disk I/O performance - T3

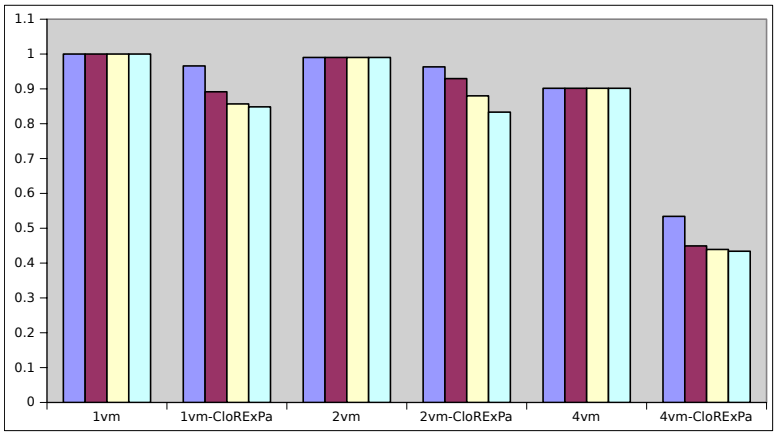

FIGURE 3.15: CloRExPa security performance trade-off on Apache performance (by number of served requests) - T4

T3 disk I/O stress tests: the Phoronix [95] iozone disk test is used to test raw read/write disk performance.

T4 network I/O stress tests: the Apache benchmark web server test is used to test network I/O performance.

The results of the CPU-bound tests are reported in Figure 3.12. Values show that the introduced overhead is quite small (max 5% performance loss due to CloRExPa) and directly proportional to the number of modeled/monitored objects. The overall CloRExPa-induced overhead increases (i.e. the performance loss is greater) when a larger number of VMs is active on the host at the same time (as in most real-world scenarios). This is due to the fact that VM event interception and model update tasks are taken care by the inactive cores when the host does not run at full load. Overall this is a very good performance result as shows that the CloRExPa system is deployable and efficient for CPU-intensive cloud workloads.

The results of the mixed workload tests are reported in Figure 3.13. Values show that the overhead introduced by CloRExPa when compiling a kernel source in the guest VMs is slightly larger here (at most 12% on an unloaded host and at most 22% on a fully loaded host machine). The performance loss is higher for the four and six layers model for a fully loaded host machine. A possible explanation is that this test causes an increased interaction with the filesystem and as such triggers more interception events and potentially model updates that are particularly evident when the overall host machine load is higher.

The results of the disk I/O tests are reported in Figure 3.14. The overhead introduced by CloRExPa is higher here, as expected from the open() microbenchmark above. A maximum of 38% performance loss (i.e. execution time increase) is visible in the last column on the right, that has to be compared with the fifth column group in Figure 3.14. Filesystem I/O is one area that affects CloRExPa guest performance in that it triggers much update activity on the VM model embedded in the action and scenario graphs. This performance penalty is being further investigated, as it can be potentially improved by adopting a more relaxed check that a priori excludes some less relevant filesystem activity. As for the above results, the good news is that the overhead is still proportional to the degree of protection/model accuracy. As seen above, this performance loss can also be reduced to less than 10% even at full load at the expense of model accuracy. Alternatively, file server machines load can be reduced (and CloRExPa impact) by deploying a reduced number of VMs on the same guest host.

Another set of results show the network service I/O performance with and without CloRExPa on the same set of deployment scenarios, and are reported in Figure 3.15.

The overhead introduced by CloRExPa when maximum model accuracy is deployed, is also quite high here, with a maximum performance loss of 50%. This is a lower bound value that has been obtained on a fully loaded host machine. Performance results on a more lightly-loaded (2 VM) host machine appear much better as CloRExPa introduces less than 17% penalty. Again, the overhead is proportional to the degree of protection/model accuracy and gives much better results if the machine is not fully loaded.

These first results are interesting, and encourage us to perform further investigation aimed at reducing performance overhead, especially regarding I/O interactions. As such real-world CloRExPa performance can be better than what appears here. Nevertheless, the impact on performance has been shown to be almost proportional to the degree of required protection. Hence, the system can be tuned in order to achieve an intended performance-prevention trade-off. The

CloRExPa system tunability allows the cloud provider to deploy as much protection and reliability improvements as the cloud client or service client require. The added resilience and protection can be sold as an additional service to the cloud user or the cloud service provider. This could convince companies to enter the cloud market, by paying a small tunable additional cost to security and resilience.

3.6 Conclusion

In this chapter we proposed an effective solution (CloRExPa) for the protection of VMs in a cloud computing environment. CloRExPa enjoys a few unique features: it allows to model VM activity and to trace guest alterations due to software faults, attacks, and environment changes. In particular, CloRExPa leverages execution path analysis on scenario graphs (state and action graphs) to prevent guests from reaching faulty or insecure states. Compared to other solutions, our approach solves fundamental issues such as: unobservability, unfaithful representation, and memory occupancy.

Performance tests on an actual Eucalyptus cloud test bed show the trade-off between the level of safety and security, and the level of system performance. Our thorough analysis shows that the impact on VM performance is almost negligible for CPU-bound workloads while being higher (but completely sustainable) on I/O and mixed workloads. However, such performance penalty can be largely tuned in order to meet performance requirements. Further, the monitoring depth as well as the number of monitored objects is fully customizable. As such, cloud management can effectively benefit from CloRExPa adoption while at the same time maintaining a high efficiency and cost-effectiveness of cloud solutions.

4 Cheating Resilience via LP Modeling and behavior Evaluation

Outsourced computing is increasingly popular thanks to the effectiveness and convenience of cloud computing *-as-a-Service offerings. However, cloud nodes can potentially misbehave in order to save resources. As such, some guarantee over the correctness and availability of results is needed. Exploiting the redundancy of cloud nodes can be of help, even though smart cheating strategies render the detection and correction of fake results much harder to achieve in practice.

In this chapter [1], we analyze the above issues and provide a solution for a specific problem that, nevertheless, is quite representative for a generic class of problems in the above setting: computing a vectorial function over a set of nodes. In particular, we introduce *CheR* (for Cheating Resilience), a novel approach based upon modelling the assignment of input elements to cloud nodes as a linear integer programming problem aimed at minimizing cost while being resilient against misbehaving nodes. Later, we discuss *AntiCheetah*, a novel autonomic multi-round approach performing the assignment of input elements to cloud nodes as an autonomic, self-configuring and self-optimizing cloud system. *AntiCheetah* is resilient against misbehaving nodes, and it is effective even in worst-case scenarios and against smart cheaters that behave according to complex strategies. We discuss benefits and pitfalls of the *AntiCheetah* approach in different scenarios. Preliminary experimental results over a custom-built, scalable, and flexible simulator (*SofA*) show the quality and viability of our solution.

[1] Part of this chapter appeared in [4] and [5]

4.1 Introduction

The Internet has paved the way to large scale distributed computing. The trend towards outsourced computing is culminated with the *Computing-as-a-Service* model, presently offered at different layers by cloud providers. In particular, the trend towards rendering application software available as-a-service (SaaS) on the cloud is increasingly successful for a number of reasons, in particular tied to licensing and management costs. As an example Matlab on cloud [96], Mathematica on Amazon [97], and in general High Performance Computing as-a-Service [98, 99], allow system administrators to avoid the setup and management costs due to the creation and software configuration of computing nodes.

Computing-as-a-Service is a form of totally-outsourced computing offered at different layers (IaaS, PaaS, SaaS) by many alternative cloud providers (Amazon, Microsoft and Google, among the others). Software-as-a-Service (SaaS), in particular, is increasingly widespread thanks to reduced licensing and management costs. Clouds offer cheap and powerful pay-as-you-go resources where splitting and offloading computation of parallel algorithms is feasible and convenient. However, remote computing nodes have historically been proven to misbehave, especially if they are rented with a pay-per-use approach [100]. In particular, remote computing nodes can save their energy and space resources by faking computation (i.e. pretending to compute) and returning erroneous results. One possible example is returning a random result instead of calculating a computationally-intensive function.

The problem of providing some forms of assurance over outsourced computation is not novel, and several efforts have been devoted to enforce some sort of control over the correctness and on the timeliness of returned results. The naïve solution that simply replicates the same computation on different sets of nodes is not satisfactory. The novelty of the problem when contextualized in the cloud scenario is that it is now economically feasible to dynamically rent a large number of computing resources (possibly from heterogeneous sources) at the same time. Further, the cloud can now be seen as an intelligent ecosystem that can self-adjust and self-select the best possible nodes for the customer.

In this chapter, we provide several contributions as for the distribution of workload over (heterogeneous) cloud nodes. In particular, we first formalize the problem of computing a parallel

function over a set of nodes; later, we introduce *CheR* (for Cheating Resilience), a novel approach based upon modelling the assignment of input elements to cloud nodes as a linear integer programming problem aimed at minimizing cost while being resilient against misbehaving nodes. We present and discuss some experimental results showing the viability and quality of our proposal. In addition, we introduce and discuss *AntiCheetah*, a novel cloud-autonomic solution [101, 102] for efficient and reliable distributed computing. Our solution assumes rational adversaries whose objective is to reduce their computing effort. The goal of the proposed approach is to enable efficient reliable computation of parallel functions of a large number m of elements over n nodes by minimizing the cost of selected cloud resources and keeping the computing time below a given threshold. We also want to minimize management cost, i.e. the system has to self-configure and automatically adapt to configuration changes. Furthermore, and more importantly, we want to ensure that the output of the distributed computation is correct (within a reasonable confidence interval percentage) even when cheating nodes are present.

The chapter is organized as follows. Section 4.2 introduces the problem and a use case where the proposed framework can be leveraged. Section 4.5 introduces system and threat models and presents a taxonomy of adversaries. In addition, it details the proposed *AntiCheetah* framework. Section 4.6 discusses the features of the ad-hoc built simulation engine and provides a first set of results by discussing some real use-case examples. Section 4.7 surveys relevant related work. Finally, Section 4.8 draws conclusions and introduces some hints for future extensions.

4.2 Problem Statement

In this Section, a simple use case is described to clarify the main problem. Suppose a system administrator is required to rent some cloud resources to perform a computationally-intensive and embarrassingly parallel task over a large data set. Two main possible scenarios exist:

- Time Priority (PR_T): the sysadmin has to ensure that the computation ends reliably within a given time frame and she is willing to spend as little as possible.
- Budget Priority (PR_B): the sysadmin has a fixed maximum budget and she has to reliably compute the function by minimizing the required time.

In the following, we will specifically consider the first scenario, leaving the second one as further work. Hence, a system administrator has to compute a function, over a large input vector, using

a set of nodes chosen from available cloud nodes, some of which may be cheaters. From the administrator's point of view it is interesting to know what is the amount of cloud resources required to satisfy the above-mentioned requirements, i.e. correctness and cost-efficiency. In the following, the adversary is modeled by assuming a standard/average (low) percentage of cheaters, such as 5% of the total number of nodes. This assumption is realistic as shown in [103]. As a general problem, nodes are required to compute an embarrassingly parallel function f over an input vector of length m, i.e., the output is itself a vector of length m where the j-th element is $f(j)$. At present, the most cost-effective computing resources are found in the cloud. Hence, in our model we assume that there are n cloud nodes (indicated as *nodes*, or *VMs*, from now on) where each node n_i (VM_i) can compute a subset (possibly overlapping) of the input vector.

The main goal here is to guarantee *reliability*, *timeliness*, *cost-effectiveness* and *correctness* of the computed results. We model the adversary as a *static* cheater P, i.e. a node that always fakes its computations with a given probability. In our model we assume guaranteed message delivery and no network cheating or lost messages. For simplicity, but without loss of generality, we also assume zero communication overhead (as in other related works e.g. [100, 104]).

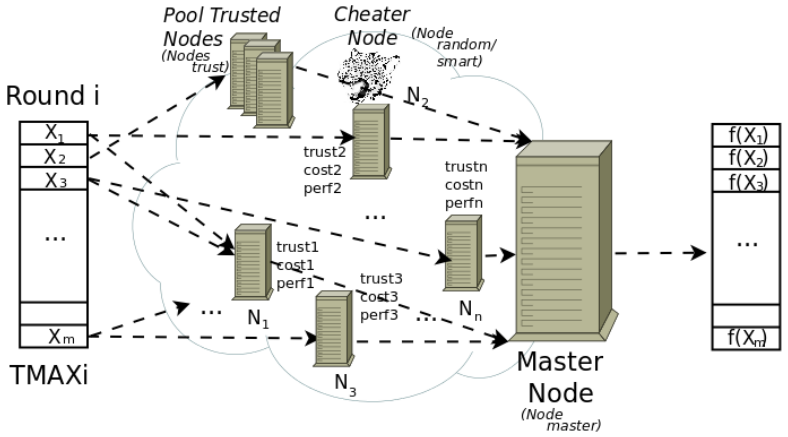

FIGURE 4.1: Use Case: Node N_2 is a Cheater (the Cheetah)

4.3 CheR: Problem Modeling

We generalize the use case discussed in the previous section, by assuming that a system administrator has to to compute a function f on a large vector X of length m. To this end, the manger has to send X to a cloud, which contains n cloud nodes (VMs), where each node VM_i has an

associated unitary cost per operation c_i and the time to perform an unitary operation is t_i[2]. Given that we assume some of the nodes are cheaters, where the percentage of cheaters (*CheaterRate*) is $\frac{k}{n}$, where k is the number of cheaters, the system administrator has to send multiple (possibly overlapping) subsets of X to the nodes to be confident that the output is correct, i.e. most of the results for the same input is correct. The confidence threshold, chosen by the system administrator to be reasonably ensured that the results are correct, is $DetConf$. The goal of the system administrator is to minimize the total cost of the operations, given a maximum time of computation T_{max} and given the fact that he/she wants all the results to be correct with an error less than $DetConf$. As an example, if $DetConf$ is equal to 0.01, we require that at most 1% of fake results are considered correct, i.e. they are not detected by the administrator as wrong results.

	VM_1	VM_2	VM_3	VM_4	VM_5
	c_1	c_2	c_3	c_4	c_5
	t_1	t_2	t_3	t_4	t_5

TABLE 4.1: Cost and Time Vectors

As an example, suppose we have five VMs. The cost and time vectors will store c_i and t_i for all nodes VM_i, as shown in Tab. 4.1. To better model the scenario we use a matrix $M^{n\times m}$, where $M_{i,j}$ means that the node n_i receives the element x_j to be computed on the function f. Indexes of the rows are coupled with the nodes, where $1 \leq i \leq n$, and indexes of the columns are associated with the elements of the vector, where $1 \leq j \leq m$. If we extend this example, by supposing that we have 7 elements, then we can model the assignment of workpiece x_j to node n_i on an $n \times m$ matrix as depicted in Tab. 4.2[3].

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
VM_1	1	0	1	0	0	1	0
VM_2	0	1	0	1	1	0	1
VM_3	1	0	0	1	0	0	1
VM_4	0	1	0	1	1	0	1
VM_5	0	1	1	0	1	1	0

TABLE 4.2: Matrix Assignment of the Example

The associated *total cost* C_i of the operations performed by the i-th node on the subset of the input received is:

[2] we assume that the application of f has the same cost and complexity for each input.
[3] the ordering of the performed operations does not take actual time into consideration.

$$C_i = \sum_{j=1}^{m} c_i \cdot M_{i,j} = c_i \cdot \sum_{j=1}^{m} M_{i,j} \tag{4.1}$$

Analogously, the *total time* T_i of the i-th node to perform the operations on the received elements is:

$$T_i = \sum_{j=1}^{m} t_i \cdot M_{i,j} = t_i \cdot \sum_{j=1}^{m} M_{i,j} \tag{4.2}$$

By taking into consideration the constraints imposed by the system administrator (as discussed in Sect.4.2), i.e. costs-effectiveness and timeless of the results, we can formulate the problem as an integer linear program, where the goal is to minimize the cost of the assignment of all the elements to the nodes, i.e.:

$$\text{Minimize} \sum_{i=1}^{n} \sum_{j=1}^{m} c_i \cdot M_{i,j} \tag{4.3}$$

Subject to the following time constraints:

$$\sum_{j=1}^{m} t_i \cdot M_{i,j} \leq T_{max} \quad \forall i \tag{4.4}$$

This value is a parameter of the model chosen by the system administrator so that results are returned in a timely fashion. As an example, the system administrator may set T_{max} in such a way that the slowest node (which is usually the cheapest one as well) cannot process more than a fraction of the input elements. Hence, each node can only process a predefined number of elements, according to its performance, so as to not exceed T_{max}. Hence, the previous equation can be rewritten as:

$$\sum_{j=1}^{m} M_{i,j} \leq Max(i) \quad \forall i \tag{4.5}$$

where $Max(i)$ is the maximum number of elements that the node i can process, considering its speed (time t_i to process each element) and T_{max}. Furthermore, in the model we have to consider that each input element can also be processed by a cheater node. To this end, we introduce the following equation:

$$\sum_{i=1}^{n} M_{i,j} \geq Repl(j) \quad \forall j \tag{4.6}$$

where $Repl(j)$ is the number of elements that has to be replicated for each input element j according to the confidence level $DetConf$. By replicating the computation, the chances of wrong results due to cheater nodes, are lower. Hence, the system administrator can verify that all results are correct within the given confidence level. $Repl(j)$ is an a-priori value that is computed out of $DetConf$ as follows: the number of requested replicas is computed using the hypergeometric distribution by considering that at least half of the replicated elements are given to, and processed by, cheater nodes. In this case, if at least half of the results are computed by cheaters, the system administrator would consider as correct their result. This is a conservative approach against a worst-case scenario were (i) all the cheaters cheat on their input and (ii) they return the same result.

Finally, in the model we have to consider the binary condition variables that are used to decide which of the input elements are given to which nodes:

$$\text{Binary } M_{i,j} \quad \forall i, j \tag{4.7}$$

All previous conditions and goals are summarized in Tab. 4.3. Once this LP model is solved, if a solution exists, an optimal assignment of input elements to cloud nodes is returned that satisfies all the system administrator-imposed requirements.

$$\begin{aligned}
\text{Minimize} \quad & \sum_{i=1}^{n}\sum_{j=1}^{m} c_i \cdot M_{i,j} & \\
\text{Subject to} \quad & \sum_{j=1}^{m} M_{ij} \leq Max(i) & , 1 \leq i \leq n \\
& \sum_{i=1}^{n} M_{ij} \geq Repl(j) & , 1 \leq j \leq m \\
\text{Binary} \quad & M_{i,j} & , 1 \leq i \leq n, 1 \leq j \leq m
\end{aligned}$$

Table 4.3: LP Model

4.4 CheR: Implementation and First Results

Starting from the above-described model, we have implemented *CheR* and validated it through a large number of simulations. In the current prototype, *CheR* is composed of a meta-program that creates linear programming problems in Cplex syntax using a range of different parameters as input. To solve such LP problems, *CheR* exploits a state-of-the-art solver such as GLPK [105]. In the following, we show and discuss some real-world examples to validate the proposed approach.

To study the behavior of the system with respect to scenario changes, the *CheR* meta-program takes as input the following parameters:

- the number of nodes n;
- the number of input elements m;
- the confidence level $DetConf$;
- the features of available nodes, such as time t_i required to process a single element and costs c_i;
- the time constraints for the termination of the reliable distributed computation (T_{max}): since every node has a corresponding cost, this means that every node n_i can process at most $Max(i)$ elements.

CheR firstly computes the number of required replicas to satisfy the given confidence level, and then it outputs the LP model that is later fed to the LP solver.

4.4.1 The Cloud Case

In this section, we introduce costs and time that are roughly representative of Amazon AWS [106], so that we can find a realistic solution for actual cloud service performance and associated costs. In order to model an Amazon-like cloud, 5 different node typologies (VMt stands for VM Type) are considered, ranging from Medium to XXXLarge according to the cost (which has been normalized) and time vectors of Tab. 4.4.

$NodeType$	VMt_1	VMt_2	VMt_3	VMt_4	VMt_5
$Cost$	48	99	249	500	1000
$Speed$	1000	500	250	100	50

TABLE 4.4: Cost and Time Vectors of the Cloud Case

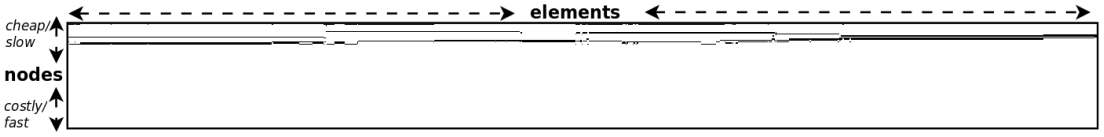

FIGURE 4.2: Amazon-like Matrix Assignment: Bitmap representing an actual assignment of elements (x-axis) to nodes (y-axis) for experiment/scenario E1 targeted at cost savings.

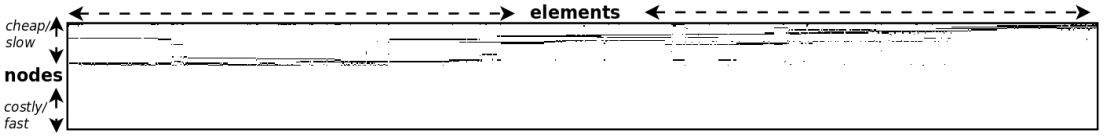

FIGURE 4.3: Amazon-like Matrix Assignment: Bitmap representing an actual assignment of elements (x-axis) to nodes (y-axis) for experiment/scenario E2 targeted at moderate cost savings.

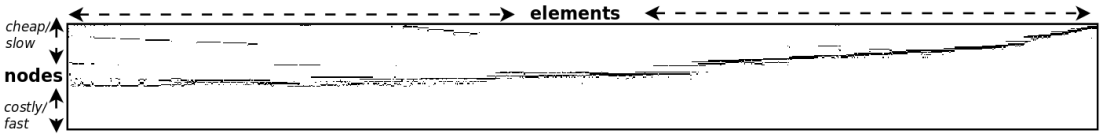

FIGURE 4.4: Amazon-like Matrix Assignment: Bitmap representing an actual assignment of elements (x-axis) to nodes (y-axis) for experiment/scenario E3 targeted at moderate time constraints.

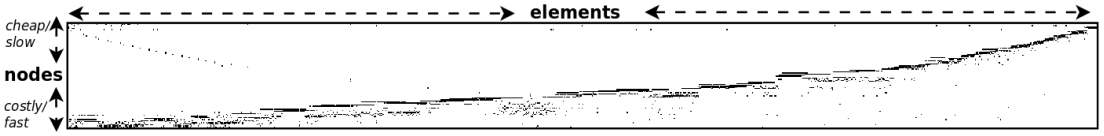

FIGURE 4.5: Amazon-like Matrix Assignment: Bitmap representing an actual assignment of elements (x-axis) to nodes (y-axis) for experiment/scenario E4 targeted at extremely tight time constraints.

In our tests, the number of nodes n is set to 100, where there are 20 nodes for each of the 5 typologies, there are 5 static cheaters P, i.e. 5% of the total nodes and 1,000 input elements m are considered.

Finally, the confidence level $DetConf$ is set to 0.01: considering the number of nodes, this level results in 3 replicas for each element. We have depicted four scenarios, where T_{max} is set so that nodes can process at most the number of elements shown in Tab. 4.5. These four scenarios depict four different alternatives: the first one targeted at extreme cost savings; the second one

with moderate balance requirements, with the bottom part (more costly, faster resources) is less used; the third scenario where tight time constraints are in place but where we also aim to cost containment; the fourth scenario, with extremely tight time constraints.

$Experiment$	VMt_1	VMt_2	VMt_3	VMt_4	VMt_5
$E1$	250	300	350	400	500
$E2$	100	150	200	250	300
$E3$	25	50	100	150	200
$E4$	12	16	25	50	50

TABLE 4.5: Maximum Number of Processed Elements for Experiments E1-E4

As regards the values chosen for the T_{max} time in experiments E1 to E4, the rationale is that we aimed at modeling real world time and budget constraints for an administrator. We have considered such limitations and put them in relation with standard computing capability of available VM instances. As regards experiment E1, T_{max} is set such that the result is obtained in less than 250 units of time; this implies that a slow VM will not be able to process more than 250 chunks. The same holds for the other experiments. It is worth noticing that the value of the T_{max} parameter is important in our model. This is chosen so that the allowed time is dependant on the administrator's requirements. In fact, in the kind of problems that we have analyzed, the global execution time is predictable given the computing capability of the heterogeneous nodes. As a matter of fact, we assume that the total cost is the real quantity to minimize. So the problem is to guarantee execution termination in a given time by efficiently using resources in order to minimize cost while remaining within a given timeframe. As such, the rationale behind the choice of the T_{max} value is as follows: T_{max} is set so that the slowest node can process at most a small fraction of the input elements. Otherwise one node could serially process all the items and the other obvious solution would be to minimize time only by assigning all chunks to the fastest nodes. These are borderline solutions that often are not realistic. Our approach is realistic and better fits the cloud model/approach.

The results of the matrix assignments are shown in figures 4.2, 4.3,4.4,4.5. In every figure, cheaper nodes are located at the top of the figure whereas increasingly more costly nodes follow towards the bottom; conversely, slowest nodes are located at the top of sub-figures whereas increasingly faster towards the bottom (results will be discussed in Sect. 4.4.3). Tests were performed on a computer featuring an Intel Core i5 CPU 750 @2.67GHz, 4 cores, 4GB of memory. We have performed several further tests to compute time and memory required by GLPK on different values of n and m. Figure 4.6 show the plotting of these values. Both the

execution time and memory requirements are $O(n \times m)$[4]. This is interesting as it shows the proposed approach can scale well to larger scenarios.

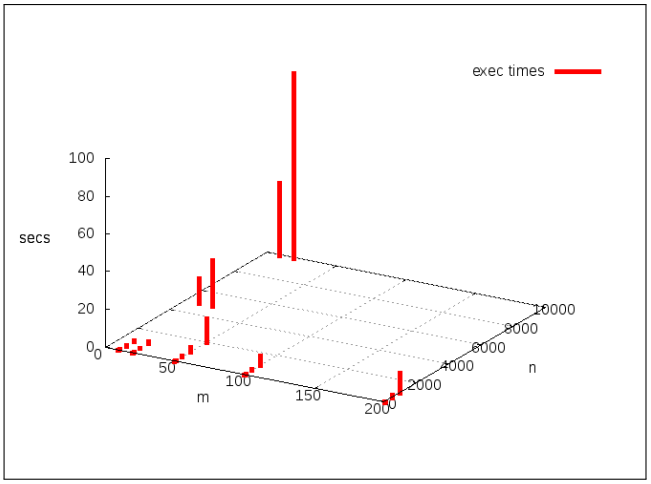

FIGURE 4.6: Execution Time of the GLPK LP Solver Software

4.4.2 Validation Tests

We have run several simulation tests that exploit the assignment matrix of E4. Such tests were aimed at discovering whether there was any wrong result that would be erroneously considered as correct by the collector. For each test, each experiment was repeated $10,000$ times by randomly choosing the set of k cheaters. In the end, for each test, we counted (i) the number of results that are wrong (ii) the number of tests that contain at least one wrong results. In each of these tests, we select a cheating probability for each static cheater P, i.e. how often a cheater returns a fake result. As a consequence, if a cheater returns wrong results more often, it will be easier to detect it. Conversely, if a smaller number of wrong results is returned, then it can be detected with more difficulty.

The number of performed simulated experiments for each test is $10,000$; the total number of processed input elements (with replicas) throughout all the experiments is $10,000,000$. Table 4.6 reports the results of the validations tests. The acronyms used in the Table are:

P cheating probability of static cheaters: how often a cheater cheats;

FNT percentage of false negative tests: ratio of failed tests without centralized control (at least one wrong results in an experiment);

[4]Given that the complexity of the LP model is the same.

FNE percentage of false negative elements: ratio of wrong results without centralized control (in all the experiments);

FNTC percentage of false negative tests (centralized scenario): ratio of failed tests with centralized control (at least one wrong results in an experiment).

FNEC percentage of false negative elements (centralized scenario): ratio of wrong results with centralized control (in all the experiments).

P	FNT	FNE (Avg)	FNTC	FNEC (Avg)
1	0.236%	0.006% (6.1)	0.00375%	0.0025% (2.445)
0,9	0.223%	0.0045% (4.5)	0.022%	0.0012% (1.17)
0.8	0.228%	0.0038% (3.8)	0.0075%	3.393E-4% (0.34)
0.75	0.226%	0.0036% (3.45)	0.0047%	2.144E-4% (0.21)
0.66	0.208%	0.0025% (2.5)	0.0011%	5.14E-5% (0.05)
0.6	0.2%	0.0021% (2.1)	6.0E-4%	1.267E-5% (0.0127)
0.5	0.19%	0.0015% (1.5)	0%	0% (0)
0.4	0.17%	9.436E-4% (0.94)	0%	0% (0)
0.33	0.16%	6.573E-4% (0.66)	0%	0% (0)
0.3	0.16%	5.358E-4% (0.54)	0%	0% (0)
0.25	0.14%	3.77E-4% (0.38)	0%	0% (0)
0.2	0.12%	2.5E-4% (0.25)	0%	0% (0)
0.1	0.05%	6.14E-5% (0.06)	0%	0% (0)

TABLE 4.6: Results of the Validation tests

4.4.3 CheR: Discussion

It is interesting to analyze the outcome of these first *CheR* tests. The workload distribution over the available computing nodes in different conditions is depicted in figures 4.2,4.3,4.4,4.5. The trend of the workload distribution with respect to shifting timing requirements is quite clear. As the maximum allowed execution time is compressed, the workload, including the replicated computations, seamlessly shifts towards the bottom lines, representing the more costly but faster nodes. Actually, this behavior is quite intuitive, but for large sizes of the problem matchmaking is nontrivial and as such an automated approach, such as the one presented in this chapter, is needed. Is it worth noticing that, even in the more tight time-constrained test, from the figure it is evident that the function mapping input elements to nodes strives to pair elements to less-costly nodes as soon as the time constraints have been satisfied.

The *CheR* approach is relevant with respect to state-of-the-art solutions. First of all, it combines LP with distributed on-the-fly/real-time resource allocation. Secondly, advanced real-world-like scenarios are considered. Finally, test results are analyzed in-depth and commented to show the viability of the proposed approach.

As an improvement to *CheR* we now introduce and discuss a natural evolution adopting an autonomic approach for effective cheating resilience.

4.5 AntiCheetah

AntiCheetah has the same goal of *CheR*, i.e. to guarantee *reliability*, *cost-effectiveness* and *timeliness* of results computed on possibly a large number of nodes. In the following subsections we first discuss the system model and different kinds of adversaries and then detail the proposed approach.

4.5.1 System Model

We assume again a generic scenario where k, over the total n nodes are rational adversaries (*cheaters*) [107] i.e. nodes that aim to minimize their resource consumption as well as the chances of being detected. We also assume that $(n - k)$ cloud servers are non-malicious and well-behaving and that among these non-malicious cloud nodes there is a small number of (completely) trusted nodes known by the administrator. Such nodes form a pool of trusted nodes ($nodes_{trust}$) that can be used on-demand. Notice that we assume that the cost of a $node_{trust}$ is much higher than other nodes as they are maintained on-premises by the administrator. As a consequence, such resources are rarely used in order to limit cost.

We assume that a central collector and comparator of results exists ($node_{master}$), which can realistically be a $node_{trust}$, which also ranks the nodes with respect to their measured past reliability, performance or cost.

Given these constraints, we need to make the computation redundant over the n nodes to detect possible anomalies—i.e. to protect the computation against cheaters. Our goal is to find an effective and efficient solution considering the following preconditions: (i) a fixed percentage of cheaters: we can realistically assume a low percentage [103], e.g. up to 5%; (ii) a given confidence threshold—where we are guaranteed that cheaters are detected (and as such their influence corrected) at the end of the algorithm execution; (iii) an optimal solution as regards the cost of renting the nodes—e.g. minimizing such a cost—; and, (iv) a given maximum time T_{max} to produce the results.

Multi-Round Assignment Matrix. We consider that the computation of the input vector is split in a number of consecutive *rounds*, through a *multi-round* approach, by considering smaller chunks of the input vector. As such, instead of performing the computation on all the m elements altogether, the sysadmin splits the input vector in disjoint pieces and assigns the elements to some of the nodes, through an *assignment matrix*. Then, at the end of each round, the sysadmin tries to discover if results are correct, within a given confidence level $conf$, and she tries to detect possible cheaters in order to dynamically update the assignment matrix by considering the reliability of nodes as well as their cost and performance. It is worth noting that a multi-round approach is especially suited when any of the following holds:

- the function f may have different priorities as regards requested cost, reliability of result, timeliness of the computation;
- cost, performance and reliability offered by the nodes may change over time to meet the need of a dynamic market.

It is worth stressing that the $node_{master}$ collecting the results of all nodes can compare the various values that are obtained. As such, if different results are returned for the same input a warning on a possible cheating is raised. To this end, a Result Comparison Vector (*RCV*) is used to keep track of the number of different answers for the same input and their occurrences. In fact, a *RCV* vector temporarily stores all the distinct returned values and the number of occurrences for each such value. If some nodes are suspected to be cheaters (i.e. if a returned result is a minority result in the *RCV*) the $node_{master}$ decreases the reputation for such nodes. In this scenario, the assignment matrix changes at every round, reflecting changes in node reputation, costs, time bounds and actual node performance. Hence, the assignment matrix is updated in order for the system to self-adjust the distribution of chunks and to offer the best actual price-performance ratio, given the current estimated reputation of the nodes. This autonomic approach is applied at every round.

Previous sections introduced *CheR* [4], an outsourced computing approach that is resilient against simple misbehaving nodes. *CheR* minimizes costs by modeling the assignment of input elements to cloud nodes as a linear integer programming problem. Differently from *CheR*, *AntiCheetah* adopts multi-round techniques to dynamically change the assignment matrix, detects smart cheaters and adopts a self-adjusting autonomic approach.

4.5.2 Threat Model

Cheaters that adopt a strategy for hiding/dissimulating their fake results are in general classified as *smart cheaters* [108]. In a general threat model, a cheaters' taxonomy comprises:

- **CH_STATIC**: a node that always fakes its computations;
- **CH_PROB_P**: a node i that returns fake results with probability p_i (or, in general, p);
- **CH_SMART_T+**: a node that does not cheat at first, but does it constantly after a given point in time/number of executed tasks;
- **CH_SMART_ADAPTIVE**: a (very) smart cheater that fakes results according to the number of requests required to serve per unit of time: when a large number of requests is served, a proportionally large percentage of fakes is returned, conversely, the percentage is lower when the node has not been used much during the last rounds;
- **CH_SMART_INTERM_T**: a node that cheats intermittently with period T;
- **CH_SMART_COALITION**: a node that exchanges information over past fake responses with other cheater nodes;
- **CH_SMART_SUPERVISOR**: a node that knows the past distribution of inputs (chunks) and of past fake responses for all other nodes.

A rational adversary aims to maximize the revenue coming from offered computations and at the same time she aims to minimize computing resource consumption. As an example, the adversary can be a battery-powered mobile cloud node that has endurance constraints [109]. Further, even a cloud service provider could pretend offering a large number of computing resources [110], while she is only offering a limited subset of the agreed resources, thus returning late or incorrect results. This might be the case when a better paying or larger customer competes for the same computing resources. For simplicity, but without loss of generality, we assume a fixed cheating percentage parameter (average or *ex abrubto* parameter), guaranteed message delivery and no network overhead, cheating or lost messages, as in other related work [100, 104].

4.5.3 The AntiCheetah approach

The proposed *AntiCheetah* solution is based upon an iterative approach that is aimed at: (i) maximizing the chances of detecting cheaters, (ii) finding an optimal resource allocation to optimize cost (considering replicas), (iii) receiving the results in a bounded amount of time. *AntiCheetah* considers three kinds of priorities for each round, i.e. *reliability*, *cost-efficiency* and *timeliness* of results. As such, nodes are ordered according to the current priority. In particular, if the priority is the *reliability* of the results, nodes will be ordered and chosen according to their trust level first, if the priority is *timeliness*, then nodes with highest performance will be chosen first; if, finally, *cost* is an issue, cheapest nodes will be chosen first.

Since some of the nodes may be cheaters, computation replicas may be requested for some of the input elements, in order to ensure that the required confidence level ($conf$) is met. In order to compute the number of replicas that are needed for each element, and to satisfy cost and time requirements, *AntiCheetah* has to consider node cost, performance and trust level. Hence, at each round, *AntiCheetah* assigns elements to nodes until the confidence level for the considered input is higher than $conf$. This confidence level is a function of the trust of the chosen nodes, of the global percentage of cheaters, and of the probability of having more cheaters than good nodes for the same element. At the end of the assignment, it may be the case that sometimes (using a random or context-based strategy) *AntiCheetah* distributes the same input to any node (potentially a cheater) and to a $node_{trust}$ in order to help detecting a cheater.

Iterative Approach. The iterative approach followed by *AntiCheetah* can be described as follows. Firstly, *AntiCheetah* computes the number of elements that each node can process in the current round based on node performance and on T_{MAX_i} (i.e. the maximum allowed processing time at round i). Secondly, to satisfy the required $conf$, (i.e. the reliability of the computation of each element) the number of requested replicas is iteratively computed for every element by considering the trust level of the chosen nodes. Furthermore, when choosing the nodes for assigning chunks, the following constraints are met by *AntiCheetah*:

1. a node cannot be asked to process the same element twice;
2. a node that has a very low reputation cannot receive an element (the node is in status *Inferno*);

3. the number of received elements for a node is lower than the maximum number of elements that the node can process in the given round time frame; such value is computed at the beginning of the round considering T_{MAX} and node performance;

4. the trust level of the global computation result, considering the trust level of all the replicated nodes and the global percentage of cheaters, is higher than $conf$.

After computing the assignment matrix among elements and nodes, a further assignment check is applied by *AntiCheetah* to detect cheaters. In fact, a small percentage of elements is given to $nodes_{trust}$: this assignment strategy is completely random in order to minimize the chances for a **CH_SMART_T+** ($node_{smart}$) to predict these checks. During this phase, if there is only one replica per element, the chances of assigning the input element also to a $node_{trust}$ is higher. This is required in order to avoid the cases where a $node_{smart}$ gains a very high reputation after some rounds by not cheating and returning correct results at first, and then succeeding in receiving one chunk just for itself in later rounds due to its high reputation. Notice that after this condition is achieved, the $node_{smart}$ can start cheating without being detected and its reputation will never be decreased. As previously said, the cost of a $node_{trust}$ is much higher than other cloud nodes and, hence, the percentage of elements given to $nodes_{trust}$ is very low.

At the end of every round, chosen nodes in the assignment matrix return results to the $node_{master}$. After the computation, this node first verifies if there are draws in replicated results for the same input, in cases where no $node_{trust}$ has been used. In this case, a further replica for the considered element is given to an arbiter node, which is chosen with the same strategy used to previously assign elements to nodes. Supposedly, this further computation would affect overall computing time. In reality, this happens quite rarely and it may reduce both the cost and time of the computation by requiring a further replica only at later stages.

After this pre-filtering step, the $node_{master}$ reaches a stable state: for every input element, either there are no draws or there is a $node_{trust}$ among the replicated nodes. Then, the $node_{master}$ analyzes the results for each input element and implements the following strategy:

- in case a $node_{trust}$ has been used, all diverging results are considered wrong and the reputation of the involved nodes is lowered by a high value ($trust_{high}$). The reputations of other nodes are raised by a low value ($trust_{low}$): these nodes could potentially be cheaters;

- otherwise, and if there are diverging results, the $node_{master}$ considers as good result the most frequent one (though it may happen that the most frequent result value is not the

correct one)[5]: in this case, these "winning" nodes have their trust level raised by a low value ($trust_{low}$) while the ones in minority have their trust level lowered by a medium value ($trust_{medium}$);

- it may also happen that all results from replicas are equal and the number of replicas is more than one; in this case, the $node_{master}$ raises the trust level of all the considered nodes by $trust_{low}$;

- at some fixed interval, nodes see their trust level raised by $trust_{verylow}$. This action is performed to possibly redeem good nodes erroneously classified as untrusted before (i.e. in status *Inferno*). In fact, in our model we assume faulty nodes may be erroneously classified as cheaters, i.e. in *AntiCheetah* good nodes may still return wrong results (due to S/W or H/W issues) but with a very low probability. This feature adds complexity to the cheating detection, since the $node_{master}$, when comparing the output of the computation and encountering diverging results, cannot be completely confident whether a node has cheated or a fault occurred, unless a $node_{trust}$ is used to replicate the computation.

At present, *AntiCheetah* focuses on cheaters **CH_PROB_P** ($nodes_{random}$) and **CH_SMART_T+** ($nodes_{smart}$), but the system is flexible enough to consider any kind of cheater. In this model we do not consider scenarios where we can distribute a subset of the input vector to *all* nodes as to quickly discover some cheaters. The rationale is tied to time requirements. In fact, a naïve cheating detection where many replicas are used at the beginning just to spot cheaters at an early stage would not work when smart cheaters are present. First of all, this would be inefficient, as it would waste many resources at the beginning. Secondly, once this simple detection strategy is known, smart cheaters would adapt and behave in a different way. This is the main rationale behind a more complex, adaptive autonomic strategy such as *AntiCheetah*. Autonomic computing in *AntiCheetah* also means continuously smart adapting to changes and to adversarial strategies.

Cheetah's Inferno. As regards point 2 above, i.e. not assigning elements to low-reputation nodes, this might end up in keeping bad nodes in a condition of *Inferno* for a very long time. The rationale for not discarding low-reputation nodes forever lies in the fact that there might be cases where their computation will be useful after these nodes have gained more reputation, such as in cost-effective computation. This is why node reputation slightly improves over time, albeit slowly as air bubbles move in oil, in order not to permanently exclude nodes from being

[5]Consider that in these scenarios we model the worst-case, i.e. where all the possible cheaters return the same value.

considered in the future. Of course, in these cases, since their past history of computation may reveal bad behavior, *AntiCheetah* usually prefers to replicate the same element to other cost-effective nodes.

We have also to consider that the chances of node misclassification (i.e. considering as a cheater a good node) are small but non negligible. As an example, if an element is given to three nodes, two of which are cheaters, the good node's result will probably be in minority, and as such, misclassified as cheater. Hence, some good nodes incorrectly classified as cheaters might "starve", especially in scenarios with a limited number of nodes, where preventing a good node from being used might actually negatively affect system convenience. An alternative strategy could be to replace low-trust nodes with other new ones: but in this case we have to consider that the chances of these new nodes being cheaters are the same (the global percentage of cheaters) and, in addition, their past history would be unknown.

4.6 Evaluating AntiCheetah

This section introduces the test environment for the experiments conducted on the proposed approach. Test results are shown and discussed in order to show benefits and limitations of *AntiCheetah*.

4.6.1 The SofA Simulator

Starting from the above-described model, we have implemented a flexible simulator, which is called *SofA* (Simulator of *AntiCheetah*), and we have validated it through a large number of simulations.

SofA is implemented in Java and it allows sysadmins to perform extensive test campaigns of different distributed computing approaches over a variety of scenarios. *SofA* simulates the behavior of a large number of nodes, although it does not currently implement neither transmission costs nor failures. At the end of every round, the simulator collects the information indicated in Table 4.7. Experimental data are then exported to be visualized by plotting programs such as Gnuplot.

Collected Value	Meaning
Nodes' status (cost, trust, time)	Describes the starting scenario
Input priority (cost, timeliness, reliability)	Customer requirements
Assignment matrix	For tracing assignments
Number of replicas for each input/round	Affects reliability and cost
Results (correct or fake/error)	Measures effectiveness
Total cost/time	Shows convenience
Percentage of wrong results (res_{wrong})	AntiCheetah's reliability

TABLE 4.7: Collected Statistics by AntiCheetah at the End of Each Round

4.6.2 Test Results

We have run these first tests by simulating four main scenarios:

(a) No Replica - (baseline case for comparison) without *AntiCheetah*, no replica for each element (each element is assigned to exactly one node) and with no $node_{master}$ cheating detection strategy;

(b) Trust Priority - *AntiCheetah* with trust priority;

(c) Cost Priority - *AntiCheetah* with cost priority;

(d) Time Priority - *AntiCheetah* with time priority.

The input parameters for the simulator are depicted in Table 4.8. The initial trust for all the nodes was set to a common value. Node cost was randomly chosen between a minimum and maximum cost, whereas node performance was set to be inversely proportional to the cost within a small variance. T_{MAX} for each round was randomly chosen between a minimum and maximum value for each round. The same set of nodes (with the same initial cost, trust, time, cheating probability) was used in all the tests. It is important to stress that we have performed worst-case cheating tests. In fact, we assume that different nodes that fake computation on the same input return identical outputs. This event is much more difficult in practice (unless cheating coalitions are considered), as the results codomain can be quite large. As a consequence, our assumption renders cheating detection much harder in general but stresses the effectiveness of the proposed approach. It is worth noting that in all the presented figures the evolution of the system over time is indicated in terms of performed number of rounds.

Parameter	Default Value
Number of rounds	2,000
Number of input elements per round	1,000
Number of nodes	500
Perc. of $nodes_{random}$ (of which $nodes_{smart}$)	0.05 (0.5)
Cheating probab. for $nodes_{random}$ and $nodes_{smart}$	0.5
Initial round for $nodes_{smart}$ to cheat	[0.2, 0.6] of total rounds
Percentage of elements given to $nodes_{trust}$	0.01; if the replicas is 1: 0.05
Fault probability	0.0001
Number of rounds to increase trust	number of rounds / 10
$Conf$	0.95
Node Cost	[10, . . . , 100]
Node Time	[5, . . . , 30]
Time Variance	3
Initial Node Trust	180
Maximum Node Trust	200
$Trust_{high}$	100
$Trust_{medium}$	200
$Trust_{low}$	2
$Trust_{verylow}$	1
T_{MAX}	[150, . . . , 200]

TABLE 4.8: Main Simulation Parameters Used in All the Described Tests

Figures 4.7,4.8,4.9,4.10 show the percentage of wrong results ($0 \leq res_{wrong} \leq 1$), i.e. those results that are considered correct by the $node_{master}$ that, instead, are due to results returned by cheaters being in majority (and that have cheated) or fault computations. Figure 4.7 shows results for the baseline case when no replication strategy is used. As we can clearly see from this figure, the number of res_{wrong} in very high, especially after $nodes_{smart}$ start returning fake results. On the contrary, as shown in Fig. 4.8, 4.9, 4.10, when applying the *AntiCheetah* approach, the percentage of res_{wrong} is considerably lower. In fact, these figures display the number of res_{wrong} in all the different *AntiCheetah* scenarios (trust, cost and performance priority). As we can see the percentage of res_{wrong} is much lower due to the strategy adopted by *AntiCheetah* to detect cheaters and to assign elements to nodes.

Notice that also in these cases there is still a small percentage of res_{wrong} due to the fact that the *AntiCheetah* approach is statistical, i.e. it strives to achieve the required confidence level (the guarantee that cheaters are detected), but it cannot guarantee the total absence of errors. In particular, in the first rounds res_{wrong} are caused by choosing $nodes_{random}$ first. On the contrary,

$nodes_{smart}$ can fake their computation more easily in later rounds by making their reputation increase slowly in the first rounds.

The total cost for round, i.e. the sum of the cost of renting each node for each element (considering the replicas), is shown in fig. 4.11,4.12,4.13,4.14, whereas the total time for round, i.e. the time required to perform all the computations, is shown in fig. 4.15, 4.16, 4.17, 4.18,. It is clear from these figures that the total cost is lower when the priority is cost and, conversely, the total time is shorter when the priority is time. Obviously, in the case where no replicas are used, the cost and time is lower but there is no strategy to detect cheaters and there are several res_{wrong} that go undetected.

To analyze the convenience of adopting *AntiCheetah*, we introduce a simple index C_{rct} (*rct* standing for Reliability-Cost-Time), which approximates the convenience of the approach in terms of a trade-off between error rate, cost, and time:

$$C_{rct} = \frac{\alpha}{1 + res_{wrong}} + \frac{(1-\alpha)}{2 \cdot (1 + uniCost)} + \frac{(1-\alpha)}{2 \cdot (1 + uniTime)}$$

where α is a weight that indicates the relative importance of cost-time versus result reliability, $uniCost$ is the cost per input chunk and $uniTime$ is the time per input chunk. The rationale

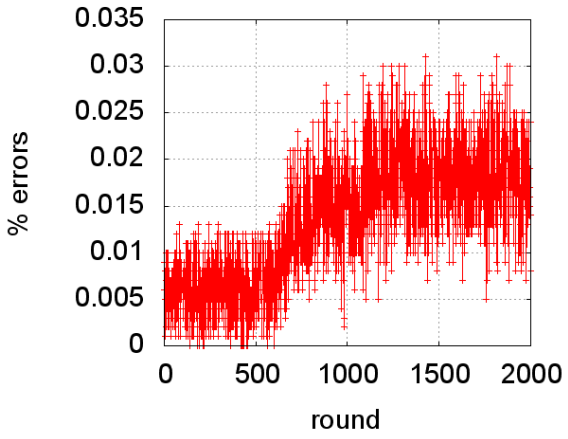

FIGURE 4.7: Percentage of Wrong Results: No Replica

behind this formula is that C_{rct}, where

$$min[(\alpha/2), (1-(\alpha/2)] < C_{rct} < 1$$

is a value that helps comparing the convenience of different *AntiCheetah* approaches for the same problem. In particular, C_{rct} tends to 0 if there is a high number of wrong results and/or a high unitary cost/time, which means that the convenience for the administrator is low. Conversely, if there is a low number of wrong results and/or the unitary cost/time is low, then the C_{rct} tends to 1.

Figure 4.19 shows the per-round C_{rct} for different approaches and values of α. We can see that, apart for the $\alpha = 0$ case, where the administrator has no interest in reliability, C_{rct} is very low for the no-replica case (due to the high rate of false negatives), whereas it is very high for all the introduced *AntiCheetah* strategies. The introduced smart approaches obtain similar results and the best possible results are given by the trust approach, i.e. when choosing first nodes with lower time to compute results[6].

One final remark concerns the trust level of the considered $nodes_{random}$ and $nodes_{smart}$. Figure 4.20 depicts the trend of the trust level of some cheater nodes in the trust priority scenario. It is clear from the figure that $nodes_{smart}$ (nodes labeled with $s-$) try at first to increase their trust level by not cheating but, as soon as they start cheating (and are detected), their trust level decreases considerably. On the contrary, $nodes_{random}$ (node labeled with $r-$) see their trust level already decreased in the first rounds. This fact positively affects the overall computation reliability since

[6]It is worth noting that the priority is only considered when ordering the nodes to whom replicas are assigned first. However, other parameters (number of replicas, minimum trust level) are still considered and they specify the required reliability of results, which is the main goal of *AntiCheetah*.

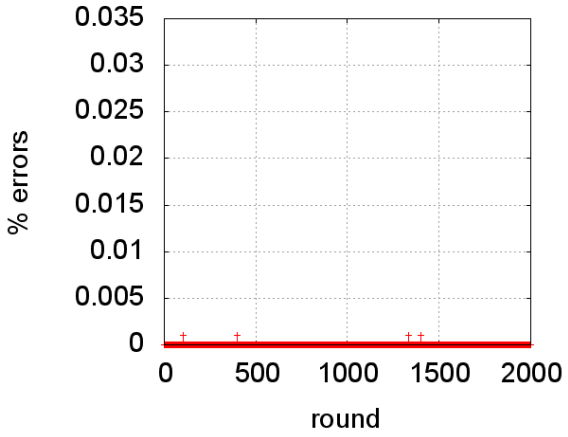

FIGURE 4.8: Percentage of Wrong Results: Trust Priority

either these nodes are not considered at all when assigning elements or their computation is shared with other more trusted nodes and, hence, their cheating can be detected more easily.

Overall it is worth noting that the proposed approach is convenient not only as a consequence of the best node choice based on reputation, cost and performance, but also given that such choice is performed autonomically by the *AntiCheetah*-enabled cloud. Hence, this approach is further economically convenient for a sysadmin as it automatically (and on-the-fly) adapts to nodes' behavior and learns from its past mistakes. As a consequence, issues such as cheating and faults are transparent to an *AntiCheetah*-enabled cloud user.

4.7 Related Work

The problem of guaranteed/verifiable outsourced computation is not novel, and various efforts have been devoted to obtaining some form of control or guarantee over the correctness [104] and on the timely availability of the results [111] of such outsourced computing. The novelty is that the cloud scenario renders now economically feasible to dynamically rent a large number of computing resources from heterogeneous sources at the same time. This is different than having many parallel processors on a single system as in that case they would be much less heterogeneous as regards performance, cost and behavior. This way the chances that a rational adversary or malicious cloud server coalitions can alter and or corrupt the final result without the user noticing it become very low. If the system administrator were able to fully control cloud VMs and ensure the integrity of the entire software stacks [112], rational or malicious behavior would not be possible, but set-up and management of cloud services would be more costly.

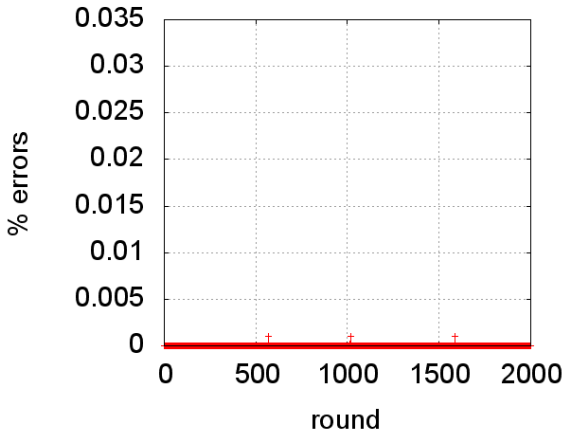

FIGURE 4.9: Percentage of Wrong Results: Cost Priority

Conversely, in this chapter we consider more realistic scenarios where the system administrator is not willing/cannot explicitly deal with bare VM configuration but uses pre-configured SaaS services.

Reliable Outsourced Computing. Golle [100] discusses the motivations of cheating by untrusted computing resources. He proposes security schemes that protect against this threat by discouraging cheating convenience. Golle also introduces a scheme that allows computing resources to prove they have done most of the work they were assigned with high probability. Das Sarma and Holzer [113] study the verification problem in distributed networks via a distributed algorithm. They give almost tight time lower bounds for distributed verification algorithms for many fundamental problems such as connectivity, spanning connected subgraph, and $s-t$ cut verification. Cheon et al. [114] study redundant work distribution techniques to exclude malicious participants. They aim at reducing work completion time by leveraging the characteristics of works and dynamic resources. They suggest a regional matchmaking technique to redistribute works to resources. In our present work we also make distinctions regarding speed and cost of available resources. Costa et al. [115] present a MapReduce algorithm that tolerate crash faults that uses twice the resources of the original Hadoop. In the SaaS model, clients get access virtual machines without having a direct access to the underlying hardware. Therefore, they cannot verify whether the provider gives the negotiated amount of resources or only a part of it. In particular, the assigned share of CPU time can be easily forged by the provider. The client could use a normal benchmark to verify the performance of the provided virtual machine but, since the Cloud provider owns the underlying infrastructure, the provider could also tamper with the benchmark execution. To detect this tampering, [116] proposes using proof-of-work functions and [117] to introduce a tamper-resistant benchmark to assess the performances of

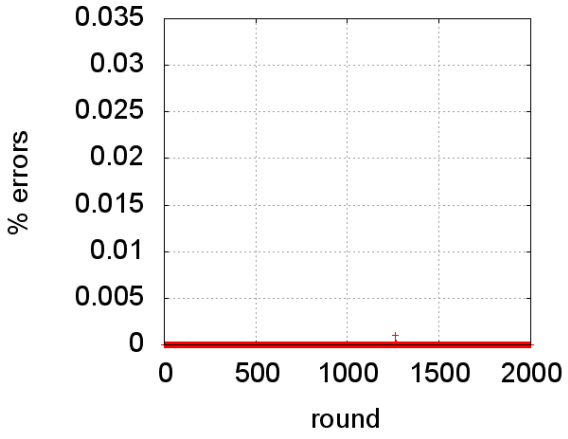

FIGURE 4.10: Percentage of Wrong Results: Time Priority

virtual machine instances. Proof-of-work functions are challenge response systems, where it is simple to generate a challenge and verify the result while solving the challenge is compute intensive. *CloudProof* [118] is a secure storage system for Cloud that allows customers to detect, among the others, violations of integrity and to prove the occurrence of these violations to a third party. Furthermore, also the Cloud provider can disprove false accusations made by clients. The proofs are based upon attestations, which are signed messaged that bind the requests made by the clients and the cloud itself to a certain state of the data. As regards Byzantine Fault Tolerance (BFT), some relevant work is due to Alvisi et al. [119], leveraging service replicas that optimistically process the request and reply immediately to the client. This model relies on the client to detect inconsistencies and correct them. This is the only similarity to our approach. However, work by Alvisi does not feature a proper cost model and does not consider cloud nodes and scenarios that are the main objectives of present work. Xin et al. [120] proposes using the remote attestation mechanism in Trusted Computing for cloud user's verification need. In this chapter, a property-based remote attestation method is designed through an attestation proxy, and users can validate the security property of the actual computing platform in the virtual cloud computing environment. *HOPE* [121] is a check-pointing and rollback recovery system for providing fault tolerance message-passing distributed systems. HOPE aims at scalability using an interesting group-based Hybrid Optimistic check-pointing and selective Pessimistic mEssage logging (HOPE) protocol. A system for compositional verification of asynchronous objects is proposed in [122]. Beimel et al. [123] study 1/p-secure protocols in the multi-party setting for general functionalities. They construct 1/p-secure protocols that are resilient against any number of corrupt parties provided that the number of parties is constant and the size of the range of the functionality is at most polynomial (in the security parameter n). They also show that when the

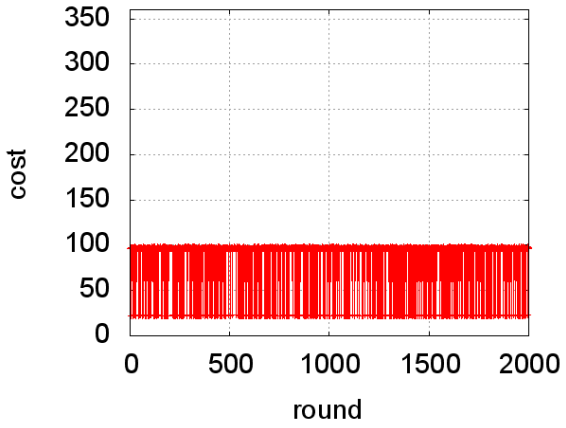

FIGURE 4.11: Total Cost: No Replica

number of parties is super-constant, 1/p-secure protocols are not possible when the size of the domain is polynomial.

Some contribution on scheduling management in clouds was given by Tang et al. [124]. They introduce a security-driven scheduling architecture that can dynamically measure the trust level of each node in the system by using differential equations and task priority ranking to estimate security tasks overheads. However, they focus on security against eavesdropping which is a different problem than the one discussed in this chapter. ScaleStar (SS) [125] is an approach to many-task workflow scheduling on clouds that takes into consideration the cost of resources. SS assigns the selected task to a virtual machine with higher comparative advantage which aims to balance execution time and monetary cost goals. SS does not address the problem of cheating nodes but it shows interesting price-performance considerations on scheduling. Parno et al. introduce Pinocchio [126] a Crypto-based system for efficient verification of remote computation. Pinocchio is an interesting specific approach using quadratic programs [127] for encoding computations and producing small-sized proof independently from the size of the computation.

Cloud Reputation Systems. A large number of reputation systems have been introduced in different fields to better rank and select items/people/restaurants/nodes. As for cloud nodes, Muralidharan et al. [128] introduced a reputation system for volunteer cloud nodes based on performance, number of crashes and result correctness. Even though the basic idea is interesting, our approach is more advanced in terms of job replication strategy (just two-fold replication for them) and node selection approach. Furthermore, we adopt a more effective multi-ranking approach that allows adjusting node selection to the user needs.

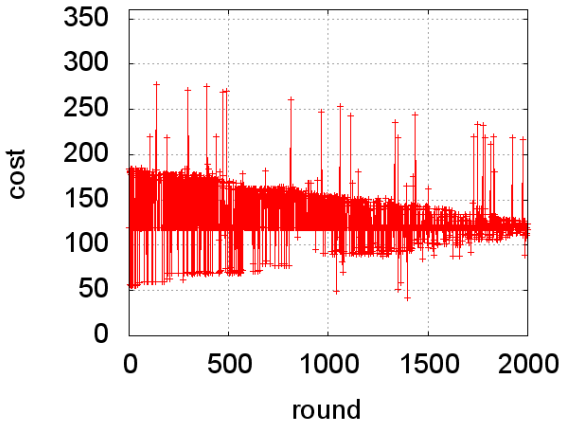

FIGURE 4.12: Total Cost: Trust Priority

Autonomic Cloud. Autonomic Computing techniques can provide better management of energy consumption, quality of service (QoS), and unpredictable system behaviors. In this context, Amoretti et al. [129] make use of the NAM Capacity Planner to show how to obtain optimal energy consumption under different working conditions and workloads. Their work shows how to minimize cost while maintaining functionality. Hariri [130] introduces BioRAC, a biologically-inspired resilient autonomic cloud using multi-level tunable redundancy techniques to increase attack and exploitation resilience in cloud computing. This work is interesting as their approach can also be of help against cheating. Casalicchio et al. [101] presents a heuristic solution for autonomic resource provisioning in the cloud. Unfortunately, they do not consider adversaries or cheaters and their model is aimed at maximizing only the cloud provider's revenue. Amoretti et al. [131] present a model for validating the convenience of cooperative cloud node strategies over selfish ones, where nodes do not help each other. Amoretti describes the architecture of the platform of autonomous clouds and the model implemented in a discrete-event simulator. Differently from Amoretti, in *AntiCheetah* we do not adopt a pre-built simulator for the sake of scalability and flexibility. Our approach is novel with respect to such state-of-the-art solutions. First of all, and to the best of our knowledge for the first time, *AntiCheetah* combines a reputation-based multi-round approach with autonomic distributed on-the-fly resource allocation. Secondly, advanced cheating scenarios are considered here. Finally, a wide range of experimental results has been presented, showing the convenience and effectiveness of the proposed approach.

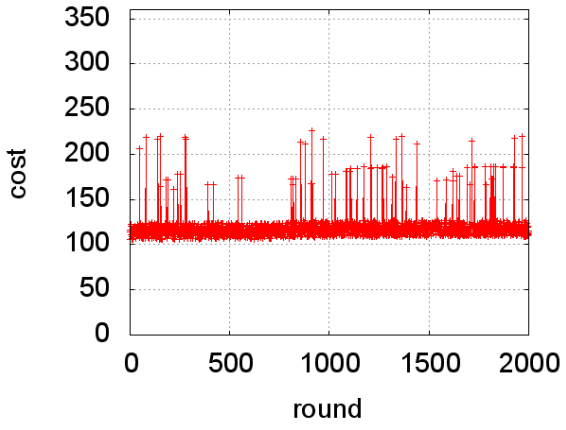

FIGURE 4.13: Total Cost: Cost Priority

4.8 Conclusion

In this chapter, we have addressed the vexed issue of enforcing integrity of distributed computations in the novel context of cloud computing. In particular, we have proposed *CheR*, a novel model for reliable execution of workload over a large number of heterogeneous (in cost and capabilities) computing nodes, where some of them can cheat according to a few identified models. *CheR* helps system administrators that are assessing whether and how to distribute computation over an heterogeneous cloud. In addition, *CheR* provides probabilistic assurance that the result of the distributed computations is not affected by misbehaving nodes, and that the incurred cost (as well as completion time) is minimized. Further, we have introduced and discussed *AntiCheetah*, an approach for reliable autonomic multiround execution of parallel workload over a large number of heterogeneous possibly cheating computing nodes. Experimental evidence, based on extensive simulations over a specially-built scalable simulator (*SofA*), shows the quality and viability of our proposal.

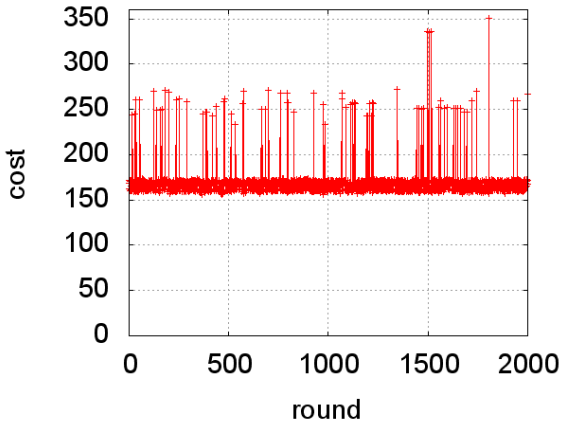

FIGURE 4.14: Total Cost: Time Priority

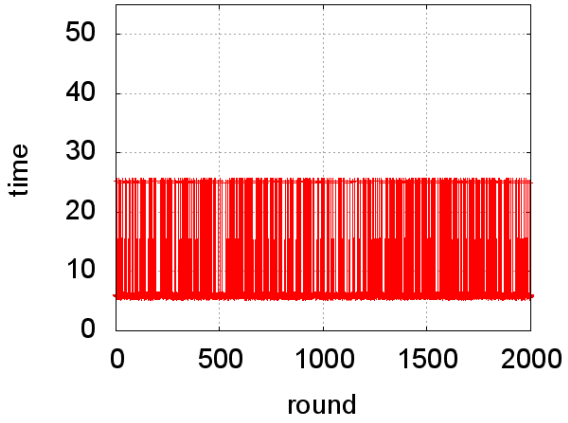

FIGURE 4.15: Total Time: No Replica

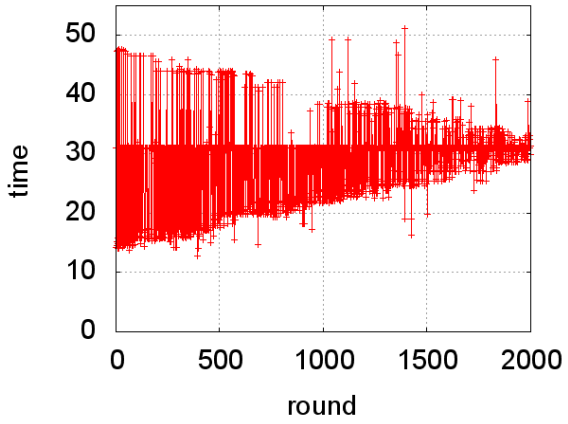

FIGURE 4.16: Total Time: Trust Priority

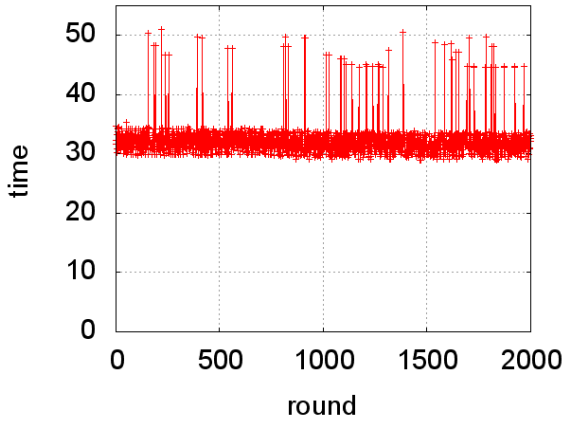

FIGURE 4.17: Total Time: Cost Priority

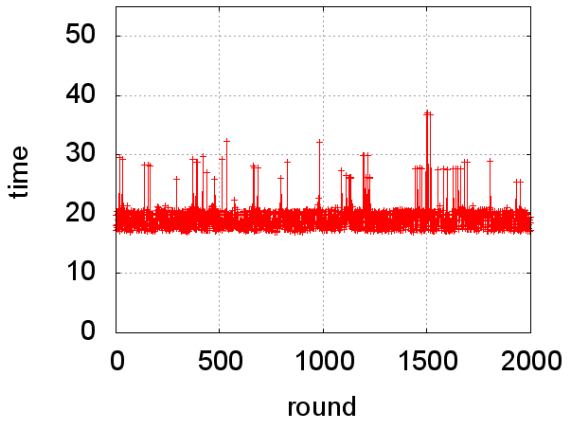

FIGURE 4.18: Total Time: Time Priority

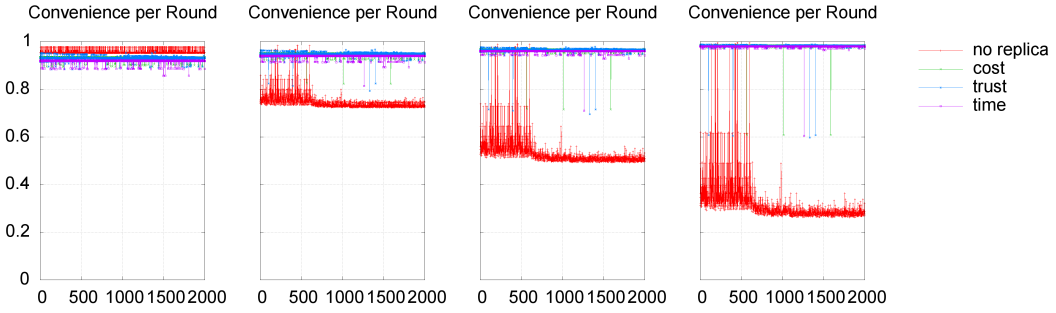

FIGURE 4.19: Per-round Convenience Index for Values (left to right) of α = 0, 0.25, 0.5, 0.75

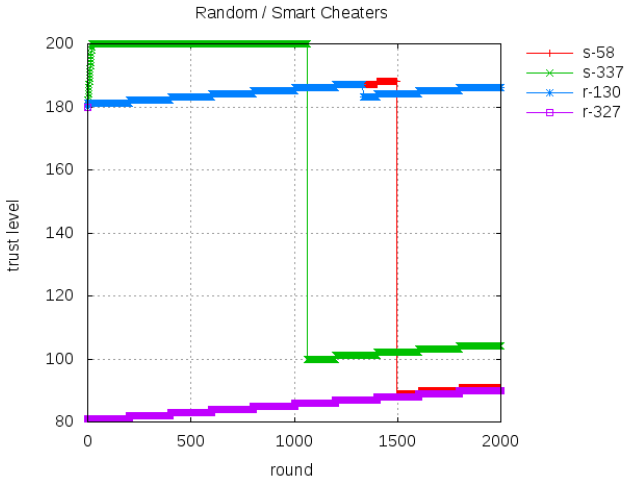

FIGURE 4.20: Trust Level of Random and Smart Cheaters in All the Rounds

5 Security Issues in GPU Cloud Architectures

Graphics Processing Units (GPUs) are deployed on most present servers, desktops, and even mobile platforms. A growing number of applications leverage the high parallelism offered by GPU architectures to speed-up general purpose computation. This phenomenon is called GPGPU computing (General Purpose GPU computing). This work reports on new security issues related to CUDA, the most widespread platform for GPGPU computing. In particular, details and proofs-of-concept are provided about a novel set of vulnerabilities CUDA architectures are subject to. We show [1] how such vulnerabilities can be exploited to cause severe information leakage. In particular, following (detailed) intuitions rooted on sound engineering security, experiments have been performed targeting the last two generations of CUDA devices: Fermi and Kepler. We discovered that these two families suffer from information leakage vulnerabilities; some of them are shared between the two architectures, while others are idiosyncratic of the Kepler architecture. As a case study, we experimentally show how to exploit one of these vulnerabilities on a GPU implementation of the AES encryption algorithm. Finally, we also suggest software patches and alternative approaches to tackle the presented vulnerabilities.

[1] Part of this chapter appears in [7]

5.1 Introduction

Graphics Processing Units (GPUs) were originally developed to accelerate 3D rendering. Nowadays, they are also used to speed-up general purpose computation. As an example, GPUs are used in scientific compute-intensive tasks and in computational finance operations [132]. Recently, GPUs have also been used to offload the CPU from security-sensitive tasks [133]. As such, several implementations of the most widespread cryptographic algorithms are now available on GPUs [134, 135].

The rapid success of GPU computing is also due to the space and power savings with respect to present CPU architectures. Indeed, GPUs contain hundreds of cores on a single silicon die and are more power-efficient than CPU cores [136]. Indeed, some of the most powerful computer clusters in the world are based on GPUs (e.g., [137]). Furthermore, Cloud providers offer specialized services designed to deliver the power of GPU processing in the Cloud [138] allowing multiple GPUs to be shared across different customers (i.e. GPU-as-a-Service). In addition, as single GPUs become more powerful, it is increasingly convenient to share even a single one among different tenants.

A different scenario is the one related to present PCs where the GPU is used for both graphical and computational tasks. As an example, modern browsers (e.g. Google Chrome) can use the GPU to render Web content; the same GPU can also be used to execute general purpose computations.

Despite their spread, a thorough analysis of the GPU environment from a security point of view is lacking. In fact, as it will be shown in the following, GPU and CPU architectures are quite different, therefore they are subject to different threats. Running a task on a GPU requires performing three main steps: i) a host application (i.e. a regular application running on the CPU) requests the execution of a *kernel* (i.e. a code to be run in parallel on the GPU); ii) the host application copies the input data from host memory onto GPU memory; and, iii) the host application launches the *kernel* and gets back the results. Data transfers from the host application to the GPU are performed via the proprietary device driver: once data enters GPU memory, the device driver takes control. Therefore, the isolation between different *kernels'* data is mainly a responsibility of the GPU device driver. Since GPU memory stores a copy of the process-specific data, a flaw in the isolation mechanisms on the GPU would undermine the isolation mechanisms of the Operating System (OS), thus causing an information leakage vulnerability.

Any kind of information leakage from security sensitive applications (e.g. encryption algorithms) would seriously hurt the success of the shared-GPU computing model, where the term shared-GPU indicates all those scenarios where a GPU resource is actually shared among different users (or tenants), whether it is on a local server, on a cluster machine or on a GPU cloud. The security implications on both GPU computing clusters and on remote GPU-as-a-Service offerings, such as those by companies like SoftLayer and Amazon, can be dramatic. It is worth mentioning that the findings of our work are not confined to the cloud environment; in present PCs (i.e. where the same GPU is used for both graphical and computational tasks), a malicious CUDA program would be able to *see* the content rendered by the GPU and thus violate the privacy of the user. Even worse, it would remain completely undetected since current antiviruses cannot analyze GPU binary code.

Due to its sensitiveness, one would expect the existence of secure and robust memory protection mechanisms on the GPU. Unfortunately, current GPUs and present device drivers are aimed at performance rather than security and isolation. As a consequence, GPU architectures are not robust enough when it comes to security [139, 140] and the adoption of GPUs actually introduces new threats that require specific considerations. Further, in view of the GPU virtualization approach offered by the upcoming hypervisors,information leakage risks would even increase.

5.1.1 Contribution

This chapter provides a number of contributions to the novel problem of secure computing on Graphics Processing Units, with a focus on the CUDA platform. In detail, leveraging perfectly standard GPU code, we were able to produce information leakage flaws; we were able to cause leakages by stressing the existing CUDA memory allocation and deallocation primitives, that lead to the discovery of three critical vulnerabilities. As for the first vulnerability described in this work, we were able to induce information leakage on GPU shared memory. Further, an information leakage vulnerability based on GPU global memory, and another one based on GPU register spilling over global memory were discovered and discussed.
As a case study we evaluated the impact of one of these leakages on a publicly available GPU implementation of the AES. In particular, we demonstrated that through the global memory vulnerability it is possible, for a not-legitimate user, to access both the plaintext and the encryption key. Finally, we also propose and discuss countermeasures and alternative approaches to fix the highlighted vulnerabilities.

5.1.2 Roadmap

This work is organized as follows: Section 5.2 discusses publicly available details of the CUDA architecture. Section 5.3 describes the rationales behind the discovered vulnerabilities and provides attack details. Section 5.4 introduces the experimental setup, gives implementation details on attacks and shows experimental results. Section 5.5 discusses a case study based on an AES CUDA implementation and analyzes the results. Section 5.6 introduces possible remedies. Section 5.7 surveys related work on CUDA and state-of-the-art results on information leakage. Finally, Section 5.8 provides some final considerations and directions for future work.

5.2 CUDA Architecture

CUDA is a parallel computing platform for NVIDIA GPUs. CUDA represents the latest evolution of GPU computing: old GPUs supported only specific fixed-function pipelines, whereas recent GPUs are increasingly flexible. In fact, General Purpose GPU computing (GPGPU) allows deploying massively parallel computing on COTS hardware where the GPU can be used as a "streaming co-processor". In this context, CUDA provides several facilities aimed at simplifying access to the GPU [141].

In particular, CUDA is composed of three parts: the Device Driver, the Runtime and, the Compilation Toolchain. The Device Driver handles the low-level interactions with the GPU (e.g. task scheduling); the Runtime handles the requests coming from CUDA applications (e.g. dynamic memory allocation) and routes such requests to the device driver. The Compilation Toolchain allows compiling CUDA applications from source code into intermediate and executable binary code.

A CUDA application is composed of host code (running on the CPU) and one or more *kernels* (running on the GPU). *Kernels* are special functions, that are executed *N* times in parallel by *N* different CUDA threads (i.e. threads running on the GPU). The number of CUDA threads that execute a kernel for a given call can be specified at launch time. It is possible to group threads together in one or more blocks, depending on the specific task that has to be performed.

Once a kernel is scheduled on the GPU, it always runs until completion and there is no clean way for the Operating System to stop its execution. Only the device driver can interrupt a running *kernel* by launching a specific interrupt.

The compilation of a CUDA application is performed in two steps: *(a)* the Compilation Toolchain

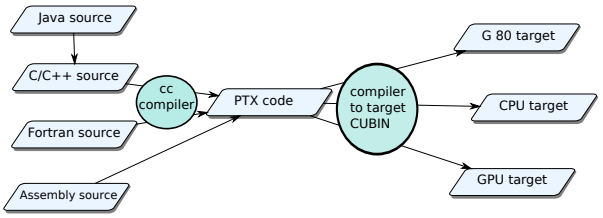

FIGURE 5.1: Main steps in compiling source code for CUDA devices.

transforms the *kernel* source code into an intermediate language called PTX [142] *(b)* the Device Driver translates the PTX into a binary code called CUBIN (CUda BINary), which is tailored to the specific GPU where it will be executed.

This approach allows specific code optimization to be tied to the actual GPU resources. An overview of the CUDA compilation process is depicted in Figure 5.1.

The CUDA architecture can be synthesized as follows: *(i)* a binary file comprising host and PTX [142] object code; *(ii)* the CUDA user-space closed source library (libcuda.so); *(iii)* the NVIDIA kernel-space closed source GPU driver (nvidia.ko); *(iv)* the hardware GPU with its interconnecting bus (PCI Express or PCIe), memory (Global, Shared, Local, Registers) and computing cores (organized in Blocks and Threads).

The Runtime offers a handle-based, imperative API: most objects are referenced by opaque handles that may be passed to functions to manipulate the objects. For our purposes, the *cudaContext* is the most important handle in the Runtime. CUDA applications use *cudaContexts* to cope with relevant tasks such as virtual memory management for both host and GPU memory. The Device Driver is responsible for allocating and managing resources belonging to a *cudaContext* as well as for freeing up these resources when a *cudaContext* is disposed of. It is worth noting that a *cudaContext* is automatically disposed of during the host process termination or, as an alternative, it can be cleaned up with a call to a specific function provided by the Runtime (i.e. *cuCtxDestroy*).

5.2.1 CUDA Memory Hierarchies

CUDA features different memory spaces and types (e.g. global memory, shared memory). All threads have access to the same global memory. Each CUDA thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. CUDA specifications do not describe what happens to shared memory

when a block completes its execution. There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are persistent across kernel launches by the same application [142]. Since this work focuses on global memory, shared memory and registers, they are detailed below.

5.2.1.1 Global Memory

Global memory is accessed via 32-, 64-, or 128-byte memory transactions. This is by far the largest type of memory available inside the GPU. When a *warp* (i.e. a group of threads, the minimum size of the data processed in SIMD fashion) executes an instruction that accesses global memory, it coalesces[143] the memory accesses of the threads within the *warp* into one or more memory transactions. This allows reducing memory access latency.

5.2.1.2 Shared Memory

Shared memory is faster than global memory and it is located near each processor core in order to have low-latency access (similarly to cache memory). Each multiprocessor (i.e. fixed group of cores) is equipped with the same amount of shared memory. The size of the shared memory is in the order of Kilobytes (e.g. 16KB or 64KB times the number of the available multiprocessors). Thanks to shared memory, threads belonging to the same block can efficiently cooperate by seamlessly sharing data. The information stored inside a shared memory bank can be accessed only by threads belonging to the same block. Each block can be scheduled onto one multiprocessor per time. As such, a thread can only access the shared memory available to a single multiprocessor. The CUDA developers guide [142] encourages coders to make use of this memory as much as possible. In particular, specific access patterns are to be followed to reach maximum throughput. Shared memory is split into equally-sized memory modules, called banks, that can be accessed simultaneously.

5.2.1.3 Registers

CUDA registers represent the fastest and smallest latency memory of GPUs. However, as we will show later, CUDA registers are prone to leakage vulnerabilities. The number of registers used by a kernel can have a significant impact on the number of resident *warps*: the fewer

registers a kernel uses, the more threads and thread blocks are likely to reside on a multiprocessor, improving performance. Therefore, the compiler uses heuristics to minimize register usage through *register spilling*. This mechanism places variables (that could have exceeded the number of available registers) in local memory.

5.2.2 Preliminary considerations

The CUDA programming model assumes that both the host and the device maintain their own *separate memory spaces* in DRAM, respectively host memory and device memory. Therefore, a program manages the global, constant, and texture memory spaces through calls to the CUDA Runtime. This includes device memory allocation and deallocation as well as data transfer between host and device memory. Such primitives implement some form of memory protection that is worth investigating. In fact, it would be interesting to explore the possibility of accessing these memory areas bypassing such primitives. Moreover, this would imply analyzing what kind of memory isolation is actually implemented. In particular, it would be interesting to investigate whether it is possible to obtain a specific global memory location by leveraging GPU allocation primitives. Further, when two different kernels A, B are being executed on the same GPU, it would be interesting to know what memory addresses can be accessed by kernel A or if A can read or write locations that have been allocated to B in global memory. What happens to memory once it is deallocated is undefined, and released memory is not guaranteed to be zeroed [144], [145].

Finally, note that the trend in memory hierarchy is to have a single unified address space between GPU and CPU (see also [142]). In fact, the CUDA unified virtual address space mode (Unified Virtual Addressing) puts both CPU and GPU execution in the same address space. This alleviates CUDA software from copying data structures between address spaces, but it can be an issue on the driver side since GPUs and CPUs can compete over the same memory resources. In fact, such a unified address space can allow potential information leakage.

5.3 Rationales of vulnerabilities research

The strategy adopted by IT companies to preserve trade secrets mostly consists in hiding architectural and implementation details of their products. Although this is considered the best strategy from a commercial point-of-view, for what concerns security this approach usually leads

to unexpected breaches [146]. The security-through-obscurity approach has been embraced by the graphics technology companies as well, and most implementation details about the CUDA architecture are not publicly available.

As detailed below, once a process invokes a CUDA routine the process has to completely trust the NVIDIA implementation of both the driver and the Runtime.
However, if we only consider the public information about the architecture, it is unclear if any of the security mechanisms that are usually enforced in the OS, are maintained inside the GPU. The only implementation details available via official sources just focus on performance. Indeed, NVIDIA suggests the usage of specific access patterns to global memory in order to achieve the highest throughput. In contrast, important implementation details about security features are simply omitted. As an example, there is no official information on whether memory is zeroed after releasing it [145]. Although this is not a problem for most scientific applications, it can cause severe security issues when sensitive data are involved in the computation. NVIDIA implemented memory isolation between different *cudaContext* within its closed source driver. Such a choice can introduce vulnerabilities, as explained in the following example. Suppose that an host process P_a has to perform some computation on a generic data structure S. The computation on S must be offloaded to the GPU for performance reasons. Hence, P_a allocates some host memory M_h^a to store S; then it reserves memory on the device M_d^a and copies M_h^a to M_d^a using the CUDA Runtime primitives. From this moment onwards, the access control on M_d^a is not managed by the host OS and CPU. It becomes exclusive responsibility of the NVIDIA GPU driver, i.e., the driver takes the place of the Operating System. Unfortunately, GPU drivers are not subject to the same thorough review to which the Operating Systems are subject.
To make things even worse, providing memory isolation in CUDA is probably far more complex than in traditional CPU architectures. As described in Section 5.2, CUDA threads may access data from multiple memory spaces during their execution. Although this separation allows to improve the performance of the application, it also increases the complexity of access control mechanisms and makes it prone to security breaches. A solution that preserves isolation in memory spaces like global memory, that in recent boards reaches the size of several Gigabytes, could be unsuitable for more constrained resources like shared memory or registers. Indeed, both shared memory and registers have peculiarities that rise the level of complexity for the memory isolation process. The shared memory, for example, is like a cache memory with the distinguishing feature that it is directly usable by the developers. This is in contrast with more traditional architectures such as *x86* where the behavior of the cache is transparent to software.

For what concerns registers, a feature that could taint memory isolation is that registers can be used to access global memory as well: in fact, a modern GPU feature (named *register spilling*) allows to map a large number of kernel variables onto a small number of registers. When the GPU runs out of hardware registers, it can transparently leverage global memory instead.

Multiple memory transfers across the PCIe bus for the global, constant, and texture memory spaces are costly. As such, they are made persistent across kernel launches by the same application. This implies that the NVIDIA driver stores application-related *state information* in its data structures. As a matter of fact, in case of interleaved execution of CUDA kernels belonging to different host processes, the driver should prevent any process P_j to perform unauthorized access to memory locations reserved to any other process P_i. Hence, this architecture raises questions as to whether it is possible for a process P_j to obtain unauthorized access to the GPU memory of any other process P_i. Our working hypothesis as for the strategy adopted by GPU manufacturers, is that they lean to trade-off security with performance. Indeed, one of the main objectives of the GPGPU framework is to speed-up computations in High Performance Computing, and not to provide context isolation. In such a scenario, the overhead introduced by a memory initialization routine run after each kernel would introduce an unacceptable overhead for the GPUs standard [147]. Indeed, as it will be proved and detailed in Section 5.4, these mechanisms have severe flaws and leak information.

It is worth mentioning that providing a thorough explaination of the root causes of the discovered vulnerabilities is out of the scope of this work. Our aim is to demonstrate that GPUs can leak information and to provide hints for countermeasures. Furthermore, even if GPUs are not designed with security in mind, they are increasingly used to access sensitive data. As such, it is important to investigate how GPU security can be improved to provide better security guarantees.

5.4 Experimental Results

The performed experiments aim at discovering whether, and under which conditions, a violation of the memory isolation mechanisms could occur—that is, whether the memory belonging to an honest process P_a can be accessed by a malicious process P_b.

	Leakage	**Preconditions**
Shared memory	Complete	P_a is running
Global memory	Complete	P_a has terminated and P_b allocates the same amount of memory as P_a
Registers	Partial	none

TABLE 5.1: Summary of the results of the experiments

This section describes the test campaign we set-up on GPU hardware to investigate the above mentioned issues. We performed the experiments on two different generations of CUDA-enabled devices (Fermi and Kepler) using a black-box approach. It is important to note that in our experiments we consider an adversary which is able to interact with the GPU only through legitimate invocations to CUDA Runtime.
In the following subsections we will detail three different leakage attacks targeted at different memory spaces. Each leakage has specific preconditions and characteristics. For each kind of leakage, we developed a C program making use of the standard CUDA Runtime Library. In just a single case we had to directly write PTX assembly code in order to obtain the desired behavior.

The rest of this section is organized as follows: the experimental testbed is described in detail in Subsection 5.4.1; in Subsection 5.4.2 the first and simplest leakage is discussed regarding shared memory; in Subsection 5.4.3 a potentially much more extended (in size) information leakage is detailed—while not leveraging shared memory—; finally the most complex and powerful information leakage is described in Subsection 5.4.4. It leverages registry usage and local memory.

5.4.1 Testbed Setup

Tests were performed on the Linux platform, as GPU clusters are mainly hosted on such OS.We performed the experiments using different CUDA HW/SW configurations.

The testbed features both COTS and professional-level CUDA hardware using production-level SDKs and was comprised of the HW/SW configurations described in Table 5.2. In order to verify whether the obtained information leakage was independent from the implementation of

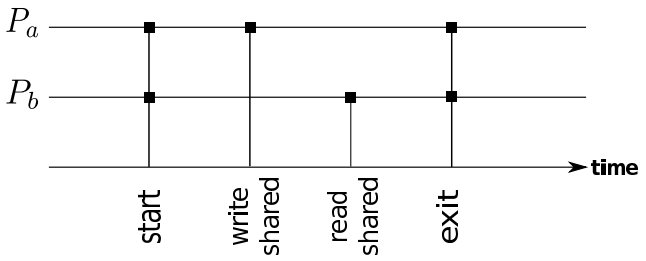

FIGURE 5.2: The schedule that causes the leakage on shared memory.

a specific GPU, and thus to make our experiments more general, two radically different configurations were chosen. On the one hand, a Tesla card that can be considered targeting the HPC sector; on the other hand, a GeForce card targeted at consumers and enthusiasts. The objective was not performance comparison but the analysis of possible leakages in different scenarios. The Tesla card implements the Fermi architecture whereas the GeForce card belongs to the newer Kepler family, i.e. the latest generation of NVIDIA GPUs. As such, the two GPUs differ with respect to the supported CUDA Capability (2.0 for the Fermi and 3.0 for the GeForce). In our experiments the compiling process took into consideration the differences between the target architectures. In Table 5.2 we report the specifications of the two GPUs. The reported size of shared memory and registers represent the amount of memory available for a single block (see [142]).

Each experiment was replicated on both configurations; in some cases we tuned some of the parameters to explicitly fit the GPU specifications (e.g. the size of shared memory).

5.4.2 Shared Memory Leakage

In this scenario the objective of the adversary is to read information stored in shared memory by another process. In order to do so the adversary uses regular Runtime API functions. Present CUDA Runtime allows each *cudaContext* to have exclusive access to the GPU. As a consequence, CUDA Runtime does not feature any preemption mechanism among different *cudaContexts*.

The Runtime keeps on accepting requests even when there is a kernel running on the GPU. Such requests are in fact queued and later served according to a FIFO scheduling policy. It is worth noting that the above mentioned requests can belong to different *cudaContexts*. As such, information can be leaked if no memory cleaning functionality is invoked, following a context-switch.

In fact, every time a malicious process is rescheduled on the GPU, it can potentially read the last-written data of the previous process that used the GPU. As such, the scheduling order affects which data is exposed. As an example, if P_a is performing subsequent rounds of an algorithm, the state of the data that can be read reflects the state reached by the algorithm itself.

The experiment to validate such hypothesis is set-up as follows: two different host-threads belonging to distinct processes are created; P_a being the honest process, and P_b the malicious one trying to sneak trough P_a memory. P_a executes K times a kernel that writes in shared memory ($K = 50$ in this test). In this experiment P_a copies a vector V_g [2]. Every element in V_g is of type *uint32_t* as defined in the header file *stdint.h*. The size of the vector is set equal to the size of the physical shared memory: 48KB for the Tesla C2050 and 16KB for the GeForce GT 640. The copy proceeds from global memory to shared memory. The host thread initializes V_g deterministically using sequential values (i.e. $V_g[i] = i$). P_b executes K invocations of a kernel that reads shared memory. In particular, P_b allocates a vector V_g and declares in shared memory a vector V_s with size equal to the shared memory size. Then data is copied from V_s to V_g.

With this particular sequence of operations (see Figure 5.2) P_b recovers exactly the same set of values written by P_a in shared memory during its execution. That is, a complete information leakage happens as regards the content of memory used by P_a. To obtain the information leakage, it is essential for the kernel of P_b to be scheduled on the GPU before the termination of host process P_a. In fact, we experimentally verified that shared memory is zeroed by the CUDA Runtime before the host process invokes the *exit()* function.
One of the parameters of the function developed for this experiment is the kernel block and grid size. In our tests we have adopted a number of blocks equal to the actual number of physical multiprocessors of the GPU. As regards the number of threads we have specified a size equal to the *warp*-size.

Quite surprisingly, the values captured by P_b appear exactly in the same order as they were written by P_a. Given that GPUs have a block of shared memory for each multiprocessor, one would expect that a different scheduling of such multiprocessors would lead to different orderings of the read values. To investigate this behavior, we made another test using an additional vector where the first thread of each thread-block writes the identifier of the processor where it is running. This info is contained in the special purpose register *smid* that the CUDA Instruction Set Architecture (ISA) is allowed to read. In fact, *smid* is a predefined, read-only special register

[2] V_g denotes any vector stored in global memory whereas V_s any vector stored in shared memory

GPU Model	Tesla C2050	GeForce GT 640
CUDA Driver	4.2	5.0
CUDA Runtime	4.2	4.2
CUDA Capability	2.0	3.0
GPU Architecture	Fermi GF100	Kepler GK107
Global Memory	5 GB	2 GB
Shared Memory per MP	48 KB	16 KB
Registers	32768	65536
Multiprocessors (MP)	14	2
Total Shared Memory	672KB	32KB

TABLE 5.2: The testbed used for the experiments

that returns the processor (SM) identifier on which a particular thread is executing. The SM identifier ranges from 0 to $nsmid - 1$, where *nsmid* represents the number of available processors. In order to read this information, we embedded the following instruction, written in PTX assembler, inside the kernel function:

```
[frame=single]
asm("mov.u32 %0, \%smid;" : "=r"(ret) );
```

The above instruction copies the unsigned 32-bit value of register *%smid* into the *ret* variable residing in global memory. Note that CUDA deterministically chooses the multiprocessors for the first *nsmid* blocks. As an example, for the Fermi card we obtained the multiprocessor ID sequence: 0, 4, 8, 12, 2, 6, 10, 13, 1, 5, 9, 3, 7, 11. This evidence explains the reason why the process P_b was able to read the values in the same order as they were written by process P_a. Due to the closed-source nature of CUDA, we cannot claim that the multiprocessor ID sequence would be always the same. Indeed, by varying some configuration parameters (e.g. the number of blocks or the number of threads), a different scheduling of multiprocessors could be triggered. However, this behavior would impact only on the order of the leaked values.

Take-away point: *GPU shared memory can leak information by leveraging interleaved access by different host programs.*

5.4.3 Global Memory Leakage

In this scenario the objective of the adversary is to read information stored in GPU global memory by another process. As in the previous scenario, we have two independent host processes,

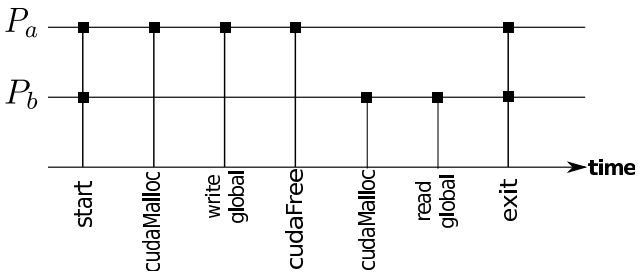

FIGURE 5.3: The schedule that causes the leakage on global memory.

namely P_a and P_b, representing the honest and the malicious process, respectively. The information leakage is due to the lack of memory-zeroing operations. In fact, as mentioned in Section 5.2, some of the resources used by a *cudaContext* are cleaned up only when the context is destroyed.

In this experiment (see Figure 5.3) P_a is executed first and allocates four vectors V_1, V_2, V_3, V_4 of D bytes each in GPU memory. The dynamic allocation on the GPU uses the *cudaMalloc()* primitive of the CUDA Runtime. The i-th elements of vectors V_1 and V_2 are initialized as follows:

$$V_1[i] = i; V_2[i] = D + i;$$

In this experiment, P_a invokes a kernel that copies V_1 and V_2 into V_3 and V_4, respectively. P_a then terminates and P_b gets scheduled. The correct synchronization between the two processes is maintained through Unix sockets. P_b allocates four vectors V_1, V_2, V_3, V_4 of size D bytes each, just as P_a did before.

The only difference is that P_b does not initialize neither V_1 nor V_2. P_b now runs the same kernel code that P_a executed before and copies V_3 and V_4 back in the host memory.

We verified that P_b obtains exactly the same content written before by P_a. This leakage is deterministic. Hence, we can conclude that the information leakage on the global memory is *full* (i.e. P_b retrieved all the data written by P_a in global memory during its execution).

We tried several values of D (starting from 4 KB up to the maximum allocable memory), always obtaining a full leakage. However, we noticed the leakage is full only if the malicious process allocates exactly the same amount of memory released by the honest one. This behavior could depend on the fact that the NVIDIA driver implements a modified buddy-system memory

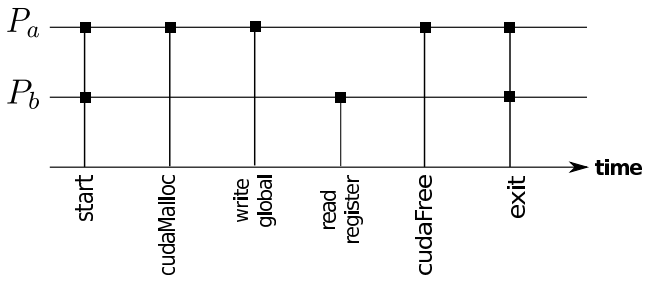

FIGURE 5.4: The schedule that causes the leakage on Registers. In this scenario P_b accesses the global memory without Runtime primitives.

```
__device__
void get_reg32bit(uint32_t *regs32) {

       # declaration of 8300 registers
       asm(".reg .u32 r<8300>;\n\t");
       # move the content of register r0 into
       # the position 0 of regs32[]
       asm("mov.u32 %0, r0;" : "=r"(regs32[0]));
       asm("mov.u32 %0, r1;" : "=r"(regs32[1]));
       ...
       asm("mov.u32 %0, r8191;" : "=r"(regs32[8191]));

}
```

FIGURE 5.5: A snippet of the code that allows to access global memory without *cudaMalloc*.

Information Leakage Amount		P_b **Register Space Size**		
		32KB	**64KB**	**128KB**
P_a **Allocated Memory**	**16MB**	32K	128K	256K
	32MB	64K	128K	256K
	64MB	64K	64K	128K
	128MB	64K	64K	256K

TABLE 5.3: Number of bytes leaked with two rounds of the register spilling exploit.

manager using different queues for memory requests of different size. Unfortunately, due to the closed-source nature of NVIDIA driver we were not able to verify this hypothesis.

Take-away point: *GPU global memory can easily leak information by exploiting the lack of adequate data allocation and data cleaning mechanisms.*

5.4.4 Register-Based Leakage

In this last described leakage, the objective of the adversary is to exploit registers to leak information. Again, we have two independent host processes, namely P_a and P_b, representing the honest and the malicious process, respectively. P_b exploits a feature called *register spilling* to

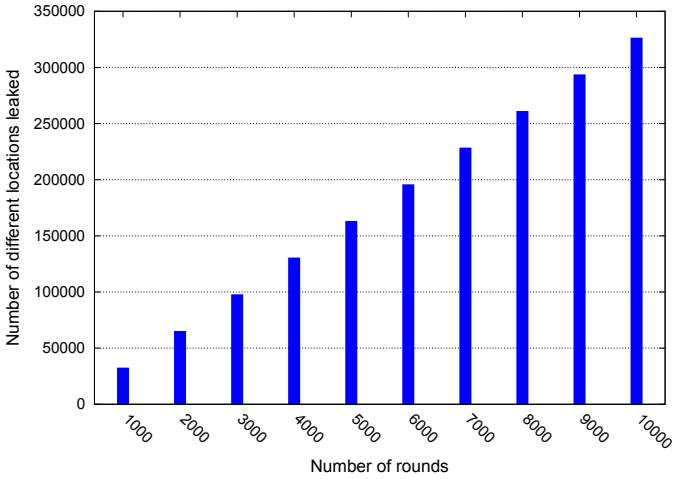

FIGURE 5.6: The number of different locations leaked depends on the number of rounds.

access the global memory reserved to P_a. *Register spilling* allows a process to reserve more registers then the number register available on the GPU. If a variable can not be assigned to a register, then it is placed in global memory. By using the PTX intermediate language (see Figure 5.1), we are able to exactly specify in a kernel how many registers it will actually need during execution. If the required number of registers exceeds those physically available on chip, the compiler (PTX to CUBIN) starts spilling registers and actually using global memory instead. The number of available registers per each block depends on the GPU capability (i.e. 32K for CUDA capability 2.0 and 64K for CUDA capability 3.0).

In Figure 5.5 a PTX code fragment that can be used by the kernel for register reservation is shown. From the point of view of an adversary, register spilling is an easy way to access global memory bypassing Runtime access primitives. In our experiments we tried to understand whether such an access mechanism to global memory would undergo the same access controls memory allocation primitives (e.g. *cudaMalloc()*) are subject to.

Surprisingly, we discovered this was not true and the malicious process can effectively exploit register spilling to access memory areas that had been reserved to other *cudaContexts*. Further, in this case the malicious process can access those locations even while the legitimate process still owns them (namely before it calls the *cudaFree()*). This is the reason why we believe this latter leakage is the most dangerous among the presented ones.

Actually, we were able to replicate such an attack only on the Kepler GPU. Fermi seems immune to this attack.

In the remainder of this section we describe in details the steps required to obtain the leakage. As a preliminary step, we zeroed the whole memory available on the device in order to avoid tainting the results. As for previous experiments, we needed two independent host processes, P_a and P_b, as the honest and the malicious processes, respectively.

P_a performs several writes into the global memory whereas P_b tries to circumvent the memory isolation mechanisms to read those pieces of information. In order to verify in a more reliable way the attack outcome, P_a writes a pattern that is easy to recognize; in particular P_a allocates an array and marks the first location with the hexadecimal value *0xdeadbeef*. At position j of the array, P_a stores the value *0xdeadbeef* + j where j represents the offset in the array represented in hexadecimal.

P_b reserves a predefined number of registers and copies the content of the registers back to the host memory; in this way P_b tries to exploit the register spilling to violate the memory locations reserved by P_a. Indeed, if the register spilling mechanism is not properly implemented, then some memory locations reserved to P_a could be inadvertently assigned to P_b. We ran P_a and P_b concurrently and we checked for any leaked location marked by P_a.

As *per* Figure 5.4, in this case P_b succeeds in accessing memory locations reserved by P_a before this latter executes the *cudaFree* memory releasing operation. Note that this behavior is different from the one observed for the global memory attack described in Section 5.4.3.

We ran both P_a and P_b with a gridsize of 2 blocks and a blocksize of 32 threads. In Table 5.3 the number of bytes of P_a that are read by process P_b is reported. The analysis was conducted by varying the amount of registers declared by P_b and by varying the amount of memory locations declared by P_a. Results show two rounds of the experiment. As an example, if P_b reserves a 32KB register space (corresponding to 8K 32-bit registers), and P_a allocates an amount of memory equal to $32MB$, by executing the mentioned experiment twice, we obtain an information leakage of 64KB (i.e. 16K 32-bit words). This is due to the fact that in different rounds the leakage comprises different memory locations. The rationale is the dynamic memory management mechanism that is implemented in the GPU driver and in the CUDA Runtime. As a consequence, this attack is even more dreadful as the adversary, by executing several rounds, can potentially read the whole memory segment allocated by P_a.

Take-away point: *GPU registers can leak information, since register allocation uses GPU memory when hardware registers are exhausted.*

In order to better quantitatively evaluate this phenomenon, we have investigated and analyzed the relationship between the number of locations where the leakage succeeds and the number of executed rounds. In Figure 5.6 results are shown with respect to a number of rounds ranging from 1000 to 10000. Growth is linear in the number of rounds. In particular, the leakage starts from $32K$ locations for 1000 rounds and reaches $320K$ locations leaked when the number of rounds is 10000. The leaked locations belong to contiguous memory areas; the distance between each location is 32 bytes. For example, if the leaked locations start from byte 0 and end at byte 320, then we obtain 10 locations: one location every 32 bytes. We claim that this behavior depends on the implementation and the configuration of the kernel.

The results of this experiment suggest a further study on the possibility for a malicious process to obtain write access to the leaked locations. In order to investigate this vulnerability, we performed an additional set of experiments.
We kept the same configuration as the previous test but, to foster the detection of the potential unauthorized write accesses, we used a cryptographic hash function. Indeed, thanks to the properties of hash functions, if the malicious process succeeds in interfering with the computation of the legitimate process—for instance, by altering even only a single bit—, this would cause (with overwhelming probability) errors in the output of the legitimate process.
For P_a, we used a publicly-available GPU implementation of the *SHA-1* hash function included into the *SSLShader*[3]: an SSL reverse proxy transparently translating SSL sessions to TCP sessions for back-end servers. The encryption/decryption is performed on-the-fly through a CUDA implementation of various cryptographic algorithms. Actually, the code implementing the GPU cryptographic algorithm is contained in the *libgpucrypto* library which can be downloaded from the same web site. For our experiments we used the version 0.1 of this library.

In this test, P_a uses the GPU to compute the *SHA-1* for 4096 times on a constant plaintext of 16KB. P_a stores each hash in a different memory location. To test the integrity of GPU-computed hashes, P_a also computes the *SHA-1* on the CPU using the OpenSSL library and then compares this result with the ones computed on the GPU.
The malicious process P_b tries to taint the computation of P_a by writing a constant value into the leaked locations.

In our test we ran P_a 1000 times and concurrently we launched the malicious process P_b. Even if in most cases we were able to read a portion of the memory reserved to P_a, the write instruction

[3]The source code is available at http://shader.kaist.edu/sslshader (Last accessed on 02/27/2014)

Algorithm 2: The pseudo-code of the attacking process in the AES case study.

Input:
$\vec{M}$: The plaintext
l: The length of the plaintext
$\vec{K}$: array of identifiers of the encryption key
Output: TRUE if the attack succeeds, FALSE otherwise

FindLeakage($\vec{M}$,l,$\vec{K}$)

```
begin
    s ← size of current allocable global memory on GPU
    P ← cudamalloc(s)
    j ← 0
    while j < s do
        w ← P[j]
        if w ∈ K⃗ then
            cudamemset(P,0,s) /*zeroing*/
            return TRUE;
        else if w == M[0] then
            i ← 0
            while i < l do
                if P[j + i] != M[i] then
                    cudamemset(P,0,s) /*zeroing*/
                    return FALSE
                i++
            end
            cudamemset(P,0,s) /*zeroing*/
            return TRUE
        j++
    end
    cudamemset(P,0,s)
    return FALSE
end
```

was ignored and all the hashes computed on the GPU were correct.

Take-away point: *The register spilling vulnerability does not allow interfering with the computation of the legitimate process*

5.5 Case Study: SSLShader

In order to evaluate the impact of the global memory vulnerability in a real-world scenario, we attacked the CUDA implementation of AES presented in [148] which is part of *SSLShader*. The SSLShader comes with some utilities that can be used to verify the correctness of the implemented algorithms. To run our experiments, we modified one of these utilities. In particular, we

changed the AES test utility to encrypt a constant plaintext using a fixed key. We chose a constant plaintext of $4KB$ (i.e. the first two Chapters of the *Divine Comedy* written in latex) and we set the 128-bit encryption key to the juxtaposition of following words: *0xdeadbeef, 0xcafed00d, 0xbaddcafe,0x8badf00d*.

In this experiment we assume that the GPU is shared between the adversary and the legitimate process. Further, we assume that the adversary can read the ciphertext.

The steps performed by the attacking process are described in Algorithm 1. We consider the attack successful in two cases: in the first case, the adversary gets access to the *whole* plaintext (line 11)—achieving *plaintext leakage*; in the second case, the adversary obtains some words of the encryption key (line 8)—achieving *key leakage*. Even if this latter case is less dreadful than the former one, it still jeopardizes security. Indeed, in order to obtain the desired information, the adversary could attack the undisclosed portion of the key (e.g. via brute force, differential cryptanalysis [149]) and eventually decrypt the message.

The experiment is composed of the following steps: first we run an infinite loop of the CUDA AES encryption; in the meanwhile we run Algorithm 1 100 times. In order to avoid counting a single leakage event more than once, each execution of the Algorithm 1 zeroes the memory (lines $9, 15, 19, 24$). We repeated this experiment 50 times on both the Kepler and the Fermi architectures, measuring the amount of successful attacks per round. In order to preserve the independence across different rounds, at the end of each round we rebooted the machine.

For the Kepler we measured a successful attack mean equal to 30% with a standard deviation equal to 0.032. As for the Fermi architecture, we measured a mean success rate of 12% with a standard deviation of 0.03. Figure 5.7 details the results of 9 randomly chosen rounds in terms of *key leakage* and *plaintext leakage* for Kepler. The *plaintext leakage* is slightly more frequent than the *key leakage*. Figure 5.8 shows the results for Fermi; in this case the frequencies of the two leakages are equal. In these two figures each bin represents the number of times that the leakage occurred in 100 runs of Algorithm 1.

5.5.1 Discussion and qualitative analysis

It is worth adding further considerations about the conditions that lead to an effective information leakage. Indeed, with our attack methodology we are able to leak only the final state of the previous GPU process. This limitation is due to the exclusive access granted by the driver to host threads that access the GPU; only one *cudaContext* is allowed to access the GPU at a given

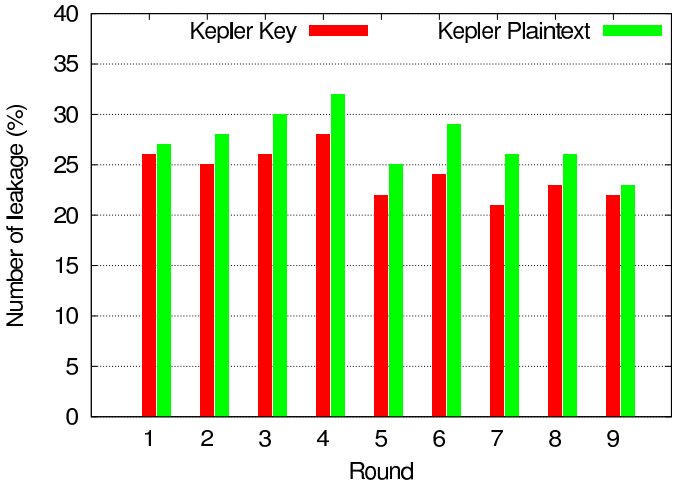

FIGURE 5.7: Number of leakages in the Kepler architecture

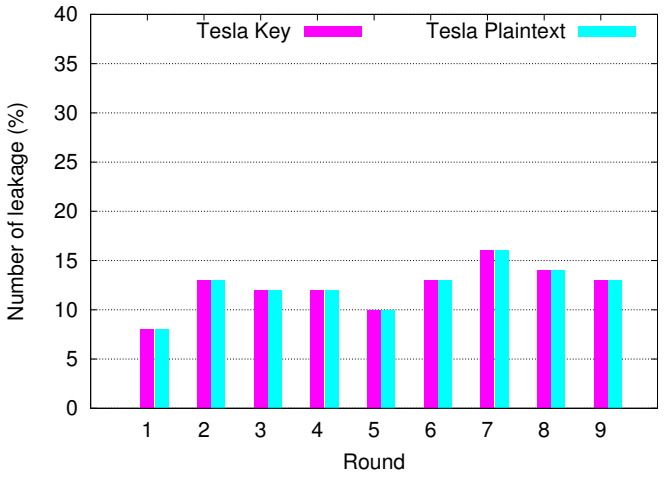

FIGURE 5.8: Number of leakages in the Fermi architecture

time.

However, note that in some circumstances the final state of a computation is sensitive (e.g. the decryption of a ciphertext, the output of a risk-analysis function or the Universal Transverse Mercator (UTM) locations of an oil well). In other circumstances the final state of a computation is not sensitive or even public. For instance, knowing the final state of an encryption process (i.e. the ciphertext) does not represent a threat.

However, in our experiments we were able to recover the original plaintext even after the encryption process ended. This was possible due to the fact that the plaintext and the ciphertext were stored in different locations of the global memory. As such, this vulnerability depends on both the implementation and the computed function and does not hold in the general case.

Another important consideration about the presented case study concerns the precondition of the attack. In order to perform the *key leakage* test we assume that the adversary knows a portion

of the key (which is needed to perform searches in memory and detect if the leakage happened). However, as shown in [150] it is possible to exploit the high entropy of the encryption keys to restrict the possible candidates to a reasonable number. In fact, secure encryption keys usually have a higher entropy than other binary data in memory.

As a further technical note we found out that SSLShader [148], i.e. the widespread publicly available implementation of cryptographic algorithms on GPU, makes use of the *cudaHostAlloc* CUDA primitive. This primitive allocates a memory area in the host memory that is page-locked and accessible to the device (pinned memory). Although this can be considered more secure than using the cudaMalloc, it is fully vulnerable to information leakage as well. In fact, the CUDA Runtime copies the data to the GPU global memory on behalf of the programmer when it is more convenient. This is important since it shows that even code implemented by "experts" actually shows the same deficiencies as regards security. This finding, together with the others reported in the chapter, call for solutions to this severe vulnerability.

5.6 Proposed Countermeasures

In previous sections the main issues and vulnerabilities of Kepler and Fermi CUDA architectures have been highlighted. It is worth noting that discovered leakages are not tied to a particular version of the device driver or GPU architecture. They are intrinsic to present GPU architectures that aim at performance without considering security.

Given that shared GPU security issues will be increasingly relevant in the future, this section suggests alternative approaches and countermeasure that prevent or at least dramatically limit the described information leakage attacks.

In general, from the software point of view, CUDA code writers should pay attention to zeroing memory as much as possible at the end of kernel execution. Unfortunately, this is troublesome for a number of reasons:

- Most often the programmer does not have fine control over kernel code (e.g. if the kernel is the outcome of high-level programming environments such as JavaCL, JCUDA, etc);
- The kernel programmer usually aims at writing the fastest possible code without devoting time to address security/isolation issues that might hamper performance.

As such, we believe that the best results can be obtained if security enhancements are performed at the driver/hardware level. From the CUDA Platform/Hardware point of view, suggestions comprise:

- Finer MMU-based memory protection mechanisms have to be devised to prevent concurrent kernels from reading other kernels' memory;
- Finer monitoring and access control: CUDA should allow the OS to monitor usage and to control access to GPU resources; this way anomalous resource usage and suspicious access patterns could be detected and/or prevented.

In the following, for each of the discovered vulnerabilities, we provide related mitigation countermeasures.

5.6.1 Shared Memory

As for the shared memory leakage shown in Section 5.4.2, the proposed fix makes use of a memory-zeroing mechanism. As already pointed out in Section 5.4.2 the shared memory attack is ineffective once the host process terminates. The vulnerability window goes from kernel completion to host process completion. As a consequence, the memory-zeroing operation is better executed inside the kernel. In our opinion, this is a sensitive solution since shared memory is an on-chip area that cannot be directly addressed or copied by the host thread. As such, it is not possible to make use of it from outside a kernel function.

In order to measure the overhead that an in-kernel memory-zeroing approach would have on a real GPU, we developed and instrumented a very simple CUDA code (addition of two vectors). Two kernel functions, K_1 and K_2 were developed: K_1 receives as input two randomly-initialized vectors A, B; K_1 sums the two vectors and stores the result in vector C; K_2 is the same as K_1 but in addition it "zeroes" the shared memory area by overwriting it with the value read from $A[0]$[4]. We measured the execution time difference between K_1 and K_2 by varying the vectors' size, as this experiment was just aimed at evaluating the scalability of the zeroing operation. In particular, for K_1 and K_2, we performed this experiment accessing an increasing number of locations up to the maximum available shared memory. Such value depends on the GPU capability and corresponds to 672KB for the Tesla C2050 and 32KB for the GT640. We noticed

[4] we did not actually "zero" the memory using the value 0 in order to prevent the compiler from performing optimization that would have affected the result.

that the introduced overhead was constant and not affected by the number of memory accesses. In particular we measured a mean overhead of 1.66 *ms* on the Kepler and 0.27 *ms* on the Tesla card. We can conclude that, for kernels with a reasonable duration (e.g. longer than 0.1 sec), the proposed fix can be applied without noticeably affecting GPU performance.

5.6.2 Global Memory

As described in Section5.4.3, accessing global memory through CUDA primitives can cause an information leakage. The natural fix would consist in zeroing memory before it is given to the requesting process. This way, when information is deleted the malicious process is not able to access it. This approach should naturally be implemented inside the CUDA Runtime. To assess the impact of the overhead introduced by this solution, we measured the overhead that the same zeroing operation imposes to traditional memory allocation. CUDA Runtime function *cudaMemset* was used for zeroing memory content. An incremental size buffer was tested in the experiments. In our tests, size ranged from 16MB to 512MB, in steps of 16MB. We then measured the overhead induced by the additional zeroing operations. To achieve this goal, we instrumented the source code with the *EventManagement* Runtime library function. Through these primitives we were able to compute the time elapsed between two events in milliseconds with a resolution of around 0.5 microseconds.
As shown in Figure 5.9, the introduced overhead is not negligible. On both Tesla and Kepler platforms the induced overhead shows a linear relationship to the allocated buffer size. However, Tesla's line steepness is much lower than the Kepler counterpart. The reason is that the two GPUs feature a much different number of multiprocessors (14 for Tesla vs. 2 for the Kepler). That is, the level of achievable parallelism is quite different.

As regards the implemented memory-zeroing approach, threads run in parallel, each one zeroing its serial memory area. This accounts for the measured overhead. However, since the data chunk has a very limited size, the overhead is—in absolute values—very small (1.66 *ms* GeForce Kepler 0.27 *ms* Tesla Fermi). It is worth noting that a low-level hardware approach would be surely faster. However, in general zeroing does worsen performance in GPU [147], as these techniques force additional memory copies between host and device memory. In addition, we only have implementation details on global memory that is actually implemented on commodity GDDRx memory, i.e. as standard host memory. Introducing an additional mechanism

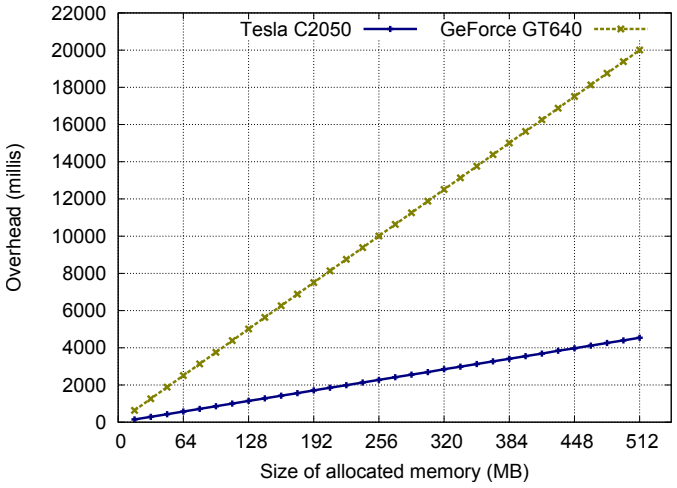

FIGURE 5.9: Overhead introduced by the proposed countermeasure for the global memory leak.

to perform smart memory zeroing would require an overall redesign of the GDDR approach and as such it will most probably increase RAM cost. Hardware-based fast zeroing would probably be the most feasible and convenient solution. However, inner details about low-level memory implementation for CUDA cards are only known by NVIDIA.

Pertaining to the selective deletion of sensitive data, selectively zeroing specific memory areas is potentially feasible and would potentially reduce unnecessary memory transfers between GPU and CPU, since most data would not have to be transferred again. A "smart" solution would probably be the addition of CUDA language extensions (source code tags) to mark the variables/memory areas that have to be zeroed since containing sensitive data. On the one hand, this would require language/compiler modifications while, on the other hand, it would save some costly data transfers. However, this approach implies some caveats, as there is the risk of pointing the adversary exactly to the memory and registers where sensitive data is. Further, such sensitive data when in transit between CPU and GPU crosses various memory areas that are still potentially accessible. As such, for performance sake, sensitive areas should be as contiguous as possible.

5.6.3 Registers

Register allocation is handled at the lower level of the software stack, hence we understand this leak is due to a flaw regarding the memory isolation implementation. Therefore, fixing this leakage at the application level is quite difficult. A much simpler workaround would be to implement the fix at the GPU driver level. Unfortunately, given the closed-source nature of

the driver, at present only NVIDIA can provide a solution for this issue. In particular, the driver should preserve the following properties: first, the registers should not spill to locations in global memory that are still reserved for host-threads; second, the locations of the spilled registers must be reset to zero when they are released.

5.7 Related Work

Due to the commercial strategy to hide implementation details from competitors, manufacturers are reluctant on publishing the internals of their solutions. In fact, documentation is mostly generic, marketing-oriented, and incomplete. This fact hinders the analysis of the information leakage problem on Graphics Processing Units. As a consequence, in the literature most of the available architectural information over existing hardware is due to black-box analysis. In particular [151] developed a microbenchmark suite to measure architectural characteristics of CUDA GPUs. The analysis showed various undisclosed characteristics of the processing elements and the memory hierarchies and exposed undocumented features that impact both program performance and program correctness. CUBAR [152] used a similar approach to discover some of the undisclosed CUDA details. In particular, CUBAR showed that CUDA features a Harvard architecture on a Von Neumann unified memory. Further, since the closed-source driver leverages (the deprecated) security through obscurity paradigm, inferring information from PCIe bus [142] is possible as partially shown in [153].

GPU thread synchronization issues are introduced and discussed in [154] whereas [155] describes multicore computing reliability issues. Such analysis are aimed towards correctness, reliability and performance, whereas in our work we focus on actual GPU thread behavior and related consequences on data access.

Vulnerabilities have been discovered in the past in the NVIDIA GPU driver [139], a key component of the CUDA system that has kernel level access (via the NVIDIA kernel module). Such vulnerabilities can have nasty effects on the whole system and can lead to even further information leakage, due to root access capabilities.

A preliminary work by Barenghi et al. [156] investigated side channel attacks to GPUs using both power consumption and electromagnetic (EM) radiations. The proposed approach can be useful for GPU manufacturers to protect data against physical attacks. However, for the attacks

presented in this chapter, the adversary does not need either physical access to the machine or root privileges.

As regards secure data deletion, Reardon et al. [157] present a taxonomy of the characteristics of secure deletion approaches. They suggest that the best approach strongly depends on the used medium. In [140] authors discuss the use of GPUs in computing clusters. They realize that applications that use GPUs can frequently leave them in an unusable state. In order to address such issue, prior to node deallocation, a node health check and memory scrubber tool is run that allocates all available GPU device memory and fills it in with a user-supplied pattern. This work suggests that some security issue for shared GPU computing exists but it does not investigate possible information leakage. The suggested coarse-grained workaround uses a wrapper library [158] to perform memory scrubbing at the end of the context.

An interesting work by Maurice et al. [159] focuses on potential information leakage in virtualized and cloud computing environments. Maurice aims at investigating the causes behind the leakage. However, his paper does not detail the proposed attacks and it is not clear whether leakages require privileged access. Our chapter introduces and details a different set of information leakages that expose to a more serious threat as root access is not needed.

Finally, a relevant limitation of the current GPU architectures is the fact that the OS is completely excluded from the management of GPU threads. The first attempts to overcome the limits of present GPU platforms aim at giving the OS kernel the ability to control the marshalling of GPU tasks [160]. In fact, the GPU can be seen as an independent computing system where the OS role is played by the GPU device driver; as a consequence, host-based memory protection mechanism are actually ineffective to protect GPU memory.

5.8 Conclusion and Future Work

In this work we provide several contributions, shedding light on some security issues of the increasingly successful GPGPU computing field. In particular, we detail some critical vulnerabilities in CUDA architectures affecting shared memory, global memory, and registers. We also experimentally show how such vulnerabilities can be exploited to generate information leakage. Furthermore, fixes are proposed and discussed to tackle the highlighted vulnerabilities.
Given the generality of the identified vulnerabilities and the architectural complexity of the GPU

field, the results reported in this chapter—other than being interesting on their own—also pave the way for further research.

6 Concluding Remarks

In this Thesis we have addressed the problem of secure outsourced distributed computing. After a brief introduction we have focused on a number of aspects, namely computing node integrity issues, execution behavior monitoring and cheating resilience through smart task redundancy. Later on, we have discussed GPU cloud privacy and isolation issues. In this final chapter we summarize the contributions of this thesis, draw conclusions and depict highlights of future work.

6.1 Summary of the Contributions

The contributions of this thesis are focused on secure monitoring, evaluation and protection of code execution and data over distributed heterogeneous cloud computing nodes. We have been working to realize a comprehensive approach for securely collecting execution data and checking computation behavior and outcomes. We have presented the components of such a novel approach for monitoring and protecting computations hosted on nodes out of the client control. In other words, in the considered scenarios, the nodes are not considered trusted. There are mainly three security aspects that have been dealt with when ensuring reliable secure computation over remote cloud nodes.

The first aspect has been VM integrity enforcement [1] where we have used advanced introspection approaches to detect and react to attacks and anomalies of cloud VMs. Then, in order to limit the ability of unauthorized/malicious code to affect the cloud node, we have proposed an

execution monitoring system [2], based on Execution Path Analysis, that allows detecting and reacting to anomalous application behavior before it can produce further damage. The knowledge we have acquired over cloud multitenancy issues and on virtualization technology has also helped publishing a book chapter on the topic [10] that sheds a light over future cloud-supporting technology. As a related result, we have devised an advanced tainting approach [3] that helps assessing mobile applications based on their behavior. The proposed approach helps classifying them as goodware, greyware or malware.

The second aspect is computation reliability, aimed at limiting the impact of erroneous or fake data and/or computation over final results. With respect to this relevant problem, we have devised and tested advanced effective task distribution [4] and result evaluation and node-checking approaches [5] that minimize the chances of having cheating nodes affect the global result. In particular, "AntiCheetah: an Autonomic Multi-round Approach for Reliable Computing" [5] won the Best Paper Award at the 10th IEEE International Conference on Autonomic and Trusted Computing (ATC) in 2013. Other related interesting results towards better cloud forensics and against action log cheating are currently under review [6].

The third aspect is related to privacy issues in the cloud. In particular we have given scientific evidence that manycore cloud computing is inherently insecure and we have suggested both new execution models and practical remedies [7]. The proposed approaches prevent users from accessing or affecting data and computation of other users sharing a multitenant GPU cloud. The knowledge over manycore issues also convinced us to contribute a chapter to a book on the topic [9]. Such chapter stresses GPU usefulness for increasing security and discusses the main security issues of the shared-GPU approach.

We believe that the overall impact of the contributions we have given on these three main aspects of outsourced distributed computing in the cloud is relevant.

6.2 Published Work

Work published during the Ph.D. :

- R. Di Pietro, F. Lombardi. Secure Virtualization for Cloud Computing. Elsevier JNCA. ISSN 1084-8045 [1].

- R. Di Pietro, F. Lombardi, and M. Signorini. CloRexPa: Cloud resilience via execution path analysis. Future Generation Computer Systems. ISSN 0167-739X [2].

- G. Suarez de Tangil, F. Lombardi, J. E. Tapiador and R. Di Pietro. Thwarting Obfuscated Malware via Differential Fault Analysis. IEEE Computer Magazine. ISSN 0018-9162 [3].

- R. Di Pietro, F. Lombardi, F. Martinelli and D. Sgandurra. CheR: Cheating Resilience in the Cloud via Smart Resource Allocation. 6th Intl. Symposium on Foundations & Practice of Security (FPS) [4].

- R. Di Pietro, F. Lombardi, F. Martinelli and D. Sgandurra. AntiCheetah: an Autonomic Multi-round Approach for Reliable Computing. 10th IEEE International Conference on Autonomic and Trusted Computing (**ATC 2013 Best Paper Award**) [5].

- F. Lombardi and R. Di Pietro. (Book Chapter) title: "Towards a GPU Cloud: Benefits and Security Issues" Book title: "Continued Rise of the Cloud: Advances and Trends in Cloud Computing". ISBN 978-1-4471-6451-7 [9].

- F. Lombardi and R. Di Pietro. (Book Chapter) title: "Virtualization and Cloud Security: Benefits, Caveats and Future Developments" Book title: "Cloud Computing: Challenges, Limitations and R&D Solutions" [10].

- F. Lombardi, R. Spigler. The Evolution of the approach to Scientific Computing: a Survey. Parallel & Cloud Computing ISSN: 2304-9456 [8].

6.3 Work Currently Under Review

Manuscripts submitted for publication during the Ph.D. and under review:

- R. Battistoni, R. Di Pietro and F. Lombardi. Enforcing a Good Timeline for Reliable Forensics in Cloud Computing [6].

- R. Di Pietro, F. Lombardi and A. Villani. CUDA Leaks: Information Leakage in GPU Architectures [7].

6.4 Future Work

We are actively working on improving and extending the results discussed in present Thesis, especially as regards advanced cheating resilience over possibly malicious distributed nodes. In particular, future research will address advanced node coalitions, i.e. where some of the cheaters can communicate among themselves to devise a common cheating strategy. In addition, we will also consider smart cheaters that may have a partial (or total) view of the current assignment matrix.

We will also further investigate application behavior in the mobile and GPU cloud scenarios. In particular, we are currently investigating GPGPU security issues within Windows and mobile OSes over heterogeneous GPU architectures. We also plan to further enhance code/application behavior analysis techniques and apply them to novel scenarios.

We finally aim at fully integrating the results we obtained and improving the achievements that contribute to an overall securer and more reliable distributed cloud computing. We expect our results to pave the way for further research.

Bibliography

[1] Flavio Lombardi and Roberto Di Pietro. Secure Virtualization for Cloud Computing. *J. Netw. Comput. Appl.*, 34(4):1113–1122, jul 2011. ISSN 1084-8045. doi: 10.1016/j.jnca.2010.06.008. URL http://dx.doi.org/10.1016/j.jnca.2010.06.008.

[2] Roberto Di Pietro, Flavio Lombardi, and Matteo Signorini. CloRExPa: Cloud Resilience via Execution Path Analysis. *Future Gener. Comput. Syst.*, 32:168–179, mar 2014. ISSN 0167-739X. doi: 10.1016/j.future.2012.05.010. URL http://dx.doi.org/10.1016/j.future.2012.05.010.

[3] Guillermo Suarez-Tangil, Flavio Lombardi, Juan E. Tapiador, and Roberto Di Pietro. Thwarting Obfuscated Malware via Differential Fault Analysis. *Computer*, 2014. ISSN 0018-9162. To appear.

[4] Roberto Di Pietro, Flavio Lombardi, Fabio Martinelli, and Daniele Sgandurra. CheR: Cheating Resilience in the Cloud via Smart Resource Allocation. In Jean Luc Danger, Mourad Debbabi, Jean-Yves Marion, Joaquin Garcia-Alfaro, and Nur Zincir Heywood, editors, *Foundations and Practice of Security*, Lecture Notes in Computer Science, pages 339–352. Springer International Publishing, 2014. ISBN 978-3-319-05301-1. doi: 10.1007/978-3-319-05302-8_21. URL http://dx.doi.org/10.1007/978-3-319-05302-8_21.

[5] Roberto Di Pietro, Flavio Lombardi, Fabio Martinelli, and Daniele Sgandurra. Anticheetah: An autonomic multi-round approach for reliable computing. In *Ubiquitous Intelligence and Computing, 2013 IEEE 10th International Conference on and 10th International Conference on Autonomic and Trusted Computing (UIC/ATC)*, pages 371–379, Dec 2013. **Best Paper Award**.

[6] Roberto Battistoni, Roberto Di Pietro, and Flavio Lombardi. Enforcing a Good Timeline for Reliable Forensics in Cloud Computing, 2014. Under review.

[7] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. CUDA Leaks: Information Leakage in GPU Architectures. *ArXiv.org*, 2013.

[8] Flavio Lombardi and Renato Spigler. The Evolution of the approach to Scientific Computing: a Survey. *Journal of Parallel & Cloud computing*, 2014. ISSN 2304-9456. To appear.

[9] Flavio Lombardi and Roberto Di Pietro. *Continued Rise of the Cloud: Advances and Trends in Cloud Computing*, chapter Towards a GPU Cloud: Benefits and Security Issues. Springer, 2014. ISBN 978-1-4471-6451-7. Book Chapter to appear.

[10] Flavio Lombardi and Roberto Di Pietro. *Cloud Computing: Challenges, Limitations and R&D Solutions*, chapter Virtualization and Cloud Security: Benefits, Caveats and Future Developments. Springer, 2014. Book Chapter to appear.

[11] Eucalyptus. Eucalyptus. http://eucalyptus.com/, 2009.

[12] Openecp. Openecp. http://www.openecp.org, 2010.

[13] Enomaly. Enomalism. http://www.enomaly.com, 2009.

[14] Siani Pearson. Taking account of privacy when designing cloud computing services. In *CLOUD '09: Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 44–52, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3713-9. doi: http://dx.doi.org/10.1109/CLOUD.2009.5071532.

[15] Lin Gu and Shing-Chi Cheung. Constructing and testing privacy-aware services in a cloud computing environment: challenges and opportunities. In *Internetware '09: Proceedings of the First Asia-Pacific Symposium on Internetware*, pages 1–10, New York, NY, USA, 2009. ACM.

[16] Frank Siebenlist. Challenges and opportunities for virtualized security in the clouds. In *SACMAT '09: Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 1–2, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-537-6. doi: http://doi.acm.org/10.1145/1542207.1542209.

[17] Christian Cachin, Idit Keidar, and Alexander Shraer. Trusting the cloud. *SIGACT News*, 40(2):81–86, 2009. ISSN 0163-5700. doi: http://doi.acm.org/10.1145/1556154.1556173.

[18] Enisa. Cloud computing risk assessment. http://www.enisa.europa.eu/act/rm/files/deliverables, 2009.

[19] Michael Armbrust, Armando Fox, and Rean Griffith. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.

[20] Thomas Ristenpart, Eran Tromert, Hovav Shacham, and al. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *CCS '09: Proceedings of the 14th ACM conference on Computer and communications security*, pages 103–115, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-352-5.

[21] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 18–29, New York, NY, USA, 1994. ACM. ISBN 0-89791-732-4. doi: http://doi.acm.org/10.1145/191177.191183.

[22] AIDEteam. Advanced intrusion detection environment. http://sourceforge.net/projects/aide, November 2005.

[23] Secunia. Secunia advisory. http://secunia.com/advisories/36389, 2009.

[24] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: http://doi.acm.org/10.1145/1315245.1315262.

[25] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07: Proceedings of*

twenty-first ACM SIGOPS symposium on Operating systems principles, pages 335–350, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: http://doi.acm.org/10.1145/1294261.1294294.

[26] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 233–247, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3168-7. doi: http://dx.doi.org/10.1109/SP.2008.24.

[27] Michael Peter, Henning Schild, Adam Lackorzynski, and Alexander Warg. Virtual machines jailed: virtualization in systems with small trusted computing bases. In *VDTS '09: Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*, pages 18–23, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-473-7. doi: http://doi.acm.org/10.1145/1518684.1518688.

[28] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87402-7. doi: http://dx.doi.org/10.1007/978-3-540-87403-4_1.

[29] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, 2009.

[30] Yih Huang, David Arsenault, and Arun Sood. Closing cluster attack windows through server redundancy and rotations. In *CCGRID*, pages 21–33, 2006.

[31] Tobias Distler, Rüdiger Kapitza, and Hans P. Reiser. Efficient state transfer for hypervisor-based proactive recovery. In *WRAITS '08: Proceedings of the 2nd workshop on Recent advances on intrusiton-tolerant systems*, pages 1–6, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-986-9. doi: http://doi.acm.org/10.1145/1413901.1413905.

[32] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Resilient intrusion tolerance through proactive and reactive recovery. *Pacific*

Rim International Symposium on Dependable Computing, IEEE, 0:373–380, 2007. doi: http://doi.ieeecomputersociety.org/10.1109/PRDC.2007.52.

[33] Mark Pollitt, Kara Nance, Brian Hay, Ronald C. Dodge, Philip Craiger, Paul Burke, Chris Marberry, and Bryan Brubaker. Virtualization and digital forensics: A research and education agenda. *J. Digit. Forensic Pract.*, 2(2):62–73, 2008. ISSN 1556-7281. doi: http://dx.doi.org/10.1080/15567280802047135.

[34] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/1496091.1496100.

[35] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, and Thomas Sandholm. What's inside the Cloud? An architectural map of the cloud landscape. In *CLOUD '09: Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 23–31, Washington, DC, USA, 2009. IEEE Computer Society.

[36] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing resource management through a grid middleware: A case study with diet and eucalyptus. *Cloud Computing, IEEE International Conference on*, pages 151–154, 2009. doi: http://doi.ieeecomputersociety.org/10.1109/CLOUD.2009.70.

[37] Andreas Haeberlen. A case for the accountable cloud. In *LADIS '09: 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, 2009.

[38] John Bethencourt, Dawn Song, and Brent Waters. New techniques for private stream searching. *ACM Trans. Inf. Syst. Secur.*, 12(3):1–32, 2009. ISSN 1094-9224. doi: http://doi.acm.org/10.1145/1455526.1455529.

[39] Matthew Smith, Thomas Friese, Michael Engel, and Bernd Freisleben. Countering security threats in service-oriented on-demand grid computing using sandboxing and trusted computing techniques. *J. Parallel Distrib. Comput.*, 66(9):1189–1204, 2006. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2006.04.009.

[40] Brian Hay and Kara Nance. Forensics examination of volatile system data using virtual introspection. *SIGOPS Oper. Syst. Rev.*, 42(3):74–82, 2008. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1368506.1368517.

[41] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing tcb size by using untrusted components: small kernels versus virtual-machine monitors. In *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 22, New York, NY, USA, 2004. ACM. doi: http://doi.acm.org/10.1145/1133572.1133615.

[42] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[43] Ronald Perez, Leendert van Doorn, and Reiner Sailer. Virtualization and hardware-based security. *IEEE Security and Privacy*, 6(5):24–31, 2008. ISSN 1540-7993. doi: http://dx.doi.org/10.1109/MSP.2008.135.

[44] Jan S. Rellermeyer, Michael Duller, and Gustavo Alonso. Engineering the cloud from software modules. In *CLOUD '09: Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 32–37, Washington, DC, USA, 2009. IEEE Computer Society.

[45] RedHat. Libvirt. http://libvirt.org, 2007.

[46] Yun Huang, Xianjun Geng, and Andrew B. Whinston. Defeating DDoS attacks by fixing the incentive chain. *ACM Trans. Internet Technol.*, 7(1):5, 2007. ISSN 1533-5399. doi: http://doi.acm.org/10.1145/1189740.1189745.

[47] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. *SIGOPS Oper. Syst. Rev.*, 39(5):133–147, 2005. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1095809.1095824.

[48] Debian. Dsa-1571-1 Openssl: predictable random number generator. http://www.debian.org/security/2008/dsa-1571, 2008.

[49] CVE. Common vulnerabilities and exposures-2008-2364. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2364, 2008.

[50] Honeynet Project. Sebek. https://projects.honeynet.org/sebek/, 2003.

[51] Sam Johnston. Cve-2008-4990 enomaly ecp/enomalism: Insecure temporary file creation vulnerabilities. http://www.securityfocus.com/archive/1/archive/1/500573/100/0/threaded, 2009.

[52] Ben Smith, Rick Grehan, and Tom Yager. Byte-unixbench: A Unix benchmark suite. http://code.google.com/p/byte-unixbench/.

[53] Dimitrios Zissis and Dimitrios Lekkas. Addressing cloud computing security issues. *Future Generation Computer Systems*, 28(3):583–592, 2012. ISSN 0167-739X. doi: 10.1016/j.future.2010.12.006. URL http://www.sciencedirect.com/science/article/pii/S0167739X10002554.

[54] Jean-Claude Laprie. Resilience for the scalability of dependability. In *NCA '05: Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications*, pages 5–6, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2326-9. doi: http://dx.doi.org/10.1109/NCA.2005.44.

[55] Marco Balduzzi, Jonas Zaddach, Davide Balzarotti, Engin Kirda, and Sergio Loureiro. A security analysis of amazon's elastic compute cloud service. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1427–1434, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0857-1. doi: 10.1145/2245276.2232005. URL http://doi.acm.org/10.1145/2245276.2232005.

[56] Dong Zhou. Diagnosing misconfiguration with dynamic detection of configuration invariants. In *Proc. Work. on Hot Topics in System Dependability*, Berkeley, CA, USA, 2007. USENIX Association. URL http://portal.acm.org/citation.cfm?id=1323140.1323149.

[57] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[58] Matthias Schmidt, Lars Baumgartner, Pablo Graubner, David Bock, and Bernd Freisleben. Malware detection and kernel rootkit prevention in cloud computing environments. In *Proc. Int. Euromicro Conf. on Parallel, Distributed and Network-Based Processing*, PDP '11, pages 603–610, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4328-4. doi: http://dx.doi.org/10.1109/PDP.2011.45. URL http://dx.doi.org/10.1109/PDP.2011.45.

[59] Kenneth Goldman, Reiner Sailer, Dimitrios Pendarakis, and Deepa Srinivasan. Scalable integrity monitoring in virtualized environments. In *Proc. ACM workshop on Scalable trusted computing*, STC '10, pages 73–78, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0095-7. doi: http://doi.acm.org/10.1145/1867635.1867647. URL http://doi.acm.org/10.1145/1867635.1867647.

[60] Jianxin Li, Bo Li, Tianyu Wo, Chunming Hu, Jinpeng Huai, Lu Liu, and K.P. Lam. Cyberguarder: A virtualization security assurance architecture for green cloud computing. *Future Generation Computer Systems*, 28(2):379–390, 2012. ISSN 0167-739X. doi: 10.1016/j.future.2011.04.012. URL http://www.sciencedirect.com/science/article/pii/S0167739X1100063X.

[61] Fabrizio Baiardi, Diego Cilea, Daniele Sgandurra, and Francesco Ceccarelli. Measuring semantic integrity for remote attestation. In *Proc.Intl.Conf. on Trusted Computing*, Trust '09, pages 81–100, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00586-2. doi: http://dx.doi.org/10.1007/978-3-642-00587-9_6. URL http://dx.doi.org/10.1007/978-3-642-00587-9_6.

[62] Doug Beck, Binh Vo, and Chad Verbowski. Detecting stealth software with strider ghostbuster. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, pages 368–377, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2282-3. doi: 10.1109/DSN.2005.39. URL http://dx.doi.org/10.1109/DSN.2005.39.

[63] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 13–13, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251375.1251388.

[64] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 1–1, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267359.1267360.

[65] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection and monitoring through vmm-based "out-of-the-box" semantic view reconstruction. *ACM Trans. Inf. Syst. Secur.*, 13(2):1–28, 2010. ISSN 1094-9224. doi: http://doi.acm.org/10.1145/1698750.1698752.

[66] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-810-7. doi: http://doi.acm.org/10.1145/1455770.1455779.

[67] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Vmm-based hidden process detection and identification using lycosid. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 91–100, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-796-4. doi: http://doi.acm.org/10.1145/1346256.1346269.

[68] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 477–487, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-894-0. doi: http://doi.acm.org/10.1145/1653662.1653720. URL http://doi.acm.org/10.1145/1653662.1653720.

[69] Ying Wang, Chunming Hu, and Bo Li. Vmdetector: A vmm-based platform to detect hidden process by multi-view comparison. In *Proceedings of the 2011 IEEE 13th International Symposium on High-Assurance Systems Engineering*, HASE '11, pages 307–312, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4615-5. doi: 10.1109/HASE.2011.41. URL http://dx.doi.org/10.1109/HASE.2011.41.

[70] Roberto Di Pietro and Luigi V. Mancini. *Intrusion Detection Systems*, volume 38 of *Advances in Information Security*. Springer-Verlag, 2008. ISBN 978-0-387-77265-3.

[71] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pages 156–161, Washington, DC, USA, 2001. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=882495.884434.

[72] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard A. Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267336.1267355.

[73] Kevin Scott and Jack Davidson. Safe virtual execution using software dynamic translation. In *Proceedings of the 18th Annual Computer Security Applications Conference*, ACSAC '02, pages 209–216, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1828-1. URL http://dl.acm.org/citation.cfm?id=784592.784801.

[74] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No.99CB36344)*, pages 133–145. IEEE Comput. Soc, 1999. ISBN 0-7695-0176-1. doi: 10.1109/SECPRI.1999.766910. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=766910.

[75] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[76] Daniela Oliveira, Jedidiah R. Crandall, Gary Wassermann, Shaozhi Ye, Shyhtsun Felix Wu, Zhendong Su, and Frederic T. Chong. Bezoar: Automated virtual machine-based full-system recovery from control-flow hijacking attacks. In *NOMS*, pages 121–128. IEEE, 2008. URL http://dblp.uni-trier.de/db/conf/noms/noms2008.html#OliveiraCWYWSC08.

[77] M. Laureano, C. Maziero, and E. Jamhour. Protecting host-based intrusion detectors through virtual machines. *Comput. Netw.*, 51(5):1275–1283, 2007. ISSN 1389-1286. doi: http://dx.doi.org/10.1016/j.comnet.2006.09.007.

[78] Xiantao Zhang, Qi Li, Sihan Qing, and Huanguo Zhang. Vnida: Building an ids architecture using vmm-based non-intrusive approach. In *WKDD*, pages 594–600. IEEE Computer Society, 2008. URL http://dblp.uni-trier.de/db/conf/wkdd/wkdd2008.html#ZhangLQZ08.

[79] Fatemeh Azmandian, Micha Moffie, Malak Alshawabkeh, Jennifer G. Dy, Javed A. Aslam, and David R. Kaeli. Virtual machine monitor-based lightweight intrusion detection. *Operating Systems Review*, 45(2):38–53, 2011. URL http://dblp.uni-trier.de/db/journals/sigops/sigops45.html#AzmandianMADAK11.

[80] Francesco Tamberi, Dario Maggiari, Daniele Sgandurra, and Fabrizio Baiardi. Semantics-driven introspection in a virtual environment. In *Proceedings of the 2008 The Fourth International Conference on Information Assurance and Security*, pages 299–302, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3324-7. doi: 10.1109/IAS.2008.17. URL http://dl.acm.org/citation.cfm?id=1437896.1438507.

[81] Wenke Lee, Salvatore J. Stolfo, and Philip K. Chan. Learning patterns from unix process execution traces for intrusion detection. In *In AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 50–56. AAAI Press, 1997.

[82] Rafael C. Carrasco and José Oncina. Learning stochastic regular grammars by means of a state merging method. In *Proc. Coll. on Grammatical Inference and Apps.*, pages 139–152, London, UK, 1994. Springer-Verlag. ISBN 3-540-58473-0. URL http://portal.acm.org/citation.cfm?id=645515.658099.

[83] Yongzhong Li, Yang Ge, and Xu Jing. A new intrusion detection method based on Fuzzy HMM. *IEEE Conf. on Industrial Electronics and Apps.*, pages 36–39, June 2008. doi: 10.1109/ICIEA.2008.4582476. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4582476.

[84] Wenqing Fan, Binbin Zhou, Hongliang Liang, and Yixian Yang. A novel program analysis method based on execution path correlation. In *Proc. Int. Symp. on Knowledge Acquisition and Modeling*, KAM '09, pages 178–181, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3888-4. doi: http://dx.doi.org/10.1109/KAM.2009.34. URL http://dx.doi.org/10.1109/KAM.2009.34.

[85] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proc. IEEE Symp. on Security and Privacy*, SP '07, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2848-1. doi: http://dx.doi.org/10.1109/SP.2007.17. URL http://dx.doi.org/10.1109/SP.2007.17.

[86] M L Thathachar and P S Sastry. Varieties of learning automata: an overview. *IEEE trans. on systems, man, and cybernetics*, 32(6):711–22, January 2002. ISSN 1083-4419. doi: 10.1109/TSMCB.2002.1049606. URL http://www.ncbi.nlm.nih.gov/pubmed/18244878.

[87] Oleg Mikhail Sheyner. *Scenario graphs and attack graphs*. PhD thesis, Pittsburgh, PA, USA, 2004. AAI3126929.

[88] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Multi-aspect profiling of kernel rootkit behavior. In *EuroSys '09: Proc. ACM european conference on Computer systems*, pages 47–60, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-482-9. doi: http://doi.acm.org/10.1145/1519065.1519072.

[89] Daniel Delling, Martin Holzer, K. Muller, Frank Schulz, and Dorothea Wagner. High-performance multi-level graphs. *9th DIMACS Challenge on Shortest Paths*, pages 1–13, 2006. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.1827&rep=rep1&type=pdf.

[90] Fabrizio Baiardi, Dario Maggiari, Daniele Sgandurra, and Francesco Tamberi. Transparent process monitoring in a virtual environment. *Electron. Notes Theor. Comput. Sci.*, 236:85–100, 2009. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2009.03.016.

[91] Anh M. Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T. King, and Hai D. Nguyen. Mavmm: Lightweight and purpose built vmm for malware analysis. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 441–450, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3919-5. doi: http://dx.doi.org/10.1109/ACSAC.2009.48. URL http://dx.doi.org/10.1109/ACSAC.2009.48.

[92] Rich Wolski. Eucalyptus. http://www.eucalyptus.com, 2009.

[93] Stealth@segfault.net. Phrack issue 61 kernel rootkit experiences. http://www.phrack.org, 2003.

[94] Nelson Murilo. Chrootkit. http://www.chrootkit.org, 2008.

[95] Phoronix. Phoronix test suite. http://phoronix-test-suite.com/, 2009.

[96] Cornell-University. Red cloud with MATLAB. http://www.cac.cornell.edu/wiki/index.php?title=Red_Cloud_with_MATLAB.

[97] Nimbis. Cloud services for mathematica. https://www.nimbisservices.com/marketplace/wolfram-research/mathematica-clouds.

[98] G. D'Angelo. Parallel and distributed simulation from many cores to the public cloud. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 14–23, 2011.

[99] Philip Church, Adam Wong, Michael Brock, and Andrzej Goscinski. Toward exposing and accessing HPC applications in a SaaS cloud. In *Proc. of the 2012 IEEE 19th International Conference on Web Services*, ICWS '12, pages 692–699, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4752-7.

[100] Philippe Golle and Ilya Mironov. Uncheatable distributed computations. In *Proceedings of the 2001 Conference on Topics in Cryptology: The Cryptographer's Track at RSA*, CT-RSA 2001, pages 425–440, London, UK, 2001. Springer-Verlag. ISBN 3-540-41898-9.

[101] Emiliano Casalicchio, Daniel A. Menascé, and Arwa Aldhalaan. Autonomic resource provisioning in cloud systems with availability goals. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, pages 1:1–1:10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2172-3.

[102] Vincent C. Emeakaroha, Marco A.S. Netto, Rodrigo N. Calheiros, Ivona Brandic, Rajkumar Buyya, and César A.F. De Rose. Towards autonomic detection of SLA violations in cloud infrastructures. *Future Generation Computer Systems*, 28(7):1017 – 1029, 2012. ISSN 0167-739X.

[103] Felipe S. Martins, Rossana M. Andrade, Aldri L. dos Santos, Bruno Schulze, and al. Detecting misbehaving units on computational grids. *Concurr. Comput.:Pract. Exper.*, 22(3):329–342, Mar 2010. ISSN 1532-0626.

[104] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In *Proceedings of the 30th annual conference on Advances in cryptology*, CRYPTO'10, pages 465–482, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14622-8, 978-3-642-14622-0.

[105] Andrew O. Makhorin. GLPK GNU Linear Programming Kit. http://www.gnu.org/software/glpk/glpk.html.

[106] Amazon. Amazon web services. http://aws.amazon.com.

[107] Adam Groce, Jonathan Katz, Aishwarya Thiruvengadam, and Vassilis Zikas. Byzantine agreement with a rational adversary. In *Proc. of the 39th Intl. colloquium conference on Automata, Languages, and Programming*, ICALP'12, pages 561–572, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31584-8.

[108] Reid Kerr and Robin Cohen. Smart cheaters do prosper: defeating trust and reputation systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, pages 993–1000, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-0-9817381-7-8.

[109] Sung Il Kim, Jae Young Jun, Jong-Kook Kim, Kyung-Chan Lee, Gyu Seong Kang, and al. Dynamic resource management for a cell-based distributed mobile computing environment. In *Proceedings of the 8th international conference on Ubiquitous intelligence and computing*, UIC'11, pages 174–184, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23640-2.

[110] CSA. Cloud Security Alliance. http://www.cloudsecurityalliance.org/.

[111] Heinrich Moser. Towards a real-time distributed computing model. *Theor. Comput. Sci.*, 410(6-7):629–659, feb 2009. ISSN 0304-3975.

[112] Flavio Lombardi, Roberto Di Pietro, and Claudio Soriente. CReW: Cloud resilience for Windows guests through monitored virtualization. In *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, pages 338–342, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4250-8.

[113] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. In *Proceedings of the 43rd annual ACM symposium on Theory of computing*, STOC '11, pages 363–372, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0691-1.

[114] Eunyoung Cheon, Mikyoung Kim, Seunghak Kuk, and Hyeon Soo Kim. A regional matchmaking technique for improving efficiency in volunteer computing environment. In *Proceedings of the 2011 First ACIS/JNU International Conference on Computers, Networks, Systems and Industrial Engineering*, CNSI '11, pages 285–289, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4417-5.

[115] Pedro Costa, Marcelo Pasin, Alysson N. Bessani, and Miguel Correia. Byzantine fault-tolerant mapreduce: Faults are not just crashes. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 32–39, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4622-3.

[116] F. Koeppe and J. Schneider. Do you get what you pay for? using proof-of-work functions to verify performance assertions in the cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second Intl. Conference on*, pages 687 –692, 30 2010-dec. 3 2010.

[117] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '92, pages 139–147, London, UK, UK, 1993. Springer-Verlag. ISBN 3-540-57340-2.

[118] Raluca Ada Popa, Jacob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang. Enabling security in cloud storage slas with cloudproof. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 31–31, Berkeley, CA, USA, 2011. USENIX Association.

[119] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, January 2010. ISSN 0734-2071.

[120] SiYuan Xin, Yong Zhao, and Yu Li. Property-based remote attestation oriented to cloud computing. In *Proceedings of the 2011 Seventh International Conference on Computational Intelligence and Security*, CIS '11, pages 1028–1032, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4584-4.

[121] Yi Luo and D. Manivannan. Hope: A hybrid optimistic checkpointing and selective pessimistic message logging protocol for large scale distributed systems. *Future Gener. Comput. Syst.*, 28(8):1217–1235, October 2012. ISSN 0167-739X.

[122] Wolfgang Ahrendt and Maximilian Dylla. A system for compositional verification of asynchronous objects. *Sci. Comput. Program.*, 77(12):1289–1309, October 2012. ISSN 0167-6423.

[123] Amos Beimel, Yehuda Lindell, Eran Omri, and Ilan Orlov. 1/p-secure multiparty computation without honest majority and the best of both worlds. In *Proc. of the 31st annual conference on Advances in cryptology*, CRYPTO'11, pages 277–296, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22791-2.

[124] Tang Xiaoyong, Kenli Li, Zeng Zeng, and Bharadwaj Veeravalli. A novel security-driven scheduling algorithm for precedence-constrained tasks in heterogeneous distributed systems. *IEEE Trans. Comput.*, 60(7):1017–1029, jul 2011. ISSN 0018-9340.

[125] Lingfang Zeng, Bharadwaj Veeravalli, and Xiaorong Li. Scalestar: Budget conscious scheduling precedence-constrained many-task workflow applications in cloud. In *Proceedings of the 2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, AINA '12, pages 534–541, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4651-3.

[126] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proc. of the 34th IEEE Symp. on Security and Privacy*, 2013.

[127] Albert A. Groenwold. Positive definite separable quadratic programs for non-convex problems. *Struct. Multidiscip. Optim.*, 46(6):795–802, dec 2012. ISSN 1615-147X.

[128] S.P. Muralidharan and V.V. Kumar. A novel reputation management system for volunteer clouds. In *Computer Communication and Informatics (ICCCI), 2012 International Conference on*, pages 1–5, 2012.

[129] Michele Amoretti, Francesco Zanichelli, and Gianni Conte. Efficient autonomic cloud computing using online discrete event simulation. *J. Parallel Distrib. Comput.*, 73(6): 767–776, jun 2013. ISSN 0743-7315.

[130] Salim Hariri, Mohamed Eltoweissy, and Youssif Al-Nashif. Biorac: biologically inspired resilient autonomic cloud. In *Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '11, pages 80:1–80:1, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0945-5.

[131] Michele Amoretti, Alberto Lluch Lafuente, and Stefano Sebastio. A cooperative approach for distributed task execution in autonomic clouds. *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, 0:274–281, 2013. ISSN 1066-6192.

[132] Abhijeet Gaikwad and Ioane Muni Toke. Parallel iterative linear solvers on GPU: A financial engineering case. In *Proc. of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 607–614, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-3939-3.

[133] Flavio Lombardi and Roberto Di Pietro. CUDACS: securing the cloud with CUDA-enabled secure virtualization. In *Proceedings of the 12th international conference on Information and communications security*, ICICS'10, pages 92–106, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-17649-6, 978-3-642-17649-4.

[134] A. Di Biagio, A. Barenghi, G. Agosta, and G. Pelosi. Design of a parallel aes for graphics hardware using the cuda framework. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –8, may 2009. doi: 10.1109/IPDPS.2009.5161242.

[135] N. Nishikawa, K. Iwai, and T. Kurokawa. High-performance symmetric block ciphers on CUDA. In *Networking and Computing (ICNC), 2011 Second International Conf. on*, pages 221 –227, 30 2011-dec. 2 2011.

[136] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3):451–460, jun 2010. ISSN 0163-5964.

[137] Cray. Titan accelerated computing. http://www.olcf.ornl.gov/titan, 2012.

[138] Zillians. VGPU GPU virtualization. http://www.zillians.com/products/vgpu-gpu-virtualization/, 2012. Last accessed 02/27/2014.

[139] Michael Larabel. NVIDIA Linux driver hack gives you root access. http://www.phoronix.com/scan.php?page=news_item&px=MTE1MTk, 2012. Last accessed 02/27/2014.

[140] V.V. Kindratenko, J.J. Enos, Guochun Shi, M.T. Showerman, G.W. Arnold, J.E. Stone, J.C. Phillips, and Wen-Mei Hwu. GPU clusters for high-performance computing. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–8, 2009. doi: 10.1109/CLUSTR.2009.5289128.

[141] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 33, New York, NY, USA, 2004. ACM. doi: http://doi.acm.org/10.1145/1103900.1103933.

[142] Nvidia. CUDA 4.2 developers guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/doc/CUDA_C_Programming_Guide.pdf, 2012.

[143] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):105–118, jan 2011. ISSN 1045-9219.

[144] Donald Evans, Phillip Bond, and Arden Bement. FIPS pub 140-2 security requirements for cryptographic modules, 1994.

[145] Henk C. A. van Tilborg and Sushil Jajodia, editors. *Encyclopedia of Cryptography and Security, 2nd Ed*. Springer, 2011. ISBN 978-1-4419-5905-8.

[146] Rebecca T. Mercuri and Peter G. Neumann. Security by obscurity. *Commun. ACM*, 46 (11):160–166, nov 2003. ISSN 0001-0782.

[147] Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. McKinley. Why nothing matters: the impact of zeroing. *SIGPLAN Not.*, 46(10):307–324, oct 2011. ISSN 0362-1340.

[148] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. SSLShader: cheap SSL acceleration with commodity processors. In *Proc. of the 8th USENIX conference on Networked systems design and implementation*, NSDI 11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[149] Howard M. Heys. A tutorial on linear and differential cryptanalysis. *Cryptologia*, 26(3): 189–221, jul 2002. ISSN 0161-1194.

[150] Marco Riccardi, Roberto Di Pietro, Marta Palanques, and Jorge Aguilí Vila. Titans' revenge: Detecting zeus via its own flaws. *Comput. Netw.*, 57(2):422–435, 2013. ISSN 1389-1286. doi: 10.1016/j.comnet.2012.06.023. URL http://dx.doi.org/10.1016/j.comnet.2012.06.023.

[151] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and al. Demystifying GPU microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 235 –246, march 2010.

[152] Nick Black and Jason Rodzik. My other computer is your GPU: System-centric CUDA threat modeling with CUBAR. http://nick-black.com/dankwiki/images/d/d2/Cubar2010.pdf, 2010. Last accessed on 02/27/2014.

[153] Shinpei Kato. Gdev. https://github.com/shinpei0208/gdev, 2012. Last accessed 02/27/2014.

[154] Wu chun Feng and Shucai Xiao. To GPU synchronize or not GPU synchronize? In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 3801 – 3804, 30 2010-june 2 2010.

[155] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A. Connors. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Trans. Dependable Secur. Comput.*, 6(2):135–148, apr 2009. ISSN 1545-5971.

[156] Alessandro Barenghi, Gerardo Pelosi, and Yannick Teglia. Information leakage discovery techniques to enhance secure chip design. In Claudio Ardagna and Jianying Zhou, editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, volume 6633 of *LNCS*, pages 128–143. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-21039-6.

[157] Joel Reardon, David Basin, and Srdjan Capkun. Sok: Secure data deletion. In *Proc. of the IEEE Symposium on Security and Privacy*, SP '13, pages 301–315, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4.

[158] Shi Guochun. Cuda wrapper library. http://cudawrapper.sourceforge.net, 2012. Last accessed 02/27/2014.

[159] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Confidentiality issues on a GPU in a virtualized environment. In *FC 18th International Conference on Financial Cryptography and Data Security*, Barbados, BARBADOS, 2014. URL http://www.eurecom.fr/publication/4205.

[160] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proceedings of the*

2011 USENIX conference on USENIX annual technical conference, USENIXATC'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2002181.2002183.