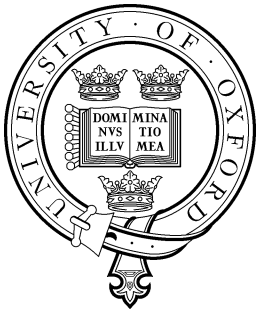
Trusted Infrastructure for the Campus Grid



Cornelius Namiluko

Wolfson College

University of Oxford

A thesis submitted in partial fulfillment towards the degree of

Master of Science
Computer Science

Trinity 2008

Statement of Originality

I, the undersigned, declare that the work presented in this thesis is sorely my own and that I have clearly acknowledged any ideas that I borrowed from other authors as well as the assistant I received from other people.

Signature:.....

Acknowledgements

My heart-felt thanks goes to my wife, Tsilazazi, and my daughter, Michelle Limpo, for all their support and encouragement throughout the year and more especially when I was working tirelessly on this dissertation. This work wouldn't have been the same without you!

I would also like to give a big thanks to my supervisor, Dr. Andrew Martin and to Dr. David Wallom for all the support and guidance rendered to me during this dissertation and for giving me an opportunity to work with them.

Lastly, but certainly not the least, thanks to all my friends.

Abstract

The University of Oxford Campus Grid, OxGrid, is a collection of Condor Pools and clusters that are inter-connected to provide high computational and storage capacity to users. Users submit their long-running tasks to the Grid, which uses Condor software to split the tasks into jobs and manages which hosts on the Grid execute the jobs. Users expect their jobs and its data to maintain integrity and confidentiallity but have no way of determining the state of the hosts that executed their jobs. The consequence is that users cannot be guaranteed that their jobs were not stolen or that the results are entirely correct. In addition, the credentials that are used in authentication are only protected using passwords and thus risk being compromised because passwords can be phished or broken using dictionary attacks. This has the consequence that malicious code could use the credentials and harm not only the user of the compromised credentials but also cause disruption to services on the Grid.

Trusted computing provides the ability to take integrity measurements of a system and a secure storage of credentials used in authentication so that the credentials only get exposed if the state of the system is correct. Additionally, trusted computing offers a way for one system to check the state of another and thereby a means of checking the trustworthiness of the other system.

This thesis investigates the security threats involved with the use of Condor in a production environment such as OxGrid and makes a significant contribution towards improving them identified security threats by using trusted computing technology. It also looks at the practicallity and intellectual challenge of improving the security of a functional system, Condor, by using new technology.

Contents

List of Figures

Chapter 1

Introduction

Grid computing is a phenomenon that has become very popular among academic and research institutions in the world. A number of universities in the UK already own grid computing infrastructure comprising of tens to hundreds and even thousands of personal computers and dedicated computing clusters. In the research circles a number of notable projects, including ClimatePrediction [8], an experiment that tries to produce the forcast of the climate in the 21st Century uses idle time of computers around the world to run simulations, SETI@Home [39], a project that performs complex computations using hundreds, thousands, or even millions of machines on the Internet to analyse data for use in the search for evidence of intelligent life elsewhere in the universe, have been able to successfully deploy grid computing technology in their work.

A common feature among the current grid computing environments is that they are based on a model that places unverifiable trust on the interacting parties. For example in an academic environment, it is assumed that users of grid resources will not intentionally execute malicious code on the grid while users trust that the grid will operate in the expected manner maintaining the confidentiality and integrity of their jobs. Similarly in research projects such as climateprediction, those contributing resources are forced to trust that the software downloaded to their machines is not malicious. The biggest problem with working on this kind of trust symmetry is that users are forced into trusting the results obtained from such systems to be genuine and coming from a secure system (that did not compromise the integrity and confidentiality of the data) and not a rogue system impersonating the genuine grid resource, providing false answers, or steal the data being processed. This problem is generally unsolved in grid computing and calls for a need for verifying the trust symmetry and allowing interactions only when parties involved are happy with each

other's state.

Trusted computing technology provides a solution that allows one system to determine the state of another and thus decide whether to believe both its identity and its computational results. It also offers a more secure storage of the most sensitive data such as private keys and prevents malicious code from making use of these secrets by only releasing them when the system is in the state it was at the time they were protected.

1.1 Introducing Grid Computing

Foster and Kesselman [46] defined the concept of a Grid as "a very large scale, generalised distributed NC (Network Computing) system that can scale to Internet-size environments with machines distributed across multiple organizations and administrative domains." The original idea of a grid was analogous to power grids. The creators imagined computers connecting to the grid in the same way that we plug in electrical appliances to power sockets so that users would not know which computer performs the computations in the same way that users of power systems are not aware about the part providing them electricity they use [10]. This analog defines one very important property of grid computing which is still very much observed, that is, the owners of the resources on the grid maintain control and can "plug in or out" their resources to the grid at any time without affecting the computations being processed on the grid. Today, grid environments still maintain this principle and add functionality such as the execution of jobs on idle resources, single sign-on and resource balancing, to allow the scheduling of grid jobs on machines with low utilisation [17] and therefore provides high computing resources, such as storage, computational power and access to shared data to users demanding for it.

A good example of an application of grid computing is the Oxford University's campus grid, OxGrid [51], discussed below.

1.2 Oxford University Campus Grid: OxGrid

The OxGrid is a result of a project undertaken by the Oxford University e-Research centre, which began as a response to the increasing demand for high computation and storage capacity and the institution's desire to maintain its leading position in

research and academics [51]. OxGrid offers a variety of computation and data storage resources to the users around the university thus providing them with the ability to perform compute intensive tasks over the high throughput network. It also provide users with a single sign-on facility that enables them to sign-in and access multiple resources without repeatedly providing their credentials. The idea is that computing resources within the University are under utilised and continue to lose their value everyday and so by using the fraction of time in which they would normally be idle, high throughput computing would be achieved while maintaining the costs very low since no new computers would need to be acquired.

OxGrid is comprised of thousands of computers from departments and colleges and will continue to grow as more departments and colleges join the grid. It allows users from around the university to prepare their long running tasks and uses its resource management system to run the tasks on any machine from a department, college or from the supercomputing facilities offered by the OSC (Oxford Super computing Centre) [26] or the National Grid Service (NGS) [24].

At the heart of OxGrid lies a resource broker in the name of Condor-G. "Condor-G combines the inter-domain resource management protocols of the Globus Toolkit and the intra-domain resource and job management methods of Condor to allow the user to harness multi-domain resources as if they all belong to one personal domain" [31]. What this means is that users simply submit their tasks to OxGrid which then uses Condor-G to split them into jobs and decide which available resource to assign the jobs.

Figure 1.1 illustrates the concept of OxGrid. Each of the department or college runs what is referred to as a Condor pool (a collection of resources each running the Condor [30] software and controlled by one machine, referred to as the Central Manager) and the pools are then controlled by a resource manager that also runs Condor.

1.3 OxGrid User Expectation

Imagine a researcher who needs to carry out a set of experiments that would take her months to obtain results when run on desktop computer. OxGrid offer resources to such a researcher that would significantly reduce the run-time of her experiments.

Figure 1.1: The Concept of OxGrid [51]

Now, imagine further that the researcher considers her experiments or the experimentor data to be highly sensitive and also that she cannot afford to have wrong results. This places a requirement on OxGrid, that is to ensure that the confidentiality and integrity of the data is maintained. OxGrid's idea of splitting a user's task into small jobs and executing them using different resources on the grid raises a number of questions such as: did some rogue system impersonate the genuine grid resources and provide false results, or steal the data being processed? Did some legitimate system fall victim to a rogue administrator or a Trojan? So how can the researcher be assured that their data was kept only to the resources performing the operation and that the resources used did not retain a copy of the data after completing the operation? How can OxGrid maintain the integrity and confidentiality of the jobs without affecting its flexibility and usability?

1.4 The need for trusted infrastructure

Trusted computing offers a means by which users can check how trustworthy the machines on the grid are and use that to decide whether to allow the machines to execute their jobs or not. It allows one system to check what software is running on a remote machine and thus determine whether that system can be expected to behave correctly. By using trusted computing technology OxGrid would be equiped with mechanisms to evalute the trustworthiness of a particular resource before jobs can be passed to it. Additionally, trusted computing provides a more secure storage of credentials and a way to measure the state of the platform so that credentials only get released when a platform is in the same as it was at the time they were protected.

This thesis, "Trusted Infrastructure for the Campus Grid", looks at the security

threats assocciated with grid computing technology, in particular the OxGrid's use of the technology and how these can be improved using trusted computing technology.

1.5 Objectives

The security and trustworthy of OxGrid, and any grid computing system in general, greatly affects how fast it gets adopted by departments, colleges, corporating institutions and by users. To ensure a positive adoption rate requires all reasonable steps that can potentially improve security to be investigated. This project seeks to investigate the security threats associated with OxGrid and how these can can be addressed using trusted computing. The main objectives set are as follows:

- develop an understanding of the operation of resource brokerage systems, in particular the Condor system, its security protocols and the security threats associated with its use of the protocols and thus identify the parts of Condor that need to be reworked to solve the threats.
- To investigate the capabilities of trusted computing with the view of developing a mechanism that can be used to improve the identified security lapses.
- design a solution that uses trusted computing to solve the identified security threats. This will not only produce the design of a solution to this particular domain but will also develop an approach that can be used to improve the security of existing systems
- Cryptophically identify the nodes to the central manager, ensure that the nodes are running condor and not some malicious code.
- and finally demonstrate the intellectual challenges and practicallity of making old systems more secure by using new technology

It is important to mention that this work is not meant to design a solution that is free from attacks as this is impossible to achieve but seeks to improve the security of the campus grid.

1.6 Motivations

The field of computer security seeks to understand what threats are associated with a given computer system, what amount of resources an attacker needs and then to

put in place measures that make it infeasible for an attacker to carry out the attack. Infeasibility does not however mean impossibility and therefore, computer security seeks to make it harder for potential attackers to carry out an attack. Beckles [2] notes that there has been no real attempt to attack grid computing (Condor is his case) or to use it to carry out an attack and therefore we cannot tell what weaknesses exist or that the measures in place are enough. Instead we should be focusing on exploring all possible attacks on the grid (or any other systems) and make it harder for potential attackers. This thesis is a good example of such a measure.

The main motivation for this project is the interest I have in distributed computing and computer security. It is also driven by my anxiety to see how much intellectual capacity is required to make a functional system more secure. I learnt from the computer security course that one of the difficult tasks in ensuring a secure environment is the storage and distribution of cryptographic keys. I also learnt the security of a system depends on how secure its security credentials are kept and that compromising the credentials may result in the loss of integrity and confidentiality of the information being protected. Seeing that trusted computing promises a more secure storage of credentials motivates me to investigate how how this can be applied to the campus grid. In addition, this work is not only a challenge but is also exciting as it provides me with an opportunity to exercise the knowledge I have gained from my Master of Science course.

1.7 Thesis Outline

This thesis starts by first looking at the resource and job scheduling system used in OxGrid, Condor System in chapter 2 to provide the reader with an insight into its operation, with an emphasis on the protocols used to provide confidentiality and integrity of computations. Chapter 3 identifies security threats within the Condor pools of OxGrid and seeks to define the main problem that the project seeks a solution for. Chapter 4 provides an insight into trusted computing technology, focusing on the main concepts applicable to the problems described in chapter 3 and the relevancy of the technology to the problems stated. Chapter 5 takes a look at the design of a solution to the problems stated in the threat model chapter. It takes an incremental approach and discusses the design decisions made at each stage, the reasons for the choice and any assumptions made. Chapter 6 is a discussion of implementation details of the prototype developed using the design. Chapter 7 presents an evaluation of the

design and implementation focusing on how the implementation compares with the design and the weaknesses of the design. It also discusses some of the work that is envisioned on the campus grid before presenting. Chapter 9 prsents the conclusion of the thesis.

Chapter 2

Condor: Grid Enabler for OxGrid

OxGrid is composed of resources from departments, colleges and dedicated computing clusters from the Oxford Supercomputing Center (OSC) [26] and the National Grid Service (NGS) [24] and offers computational and data storage facilities to users from the university. Each of the departments and colleges run what is referred to as a Condor pool, which is a mixture of inter-connected resource requests and resource offerings and, more importantly, all of which are connected to a machine with a special role called a central manager. It is referred to as a Condor pool because each of the machines uses the Condor Software as an interface to the Grid, for submitting and managing jobs. The Condor software is also used at the central management center to link the pools and the clusters into a grid and hence the Condor software can be referred to as the grid enabler for OxGrid.

Each pool is setup with one central manager which allocates resource needs to the available resource offerings. At the same time, each central manager communicates

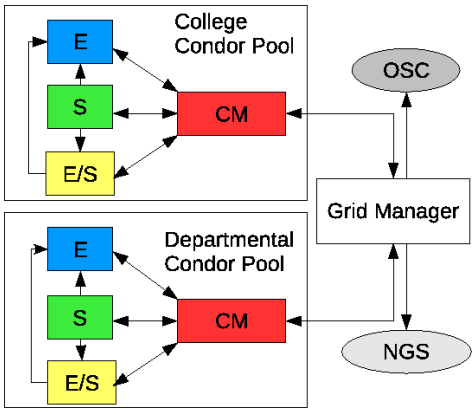


Figure 2.1: OxGrid setup showing the Condor Pools

with central managers on other pools to send jobs to resources on their pools if more resources are needed beyond what the pool can offer, illustrated in figure 2.1. Within the pool, the resource requests originate from the user, through entities called submit machines (S) and are serviced by entities playing the role of execute machine (E). This is enabled by Condor software which, depending on the components installed, allows the machine to perform the role of an execute machine, a submit machine, a combination of execute and submit, or a central manager.

The Condor software provides methods for job submission and allocation, error recovery and a friendly execution environment and interact with the Globus toolkit [33], which provides protocols for secure inter-domain communication, thereby producing a system, OxGrid, that provides an interface to computational resources that spans across multiple domains [47].

This chapter presents a brief description, refer to the Condor Manual [28] for more detailed information, of the components that are relevant to the understanding of the operation and security protocols used within Condor. This knowledge is essential for the identification of the security threats within the pools in the campus grid, discussed in the next chapter.

2.1 Definition of Condor

"Condor is a specialized workload management system for compute-intensive jobs" [30]. Tannenbaum et al [49][50] defined it as a high-throughput distributed batch computing system that provides job management mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. It allows users to submit their jobs and manages when and where to run them based upon a policy, monitors their progress, and ultimately informs the user upon completion [48]. It is very flexible and provides users with the ability to retain control for their resources so that they can decide just how much to contribute to the grid.

2.2 Condor Components

The following sections describe some of the components of Condor and what they do.

2.2.1 Condor Daemons/Services

Condor uses what are called daemons to carry out its operations. Each daemon is implemented to serve a specific functionality and runs independently from other daemons. This means that each daemon can be started, stopped or restarted without affecting others running on the same machine. The daemons that run on a given machine determines what services a machine can provide and thus its role within the pool, discussed above. The following sections briefly describe some of the daemons in Condor.

condor_master

The condor_master, most often referred to simply as master, is responsible for starting, restarting and monitoring all other daemons running on a host and also keeps the system administrator informed of any failures (such as a daemon failing to start or stopped) in the system via email [28]. For this reason, the master is crucial to the operation of Condor and has to operate in the specified manner otherwise the entire installation would not work as expected.

condor_startd

The resourcs offerings need to correctly specify the resources they are offering together with a policy specifying who can use them and for what. This task is handled by the condor_startd daemon, also called startd. The startd has algorithms built in it that allows it query a platform for the description of its resources such as the type of operating system, amount of storage capacity and the amount of Random Access Memory (RAM). This information is collected, placed into structures called classads, discussed below, and sent to a central manager [28], as discussed above. The startd also enforces the policies specified by the resource owner and starts the condor_starter daemon, discussed below, when the resources have been matched with some request.

condor_starter

The condor_starter runs as a child process of the startd daemon. It represents an actual job running on a machine and is therefore required on any machine that will execute jobs. It is responsible for setting up the necessary environment for the execution of the job, monitoring the status of the job [28] and sending updates to the shadow daemon, discussed below, about the progress of the job. It also services any requests that need to be performed on the job, such as stopping it.

condor_schedd

While the startd represents resource offerings, the condor_schedd, or schedd, represents resource requests. In the same way that the startd collects resource information and sends to the central manager, the schedd collects information about the resources required for a given job and sends to the central manager. The schedd daemon manages the jobs submitted by the users and when the resource meeting the requirement of a particular job is found, the schedd arranges to send the job to the matched resource offer and creates the condor_shadow daemon, discussed below. The schedd also remains as a contact point for the user allowing them to view or manipulate their jobs [28].

condor_shadow

The condor_shadow, or just shadow, daemon represents a resource request that has been matched and is being serviced by some resource offer. When a user submits a job, the schedd awaits for a match and once one is found starts the shadow daemon for the particular job. The shadow process maintains a handle to the resource executing the job to get updates on the progress of the job. It also decides where to store any files required for the job [28].

condor_collector

The condor_collector, also called collector keeps track of the status of every host within a pool by receiving updates from all other daemons describing the state of the daemons and the state of the resource on which they reside [28]. As a result all other daemons, as well as users, can query the collector to find out information about the status of the resources on a pool.

condor_negotiator

The condor_negotiator daemon, or simply negotiator, is responsible for matching the resource requests to the available resource offers. It does this by querying the collector for the state of the resources, assigns priorities to the resource requests and assigns the requests to the available resource offerings with the best possible match, a process called match-making described in section 2.5.

Other Daemons

Condor also includes other daemons such as condor_kbdd, for determining if a host is idle and the condor_ckpt_server daemon for processing checkpoint files [28].

2.2.2 Classads

Classad are a means by which machines using Condor specify the policy for matching the machines that can either make use of their resource or offer resources to them. It is implemented as a schema-free resource allocation language and is used as a framework for matching resource requests with resource offers [48]. "Condor's ClassAds are analogous to the classified advertising section of the newspaper. Sellers advertise specifics about what they have to sell, hoping to attract a buyer. Buyers may advertise specifics about what they wish to purchase. Both buyers and sellers list constraints that need to be satisfied" [29]. A resource offering machine would specify information such as the operating system, RAM, CPU type and speed, virtual memory size, the type of jobs as well as the conditions under which it is willing to run a job [29]. While a resource request would its requirements from the least restrictive, such as any available machine, to the most restrictive that includes a combination of the various specifications.

Classads are also used to specify the policies for the software required for jobs. This is specified on the submit machine by stating:

```
Requirements = (SOFTWARE_NAME =?= True)
```

where SOFTWARE_NAME indicates the name of the software package required for the job [32]. The execute machine states the availability of the software using the startd's expressions [32]:

```
STARTD_EXPRS = SOFTWARE_NAME
```

Signed Classads

Classads may be read by anyone but pose a problem in that an intruder may modify its contents especially if sent in plane text, which is the case by default. An intruder sitting between a machine and the central manager can get hold of the classad by sniffing the network, modify it and then forward it to the central manager. This can cause denial of service when the policies in the classads are modified. Condor's solution to this problem is the concept of signed classads, where a machine, A, creates

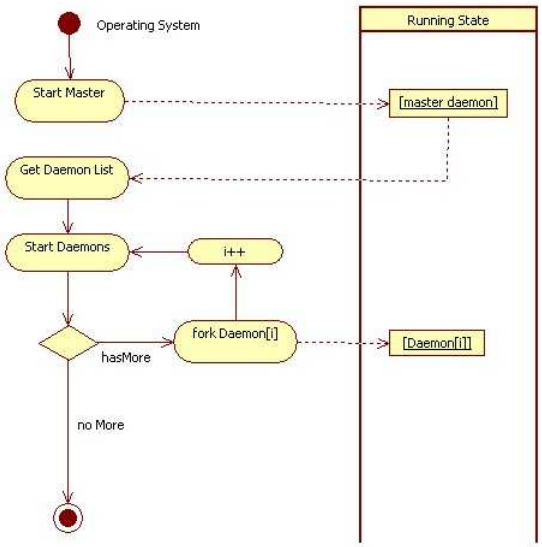


Figure 2.2: Condor startup process

and digitally signs the classad before sending it to the central manager, which can then verify A's signature with a third party and thus be sure about the origin of the classad. So that an intruder intercepting the communication would not modify the classad because she cannot fake A's signature.

2.3 Condor Startup Process

Each Condor daemons is an executable file that runs in a process that is linked to its parent process. As described above, the master daemon is responsible for starting all the other daemons and therefore is started first. The master daemon then reads the configuration files to determine the list of daemons that need to be started. This results in a DAEMONLIST which the master goes through to start each daemon by first determining the location of the executable, forking a child process and then executing it. This is illustrated in figure 2.2. When a daemon starts successfully, it goes into the running state where it can either initiate communication or respond to communication initiated by another daemon. Other daemons not specified in the DAEMONLIST can be started by a parent daemon when needed.

2.4 Daemon Interaction

In order to carry out any particular activity, the daemons communicate among themselves and exchange information through some defined protocols.. Each communica-

tion is defined with two entities, the client and a daemon. Where the client is the one that requests a service from a daemon [28]. This section describes an example interaction in a Condor pool within OxGrid, to show the daemons involved at each stage.

Let there be a user U with job j sitting on machine S, a central manager CM, and another machine E that is willing to have its resources used by other machines on the pool. S is configured to run the schedd daemon to allow it to submit jobs and keeps a job queue, Q, CM requires to run both negotiator and collector daemons while E requires startd. Each of the machines also runs the master daemon. The following is a description of what happens when a user submits there job, j.

1. U creates a description file for the job that includes the kind of environment required to execute the job, amount of resources such as RAM or disk space

2. U invokes the condor_submit command passing as an argument the description file created in step 1. The submit command acts as a client in a communication with the schedd, running on S. The schedd services the submit command and places j in a job queue Q.

3. The schedd, on S, initiates a communication with the Collector on CM. The collector on CM acts as a daemon in the communication and receives the resource request from schedd

4. Meanwhile, E independently queries the platform its running on for a description of its resources, creates a classad and sends to CM. Again the collector acts as a daemon in this communication with the startd on E as the client.

5. The negotiator deamon, on CM, performs the matching between the request from S and the offer from E as described in section 2.5.

6. Schedd initiates a communication with startd to claim the resource and if everything goes well, schedd starts up a shadow daemon to monitor the progress while startd starts up a starter daemon to carry out the execution of the job, j, and to update the shadow about the progress on the job.

7. Once the job is finished, shadow collects the results from starter and sends them to its parent, that is the schedd that started it and the schedd passes the results back to the user.

At each stage in the communication, the daemons may be required to authenticate to each other and generate sessions keys depending on whether authentication and integrity has been enabled.

2.5 Match-making process

The Condor match-making is a four-step process (advertisements through classads, matching algorithm, matching notification and claiming) that is performed by the machine playing the role of a central manager (called a matchmaker) to match the resource needs to resource offerings. It starts with the user who specify requirements (preferences and constraints) for their jobs and upon submission, the submit machine sends a classads to the matchmaker. The matchmaker also receives periodic updates about the state of all the other machines on the pool, through classads, but this time describing the resources available. It then performs the negotiation process in which each requirement is evaluated in the context of the job and the possible machine that can execute it [36]. Once a match is found the matchmaker informs both the requesting machine and the offering machine by sending them an authorisation ticket [19] which the resource requester can use to claim the resource.

2.6 Condor Security Model

Condor uses various mechanisms to ensure that data and jobs performed on a Condor pool maintains both confidentiality and integrity. It allows users and hosts to be correctly identified using authentication and controls who has access to a particular resource using various authorisation levels. Condor security model lies on the idea of performing checks at different points to determine whether access should be granted for a given request or to prevent subversion of the system [29]. To achieve this, it makes uses of various security protocols to enable secure inter-domain communications and standardized access to a variety of remote batch systems [48]. The security features exposed by condor are provided through a message-based communication library developed by the Condor team called CEDAR [48]. The CEDAR library allows commands to be sent via TCP/IP (Transport and communication Protocol/Internet Protocol) or UDP (User Datagram Protocol) and provides authentication, session management and encryption functionalities.

2.6.1 Authentication in Condor

Authentication is the process of establishing that an entity is who they say they are. Various forms of authentication exists such as the use of username/password combination or the use of public cryptography where the users are issued with a public/private key pair and a certificate by a trusted entity. Condor supports various authentication protocols including Kerberos [40] and GSI [34] referred to as the strong authentication methods in Condor [29], described below, Secure Socket Layer (SSL), password, Windows Authentication, File System and File system restore.

Grid Security Infrastructure (GSI)

Grid Security Infrastructure, GSI, is a security protocol that is based on public key infrastructure. It was developed as part of the Globus Toolkit by the globus project [33]. GSI operate using certificates encoded in the X.509, a standard data format for certificates established by the Internet Engineering Task Force [16], where each user or service that requires to be identified is issued an X.509 certificate indicating their name, their public key, a validity period and an identity and signature of the issuing authority, normally a certification authority (CA) as well as a private key corresponding to the public key in the certificate. To Authenticate a client, say A, sends her certificate (composed of her identity (id_A), her public key (PBKa), the identity of the certification authority (id_{ca}) and the signature of the certification authority (sigca)) to a server, say B. B then examines the certificate to establish whether it is valid and whether it was issued by a CA which she trusts. Once B has established the validity of the certificate, she needs A to prove that they have the stated identity. So B generates a message (m) together with a nonce (n_b) and sends to A asking her to encrypt it. A encrypts the message using her private key (PRKa) and sends to B. B decrypts the encrypted message using A's public key and if this works, A would have proved her identity to B. This could be summarised as follows.

$1 : A \rightarrow B : id_A, PBKa, id_{ca}, sigca$
$2 : B \rightarrow A : m.n_b$
$3 : A \rightarrow B : \{m\}_{PRKa}$

GSI uses Secure Socket Layer to perform mutual authentication, in which both communicating parties authenticate each other. This means that the above process would be repeated but with B trying to prove its identity to A.

GSI expects the private key and certificates to be placed in a directory and be encrypted with a password to prevent other entities from using them [34]. To make use of GSI in Condor, the certificates have to be placed in a directory controlled by the setting GSI_DAEMON_CERT or X509_USER_CERT and the key placed in GSI_DAEMON_KEY or X509_USER_KEY. Condor can then be configured to pass these credentials to GSI when authentication is requested. In which case establishing whether the issuing CA is trusted is performed by checking the list of CAs' signatures in the directory specified by the setting GSI_DAEMON_TRUSTED_CA_DIR.

Kerberos

Kerberos is an authentication protocol that makes use of a trusted third party, a kerberos server, to verify the identity of a user or service [40]. The third party has to be trusted so that authenticating parties can believe the identities it verifies. The original protocol was designed by Needham and Schroeder and it has been revised many times to make it more secure, for example, timestamps have been added to prevent the messages from being stolen and replayed later [40]. The Kerberos server keeps a record of all the entities together with their private keys, which it distributes to the owners, at the time of registration. Kerberos provides authentication of both entities such as users or services as well as the messages that they send over the network. It also provides encryption, based on DES [25] as well as an extension to the DES Cipher Block Chain (CBC), called propagating CBC, in which errors are propagated throughout the entire messages rather than restricted to a particular block in the message [40].

To carry out authentication, the entities referred to as the principal, are given names in a specified format consisting of the primary name, instance name and a kerberos realm [40] and are also given private keys which will only be known to the Kerberos server and the assigned principal. Authentication proceeds as follows:

$1 : C \rightarrow Ker : \{c, tgs\}K_c$
$2 : Ker \rightarrow C : \{\{c, addr_c, tgs, t, l, K_{c,tgs}\}K_{tgs}, K_{c,tgs}\}K_c$
$3 : C \rightarrow TGS : \{c, addr_c, tgs, t, l, K_{c,tgs}\}K_{tgs}, \{s\}K_{c,tgs}$
$4 : TGS \rightarrow C : \{\{c, addr_c, s, t2, l2, K_c, s\}K_s, K_{c,s}\}K_{c,tgs}$
$5 : C \rightarrow S : \{c, addr_c, s, t2, l2, K_{c,s}\}K_s, \{c\}K_{c,s}$

1. C requests for a ticket from the Kerberos server (Ker) to use for communicating with a ticket-granting server (TGS). A ticket is a credential used to securely pass the identification of the ticket holder to another service.
2. Then Ker issues a ticket to C that contains C's identity, C's IP address (addrc), the name of the ticket granting server (tgs), a timestamp (t), lifetime of the ticket (l) and a random session key, $K_{c,tgs}$ to use in the communication with the TGS. This is encrypted, together with $K_{c,tgs}$, using the private key, K_c, known only to the Kerberos server and C.
3. C uses this in communicating with the TGS by sending the ticket and the name of the server (s) hosting the service they wish to access to TGS which checks for the validity of the ticket and generates a new random session key, $K_{c,s}$, to be used between the C and S.
4. The TGS then creates a ticket containing the client name, c, the name of the server, s, a timestamp, t2, lifetime of the new ticket (l2) and the session key it has generated and sends back to the client encrypted with the initial session key that was generated by the kerberos server.
5. C sends the ticket and her identity encrypted with the key $K_{c,s}$ also found in the ticket to the server for authentication. The server first decrypts the ticket and checks its validity by comparing the lifetime to the current time and if valid, uses the key $K_{c,s}$ to decrypt the other part of the message which contains identification information for C. S compares this identity to that in the ticket if its the same then C is authenticated to S.

In Condor, Kerberos authentication is enabled by including it in the list of authentication methods in the configuration settings and then specifying a file that maps between the names in the Kerberos database and the user ids used in the domain in which condor operates. The location of the private keys must also be specified.

Other Authentication Methods

Condor also supports other authentication methods including: Secure Socket Layer (SSL), uses X.509 certificates to authenticate users and resources; password, uses shared secrets; File System, uses the ownership flag on a file to determine the identity of a client; Windows, uses a proprietory windows security interface SSPI using the password as the keys; ClaimToBe, accepts anything that a user or service claims to be; and anonymous which implies that authentication be entirely skipped [29].

2.6.2 Credential Storage

The authentication methods described above form the basis of both providing a confidential communication mechanisms as well as enforcing authorisation policies. The two authentication methods, GSI and Kerberos, referred to as the "strong authentication" methods [29], are preferred when a more secure environment is required. But just how secure are these methods? To answer this question requires a quick look at how the credentials used are stored and secured.

As described above both GSI and Kerberos, expects the private key to be stored in a directory on the disk. The private key will normally be encrypted using a password as an encryption key in GSI while Kerberos uses the encrypted password itself as the private key. Therefore the strength of these two methods depends on the strength of the password used.

During authentication, the daemons first specify the location of the credentials in environmental variables that are read and used by the authentication method. Authentication also produces session keys that can be used to encrypt user data, job information as well as resource information in the case of signed classads. The session keys are generated by the party acting as the daemon in the communication using a cryptographic library that is part of Condor.

The usage of the authentication methods as described above brings up a number of security threats which are described in the next chapter.

Chapter 3

Threat Model

Computer systems are never completely secure and may have security lapses of varying severity. These lapses are not always evident at first sight and may be due to the system design, bugs in the source code or perhaps its interaction with other systems. Identifying these lapses require carrying out investigations to develop an understanding of both the system and the environment in which it operates.

The Condor system has been in operation for a number of years now and has not suffered any known attack nor has anyone used it to launch an attack [2]. However, this does not mean that it is completely secure, but might imply a lack of sufficient incentives for potential attackers. This situation might change as its use increases.

OxGrid continues to grow its user base as more departments and colleges continue to join the grid. And with its growth comes an increase in the number of computers connected to it and amount of data and computations being carried on it. A consequence of this growth is an increase in the possible attack points and incentives for potential attackers who might be seeking to either disrupt its operation, modify data or read confidential data. Therefore an investigation that would point out lapses in its security would be invaluable to its success.

3.1 Arbitrary Code Run on the Grid May be Malicious

One of the biggest security threats faced by OxGrid, and grid computing systems in general, is brought about by its ability to allow users to run arbitrary code. OxGrid allows users submit jobs in form of executable code and input data. This presents

an opportunity for a malicious user to submit malicious code with an intention of disrupting the operations on the grid, modifying the data, reading confidential data or changing the running job submitted by another user, as Milner et al demonstrates [21]. Though it is possible to restrict what kind of code can be run on the grid, this is not a desirable option because it takes away the flexibility and thus become unattractive to users.

3.2 Malicious Host May Steal Sensitive Job Data

The main feature of grid computing is the ability to run jobs on any resource that is available and meets the requirements of the job. Furthermore, the users may not be aware about the resources that run their jobs. Implying that users cannot tell the state of the resources and therefore have no guarantee in their correct behaviour. Humphrey and Thompson [14] notes that the ability to locate and determine the status of the resources on the Grid is crucial to its functionality. The problem is that if a resource on the Grid is malicious, it may steal the data, modify it or expose it to un-wanted parties and thus compromising its confidentiality, integrity and availability [9].

3.3 Credential Compromise Due to Use of Delegation Services

One characteristic of the jobs that are run on the Grid is that they are long running. Since the users cannot always be there to authenticate each service that needs to access the data for the jobs, users normally delegate this duty to other services which act on their behalf. This means that users have to also give their credentials and privileges to these delegation services. A potential problem with this scenario is that the delegation service maybe malicious or may be attacked and thus compromise the credentials of the users. One of the solution to this problem is giving the delegation services the minimum required privileges. However, it is difficult to know exactly what set of rights the delegation services will require as well how many delegation levels [14]. As a result, users will normally trust the delegation services with all their credentials and privileges to act on their behalf.

3.4 Weak Host Identification by Central Manager

The idea of the campus grid is to maximize the use of resources within the university without drastically lowering the quality of service the equipment renders to the owners [18] as well as to allow resource owner to decide when to contribute resources to the grid. As the resource join the grid, they need to be identifiable to the central manager. Currently, they are identified using their IP address while users are identified using their name and other credentials assigned to them. Though it might be safe to use IP addresses to identify the resources, it is not very secure as malicious individuals may steal the IP addresses and use them to impersonate the real resource [2]. It is important that a better method of identifying the resources be employed [2] because all other policies such as authorisation and access control depend it.

Condor relies on what it refers to as "strong authentication" methods, i.e. Kerberos and GSI, to identify entities on the grid. The problem with this approach comes from the way the Condor daemons uses them. As described in the previous chapter, the daemons specify the location of the credentials and then make calls to the underlying authentication method. This posses the risk that other programs running on the resource can to authenticate to the central manager in the same way the daemons do. This is allowed because the daemons are not securely tied to the platform on which they run as a result the central manager cannot securely identify the hosts using information provided by the daemons.

3.5 Access to Credentials

One of the main challenges in securing computing systems is the storage of security credentials. The overall security of a given system depends on the security of the credentials used and therefore the credentials have to be protected with the same effort. The campus grid is configured such that entities, users, hosts or programs, are assigned certificates and private keys encrypted with a password. When an entity requires to authenticate to another entity, they have to provide a password which is used to decrypt the private key before it can be used in authentication. Two problems exist with this approach, the first is that the entity may be careless in handling the password and expose it to an adversary and the second is that an adversary can guess the password by performing a dictionary attack and therefore get hold of the private key [4]. These problems are caused by the use of passwords only to protect the

credentials and the lack of a mechanism to verify that the entity using the credentials are the true owners. This would allow malicious code to pretend to be the entity that owns the credentials and perform actions on behalf of the real owner.

Condor can also be configured such that each of the daemons is assigned its own credentials [28], a process that has not been documented or implemented anywhere, so that during authentication, the daemon's name has to match that in the certificate. This offers a more secure operation but is not sufficient because if malicious code can stop the actual daemon, then it can get authenticated by pretending to be the real daemon. The problem here is that the subject name in the certificate is not strongly tied to the name of the daemon. This means that having the right credentials does not imply that the program having the credentials is a true Condor daemon. This may be solved by placing the hash in one of the fields for the certificate, such as the Subject Unique Identifier. However, this might imply computing the hash of the daemon everytime it needs to be authenticated and this may not be scalable.

What is needed is a mechanism that only allows the right owners of the credentials to make use of them and prevents any other malicious code from using credentials. Without such a mechanism, results of the jobs risk losing its integrity. For example when a user submits a job from a machine whose credentials have been exposed to some malicious code, the malicious code will use the credentials and act on behalf of the user as if it were a Condor daemon. It can then change the inputs to the job or wait for results and modify them before passing them to the user. Similarly, if the credentials on an execute machine has been exposed to malicious code, it can pretend to be a Condor daemon and authenticate itself to the submit machine, and then modify the results.

3.6 Match-making Risks Failing

The central manager is responsible for performing match-making within the Condor pools. But what guarantee does both parties being matched have that the match was correct? What if it was a random assignment? How can the central manager provide an assurance to the matched hosts that the matching was correct? The answers to these questions would seem to natural be that the hosts trust the central manager to do the right thing. However the central manager is a machine like any other and may be compromised in the same way as the other hosts. If any of the daemons that

enable match-making, negotiator to be precise, are replaced by some malicious code, the entire pool, and consequently the grid, may become un-usable as the malicious code may raise the priorities of the machines, perform wrong matching or skip the process entirely. Therefore without a secure assurance mechanism of the state of the central manager, it cannot be completely trusted to perform match-making correctly or to correctly enforce any policies.

3.7 Wrong Classads may lead to DDOS attacks

Distributed Denial of Service (DDOS) attacks on the campus grid would fall into two groups. The first one is an attack against an individual host and the other is an attack aimed at the entire pool [2]. The former may occur if malicious code present on a machine the user uses to submit their jobs creates classads with job specifications that have the lowest probability of being satisfied. This proceeds as follows:

- User submits job description information
- Malicious code running on the submit machine receive the description and generate a classad that has the least probability of being matched. For example, the user may specify the need for Java 1.6 and malicious code would change that to 1.7 or something higher and thereby lowering the chances of a match being found
- S would then send the classad knowing for sure that it will never be satisfied

In the later case malicious code would flood the pool with job requests with elevated priority. The central manager would keep matching resources on the grid to the requests coming from the malicious code and render the pool unusable to other hosts. Though this problem is not restricted to malicious code running as Condor, for example any who can submit can carry out a DOS attack [2], it differs in that it is caused by the software which the user trusts to carry out their instructions but are similar in that they are both an implication of one entity trusting another without verification.

3.8 Weak Policy Enforcement on Software Requirements

Sometimes a user might require that the host that executes their job be running some software package, for example, antivirus or operating system patch. Currently,

Condor supports this using configuration settings where the job description is defined with a setting that indicate the software that should be running. These settings, discussed in section 2.2.2, are performed in configuration files and the user has to trust that the execute host will not lie about the software they state to be running. However, an impostor may change these settings to falsely state what software is running. As a result the central manager matches the requirements without verifying the statement from the execute host and thus does not observe the policies of the user. This weakness is caused by the inabality for the Central Manager to securely verify the existence of the required software.

3.9 Threat Model Summary

The threat model has identified several serious threats, among others. Cooper and Martin [9] proposed a solution for the threats discussed in sections 3.1 and 3.2, which uses trusted virtualisation to isolate the machines from the jobs. Due to a limitation on time, this thesis will concentrate on the threats related to the use of Condor in a production environment such as OxGrid. This will include the threats covered in sections 3.4, 3.5, 3.6, 3.7 and 3.8, which are mainly associated with the lack of a secure identification mechanism for hosts on the Grid as well as the lack of a procedure for verifying that the host run the correct Condor software and not some malicious code that pretends to be Condor.

Chapter 4

Trusted Computing Infrastructure

This thesis makes use of the term "trusted infrastructure", but what exactly is meant by it and how does it relate to the problems described in the previous chapter? This chapter presents an overview of the trusted computing technology, discussing some of the most important concepts and how relevant they are in such a problem domain. It does not, however, cover all concepts within trusted computing because that would take up an entire thesis. A complete coverage of the concepts can be found in texts such as [22], [5] and specifications from the Trusted computing Group [42].

4.1 What Trusted Computing is

The term trust may be understood to mean different things depending on the context in which it is used. In order to understand what is meant by a trusted infrastructure, one has to understand the meaning of trust - what or who is being trusted and with what and what the consequences, of breaching this trust, are.

4.1.1 The Notion of Trust

The notion of trust is very difficult to define precisely and is often used without giving a thorough thought on what it means within a given context. When applied to objects, it implies one's belief in an object to act according to their expectations. When applied to humans, it involves elements of responsibility, accountability and discretion. So that when one says they trust someone else, they expect that person to act responsibly and to be held accountable if anything goes wrong. But how can trust be interpreted when applied to computer systems?

In this context, trust implies believing the actions or results of a process to be correct

and accurate. Because there is a commonplace use of the word trust in this concept, the words "lie" and "believe" be used in the same spirit, aware that to imply such characteristics for a computer is potentially problematic. Therefore a system is trustworthy when it operates in an expected way and cannot lie about the results of its operation or its state within a given time period. It is also important to point out that the term trust is relative, in this context, so that a system can be deemed to be trustworthy by one system but at the same time be deemed un-trustworthy by another system.

4.1.2 Trusted computing technology

Trusted computing technology provides a mechanism for verifying the status of a given system at a point in time thereby allowing other systems to evaluate the extent to which they can believe anything from it and use that information to decide whether to interact with it or not.

4.1.3 Trusted Platform (TP)

A platform is a single entity compromising of storage and computation capacity, software, a superuser and a Trusted Platform Module operating as specified by the TCG, which provides services to a user in such a way that it cannot lie about the results of its operation and also allows other entities to evaluate how trustworthy it is.

4.1.4 Trusted Computing Infrastructure (TCI)

A trusted computing infrastructure (TCI) is a collection of inter-connected platforms each of which can be evaluated for its trustworthiness at a given time that ranges from completely untrusted to fully-trusted.

4.2 Trusted Computing Standards

The many security threats in computer systems have led major hardware and software vendors to invest substantially in trusted platform technologies. These efforts are supported by the Trusted Computing Group (TCG) [42], an international standards body compromising of 140 companies, including HP, IBM, Microsoft, Intel and other I.T business leaders, which continues to develop specifications for the design and use of trusted computing technology. These specifications aim at producing an environment

in which the sensitive data, such as cryptographic keys, is protected from malicious software and the entities within the environment can report, accurately, what their state is. The specifications ensure interoperability and platform independence and include standards for a hardware chip called trusted platform module (TPM), Application Programming Interfaces (API) for using the technology, protocols to ensure a trusted computing environment [44], trusted network connection, virtualisation using trusted computing, storage and many others.

4.3 Trusted Computing Concepts

In order to effectively use trusted computing technology, we ought to understand a number of concepts and how each one of them helps to enhance security. This section describes the concepts that were thought to be essential for this thesis.

4.3.1 The Trusted Platform Module (TPM)

The idea behind the trusted platform module comes from a realisation that software alone cannot defend itself but by combining it with hardware, computer security can be enhanced in both its strength and performance. The Trusted Platform Module (TPM) is a hardware chip that has been specially designed to provide hardware based cryptographic functions to a computer system in a way that cannot be subverted by software because all the sensitive information remains internal to the TPM itself. The TPMs are manufactured by different manufacturers (Atmel [1], STMicroelectronics [41], National Semiconductors [38] and Infineon Technologies [45]) who are free to include extra features in their chips but are required to meet the minimum features specified by the TCG.

The most recent specification includes functionality in the TPM that allows it, by using the Trusted Computing Group Software Stack (TSS) API, to securely generate and store RSA key pairs with the choice of making them non-migratable, in which case they can only ever be used on the specific platform that generated them, provision for RSA encryption and decryption, RSA signing and verification, sealing of data to the state of a platform, and random number generation. It was designed to meet the following requirements:

1. to be inexpensive so that it does not significantly impact the prices of the devices in which they are used and thus ensure a wider adoption

2. to securely store sensitive data in a way that cannot be subverted by external software running on a platform

3. measurement of the state of a platform and accurately reporting it to any interested party so that they can evaluate how trustworthy the platform is

4. to provide cryptographic functions such as hashing using SHA-1 algorithm, random number generation (for nonces and prime numbers used in RSA cryptographic), generation and storage of RSA cryptographic keys and the ability to "wrap", discussed in section 4.4.3, existing keys with its own keys

4.3.2 Platform Configuration Registers (PCR)

The TPM is compromised of many internal components including Input/Output, Random Number Generator, non-volatile secure storage and an area of memory called platform configuration registers (PCRs). PCRs enables the TPM to store cryptographic hashes of data computed using the SHA-1 algorithm. The TCG specifications for the TPM does not specify the exact memory capacity of each PCR or how many PCRs there should be in a TPM but most of them are 20-bytes long (large enough to contain SHA-1 data) and typical TPMs come with atleast 16 PCRs, out of which a certain number is reserved for specific purposes [5]. The main use of the platform configuration registers, infact the name says it all, is in the measurement of the state of a platform. This is achieved by a process called extend, described in section 4.4.3. The PCRs are also used in reporting the state of the platform, they are read and the value digitally signed by the TPM, in the "Quote" operation described in section 4.4.3.

4.3.3 Stored Measurement Log (SML)

Stored Measurement Log, most often called event log or simply log, maintains an ordered database of integrity changing events [22]. This means that every operation performed on any PCR register is logged to indicate the order and result of each operation. The log is very useful in integrity measurement in that the same operations specified in the log can be performed, in the specified order, and the results compared. So that any disparity would indicate a loss of integrity.

4.3.4 Roots of Trust

Challener et al [5] notes that it is hard for software to prove that it can be trusted. This is because there is nothing that is trusted to answer the question. The notion of root of trust in trusted computing platforms provides a means of believing the answer if it can be traced back a root of trust because it is assumed that the root of trust cannot be subverted.

Root of Trust for Measurement (RTM)

In order to determine the trustworthiness of a platform, its state must be measured first. The root of trust for measurement is a point, assumed to be uncompromisable, from which measurements of a platform can be started. It provides the functionality to measure the state of a platform and record it in the PCR through the extend operation, discussed in section 4.4.3.

Root of Trust for Storage (RTS)

The root of trust for storage is a TPM specific capability that provides a point from which any storage on a given platform can be trusted to be secure. The design of the TPM allows one to use this root of to store data outside the TPM and be sure that it will be secure because it encrypts with a private key that never leaves the TPM. It uses the measurements in the PCRs and RSA encryption to securely store data and cryptographic keys.

Root of Trust for Reporting (RTR)

One of the essential functions of the TPM is that of accurately reporting the state of the platform. This is enabled by the Root of trust for reporting (RTR). It offers a point from which integrity measures reported from a platform can be trusted. It uses integrity measurements in the PCRs and RSA cryptography to securely and accurately report the state of a platform.

4.3.5 Transitive Trust Property

The transitive trust property allows one component to determine how trustworthy another component is and transfer its trust to that component. For example component, say A, measures another component, say B, and passes control to it once it has determined its measurement to be correct. This property allows the trust within the RTM to be passed to the next component after determining whether it is in a correct

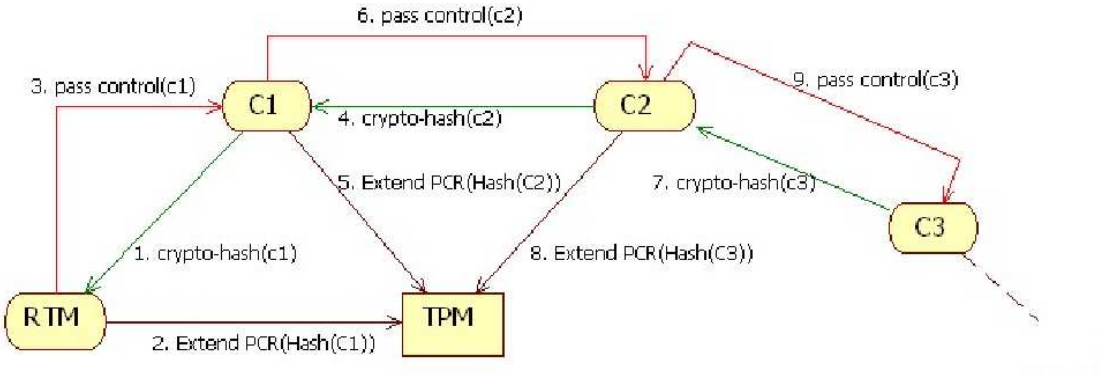


Figure 4.1: The chain of trust

state. Each component subsequently measures the next one before passing it control, thereby creating a chain of trust that can be traced back to the RTM, as illustrated in figure 4.1.

4.3.6 Protected Storage

The current specification of the PC (i.e. without trusted computing capabilities) has a major weakness of storing secrets such as passwords and cryptographic keys. These secrets are normally stored in a way that exposes them to phishing and dictionary attacks because they are stored on the same machine that holds the information being protected (and uses OS access control to restrict their exposure). But how can the secrets be stored more securely? Trusted computing technology's answer is the TPM's Protected storage functionality.

Protected storage is a mechanism for ensuring that sensitive data is protected with keys generated by the TPM so that the data can only be exposed if the TPM determines that the platform is in the same configuration it was in at the time the data was protected. It ensures that the confidentiality of data is maintained using asymmetric encryption [22]. This is achieved through a process called sealing and its reverse operation unsealing, discussed in sections 4.4.3 and 4.4.3 respectively.

4.3.7 Cryptographic Keys

The TPM makes use of a number of keys to carry out its functions. These keys are usually between 1000 and up to 2048-bits RSA keys. The following sections cover the main keys and their usage.

Endorsement Key (EK)

The Endorsement Key (EK) is a 2048-bit RSA key pair uniquely generated by the manufacturer for each TPM. Once generated it gets stored in a TPM-shielded location [22] to prevent it from being modified. The EK is the only key that ships with a TPM and is issued with a certificate which together can be used to sign data to prove that it comes from a valid TPM. However, it is strongly advisable not to use it in this way because it can be used to uniquely identify the host (which has privacy concerns). Instead the Attestation Identity Keys, described below, should be used.

Storage Root Key (SRK)

The Storage Root Key is a 2048-bit RSA key generated at the time of activating a TPM. It is a root of all the other keys, apart from the EK, used in various TPM operations and is also described as being non-migratable [5], which means that it cannot be used on another TPM.

Storage Keys

The TCG specifications defines storage keys as "asymmetric general purpose keys used to encrypt data or other keys" [43]. They are 2048-bit RSA cryptographic keys that can either be migratable (can be moved to another TPM) or non-migratable (only usable on the TPM on which they were created.)

Atestation Identity Key (AIK)

The use of the Endorsement key for signing data coming from a TPM raises privacy concerns because they are unique to a platform. The Attestation Identity Keys (AIK) are used, instead, to digitally sign data from the TPM. Though they are linked to the EK, they still can be used without exposing the identity of the platform because the TPM allows multiple AIK's to be generated for each operation, if needed.

4.3.8 Attestation

Attestation provides a mechanism for determining the software stack running on a given platform and use it to evaluate how trustworthy a platform is. This can also be verified remotely, in which case it is referred to as remote attestation, defined by Cooper and Martin [9] as "a powerful security primitive that allows a party to obtain assurance in the correct operation of a system that is beyond their physical

control" while Bottoni et al [3] describes it as "a process that allows a local platform to cryptographically authenticate the hardware and software stack running on a remote platform". Both of these descriptions point to the fact that a remote host can determine whether another platform is operating correctly. To achieve this, the system wishing to attest has to have its integrity measured and compared against a set standard. For example, the TCG specification [43] highlight the measurement of binary code while [7] proposed a protocol that users properties of a platform rather the cryptographic hash value of the binary code. The general idea behind these measurements, called integrity metrics, is that they can be compared with some other metrics measured at a different point while verifying that they were generated by a valid TPM.

Let there be a platform A whose state is to be verified and a remote host B challenging A to verify its state. Let A be running a software stack S on hardware H and let PrkA be A's private key (private component of the AIK pair). Also assume that B already has the initial settings to compare any metrics that will be sent (referred to as the root of trust for authentication by [3]). Then attestation could be summarised as follows:
$1 : B \rightarrow A : n_b$
$2 : A \rightarrow B : \{S, H, n_b\} PrkA, cert$
In 1, B challengers A by sending a non-predictable nonce n_b. When A receives, it gets a quote of the hardware and software running, adds the nonce received and encrypts with the private key (PrkA) and sends this together with a certificate (cert), in 2. B can then check the authenticity of the message by checking the validity of the certificate to ensure that the message is coming from a valid TPM. B decrypts the message using public component of the A's AIK, compares the nonce with what was sent earlier and then compares the metrics S and H with what it already has. If the metrics match then A would have successfully attested its state to B.

4.4 TCG Software Stack (TSS)

The TCG Software Stack (TSS) is a set of modules specified by the TCG to provide an interface between a trusted computing application and the TPM. These specifications allow a platform independent access to the TPM. Some notable implementations of these specifications include, the trousers project [35], a C++ implementation and the

IAIK jTSS stack [15] which provides a Java implementation. Both of these implementations allow both the management of the TPM and the programming of applications to make use of trusted computing features. The components of the TSS are discussed below:

4.4.1 TCG Device Driver Library(TDDL)

The use of the TPM is enabled by a device driver called TDD (TPM Device Driver). This driver is specific to an operating system and a particular TPM. It is created by the manufacturer of the TPM who also provides an interface, the TCG Device Driver Library (TDDL), for accessing the device driver.

4.4.2 TCG Software Stack Core Services (TCS)

This is an interface that provides a common set of services for managing keys and other credentials, storing, managing and reporting of operations and resulting values on the PCRs and the generation of byte streams from the parameters passed to it for use by the TPM [43].

4.4.3 TCG Service Provider (TSP)

This is the module that provides an object oriented interface to any application that requires trusted computing features. It provides a handles for efficiently using and managing the resources within the application. It also provides cryptographic functionality such as hashing and byte stream which is not provided by the TPM [22].

Object Types

The TSP includes various objects that allow it to perform the many TPM functions. These objects are the main TSS components available to a programmer wishing to write a trusted computing enabled application. They include; TSS_HCONTEXT, which is used to maintain a handle to a particular TPM chip and for creating other TSP objects; TSS_HTPM, an object representing a TPM to which the context is connected; TSS_HPOLICY object is used for passing authorisation data to various operations; TSS_HKEY object offers functions that enable various operations on cryptographic keys; TSS_HHASH object is used for holding the hash value resulting from the SHA-1 hashing performed by the TSS; TSS_HENCDATA object, holds

encrypted data and the TSS_HPCR object represents the platform configuration registers and their digest values. Others include TSS_HNVSTORE, TSS_HMIGDATA, TSS_HDELFAMILY and TSS_HDAA, see [5] for full details.

Cryptographic Hashing

The TSS provides cryptographic hash functionality using the SHA-1 algorithm without the involvement of the TPM chip. Cryptographic hashing produces a valued that is non-associative, implying that the order of operations to obtain a given hash value matters. To hash data, say rgbData, pass it to the update hash value function as follows [43].

```
TSS_RESULT Tspi_Hash_UpdateHashValue
(
   TSS_HHASH hHash, UINT32 ulDataLength, BYTE* rgbData
);
```

hHash is the object that will hold the resulting hash value and ulDataLength is the length of rgdData.

Extending the PCR

The extend operation is the only way of changing the information stored in the PCRs. It is a non-associative process that computes the cryptographic hash of data passed to it using the SHA-1 algorithm and stores the result in a given PCR. This operation can be summarised as:
$extend(n, "ABC") \rightarrow SHA-1(getValueAtReg(n)||"ABC")$ where getValueAtReg simply gets the value stored in register n. The result is a 20-byte cryptographic hash stored in register n and a SML indicating the operation on register n and the result. This operation can only be achieved using the extend operation provided through the TSP as follows [43]:

```
TSS_RESULT Tspi_TPM_PcrExtend
(
  TSS_HTPM hTPM, UINT32 ulPcrIndex, UINT32 ulPcrDataLen, BYTE* pbPcrData,
  TSS_PCR_EVENT* pPcrEvent, UINT32* pulPcrValueLen, BYTE** prgbPcrValue
);
```

where, hTPM is the object representing the TPM onto which the operation will be performed, ulPcrIndex = PCR index to extend, ulPcrDataLen and pbPcrData is the data to extend and its length (respectively), pulPcrValue and pulPcrValueLen is the data of the resulting hash value and its length (respectively) and the pPcrEvent saves as an SML.

It is important to point here that the TPM does not allow data to be written directly to it but provides the extend operation as the only way of changing the values in the PCRs. This implies that external software cannot give the force TPM to write a value of their choice.

Sealing

Sealing is a TPM functionality that allows data to be protected with cryptographic keys and tied to the state of a platform at a particular time. The data being protected is encrypted together with the platform state using a 2048 bit RSA cryptographic key generated by the TPM (specifically for the platform it is running on) and is only decrytable if the platform and its state matches that at encryption. For example if a platform, running say program A and B, protects data, it produces what is called a sealed package that has information indicating that programs A and B is running on the platform. Now when another program, say C, runs on the platform and tries to access the data in the sealed package, the TPM detects that another program C, is running and refuses to expose the data. Sealing is achieved using the TSP's seal function as follows [43]:

```
TSS_RESULT Tspi_Data_Seal
(
   TSS_HENCDATA hEncData, TSS_HKEY hEncKey,
   UINT32 ulDataLength, BYTE* rgbDataToSeal, TSS_HPCRS hPcrComposite
);
```

Where hEncData is an object that will hold the encrypted data blob, hEncKey is the TPM generated key used to encrypted the data, rgbDataToSeal is the data to be sealed and ulDataLength is its length and the hPcrComposite is an object that holds the indices and the digest value of the PCR that should be checked at unseal. Optionally, the Tspi_Data_SealX function can be used which only differs in that the data to be sealed should be encrypted before passing it to the TPM.

UnSealing

Unsealing is the reverse process of sealing. It takes the sealed package and presents it to the TPM which checks if the platform state is the same specified in the encrypted data blob before decryting it. The unseal operation is the only operation that can review the data in the encrypted data blob and will only work if the it occurs on the same TPM that performed the sealing. This is achieved as follows [43]:

```
TSS_RESULT Tspi_Data_Unseal
(
   TSS_HENCDATA hEncData, TSS_HKEY hKey,
   UINT32* pulUnsealedDataLength, BYTE** prgbUnsealedData
);
```

Where hEncData is an object that will holds the encrypted data blob, hKey that should be used to decrypted the data, prgbUnsealedData is pointer that will hold the un-encrypted data if the operation is successful and pulUnsealedDataLength will hold its length.

Quoting Platform State

The Quote operation is the main operation that enables attestation, described above. It is used to take a note of the platform state in a way that can only be performed by a TPM and cannot be forged. The operation reads the contents of the PCRs, adds a nonce (which can also be passed in by a remote host) and signs it with an AIK, described above. This information can be sent to a remote host who can then verify the state of the platform by checking whether the key used, AIK, comes from a valid TPM. The quote operation can be performed on the entire set of PCRs of a subset depending on which of the registers are of interest. It is performed by invoking the following TSS operation [43].

```
TSS_RESULT Tspi_TPM_Quote
(
   TSS_HTPM  hTPM, TSS_HKEY hIdentKey,
   TSS_HPCRS  hPcrComposite, TSS_VALIDATION* pValidationData
);
```

Where hTPM is the object handle to a particular TPM which needs to be quoted, hIdentKey is the key that will be used to sign the results (this is normally the AIK), hPcrComposite holds the list of PCR indices that need to be quoted and pValidationData holds both the data passed in (nonce from remote host) and the data that is signed and passed to a remote host for verification.

Wrapping

Wrapping is the process of encrypting externally generated keys with a key generated by a TPM [22]. Optionally, the key can also be linked to the PCR so that it only becomes decryptable if the platform configuration can be verified to be the same as that at encryption. Wrapping is performed using the TSS function [43]:

```
TSS_RESULT Tspi_Key_WrapKey
(
    TSS_HKEY  hKey,
    TSS_HKEY  hWrappingKey, TSS_HPCRS hPcrComposite
);
```

Where hKey is the key that will be wrapped with hWrappingKey as the wrapping key and hPcrComposite indicates the PCR that should be checked at the time of unwrapping the key (unwrapping is the reverse process that exposes the key that was wrapped).

Key Management

The TSS provides several mechanisms for managing cryptographic keys used in the operations. This includes registering keys in system or user persistent storage, creating new keys, loading keys into the TPM (to be used for some operation) and many more. For a comprehensive coverage of the key management, the reader is referred to the TCG specification [43].

4.5 Trusted Boot and Operating System

Achieving the full potential of trusted computing requires the support of operating systems. [12] notes that the current operating systems do not provide a security model that is sufficient for trusted computing because they lack a mechanism for providing assurance of the platform they run on. "High assurance is essential as a trusted OS must instill confidence in remote parties that it can be relied upon to

execute their code in a well-specified fashion" [12]. The TCG standards provide a means by which a trusted boot process can be achieved. In this process when the system is powered on, control is immediately switched to an immutable base which measures the net component in the boot process before passing control to it. This goes on until the boot loader is loaded. This ability to measure a component before passing control to it ensures that any changes to the boot process can be detected and reported to an external challenger and can also be used to prevent an attacker from taking control of the machine. However, these specifications end at the boot loader. Fortunately significant work such as [37] has demonstrated that this process can be extended to the operating system and up into the application layer. Sailer et al [37] describes their work on the Linux kernel which would provide an operating system with a non-intrusive software stack that is verifiable to a remote machine and trustable attestation for the software stack using existing hardware and operating systems to ensure integrity without which s system cannot be relied on to act correctly.

Though this is far from being achieved, significant effort such as the work on Terra [11], an architecture built on trusted virtual machine, Microsoft's efforts on the Next Generation Secure Computing Base (NGSCB), which provides a way of separating trusted and untrusted parts each with its own operating system and the research by [13], in which they demonstrate a way of proving that a valid operating system was booted show that this is a worthwhile effort and promises more benefits towards the maximal use of trusted computing technology.

4.6 Relevance to the problem

As the number of users connecting and using resources on the OxGrid increases, the volume of data computed on the grid increases and thus more concerns on its security. This is because increased usage comes with an increased number of varying security requirements. Paramount to all the security requirements is the requirement that making use of OxGrid should not compromise the integrity and confidentiality of the jobs and its data. This implies a need for a mechanism that would allow the hosts on the grid to be identifiable in a way that cannot be subverted as well as a means for the hosts to prove that they are running the genuine Condor software. Such requirements cannot be met by the use of software mechanisms alone such as the use username/password because they are liable to spoofing. The use of trusted computing enables systems to not only provide an authentication means that cannot

be broken, unless they break the hardware, but by so doing enhances the amount of trust one system has in another. Additionally, trusted computing offers a way for one system to check the state of another and thereby a means of checking the trustworthiness of the other system. These properties would help solve the threats outlined in the previous chapter and thus increase the reliability of the computations performed on OxGrid. It would also eliminate the current trust symmetry, which is based on un-verifiably trusting either the users of the system or the software running on the host machines, so that users can specify the conditions under which they can trust the machines that are to execute their jobs. and one that prevents malicious code from having access to the credentials.

Chapter 5

Solution Design

This chapter presents a design of the solution to the problems identified in the threat model. It follows a two-staged approach where the first phase of the design can be implemented as is or can be extended to the second phase if a more secure environment is needed. This provides flexibility to the implemeter so that they can decide whether to take full advantage of its features or not.

This design focuses on the case in which GSI or Kerberos authentication is being used on the host, but can be extended to any other authentication method with minor adjustments.

5.1 Design Phase 1: Integrity Measurements and Protection of Credentials

This phase provides a mechanism that measures the integrity of the Condor system at installation and startup and protects the credentials used in authentication. The following sections describe the design, giving its goals, steps and defining any assumptions made.

5.1.1 Goals

1. enable integrity measurement of a platform on which Condor runs
2. enforce the measurement of the components of the condor system before they start running
3. protect the authentication credentials from malicious code

5.1.2 Trusted Condor Installation

The first step in this design is to install Condor in a "trusted mode". "Trusted mode", as used in this context, implies that the platform on which Condor is installed can be evaluated for trustworthiness both locally and remotely. This will provide the parameters necessary to use in the evaluation of trustworthiness. This, however, raises the question of how the parameters can be measured and why the measurements would be trusted to be correct? To allow this design to work, the following have been assumed:

ASMP-1: Trusted Operating system

The idea of a trusted operating system, discussed in section 4.5, provides an assurance to the users that it operates as expected. It can be relied on to execute instructions passed to it and the results of its operations to be of high integrity. This ensures that a component measured by the OS or by a component that was measured and loaded by the OS can be trusted, using the transitive property discussed in section 4.3.5.

ASMP-2: Trusted Condor Source

This assumption implies that the Condor software that will be installed is not in a compromised state so that it works as specified by the producer. This is important because it avoids measurement of code that is already malicious and also helps concentrate the effort on improving security of the installed system rather than on authenticating the source of the software.

The sections below discusses the necessary steps for installing Condor in a trusted mode, also illustrated in figure 5.1.

Machine Preparation: Enabling a Trusted Platform

When a system, with a TPM chip, is shipped, the TPM is not enabled by default. It requires the owner to active it and establish the owner credentials and other authorisation data. A TPM will normally ship with three things: an endorsement key, the only key in the TPM; driver file; and a certificate signed by the TPM manufacturer that shows the authenticity of the endorsement key. To run a trusted platform requires the installation of the device driver, activation and taking ownership of the TPM [23] and installation of a TSS. The driver is installed according to the instructions for the operating system being used. For example, the linux kernel 2.6.23 ships with the necessary drivers for the TPM, so there is no need to install them. The TSS

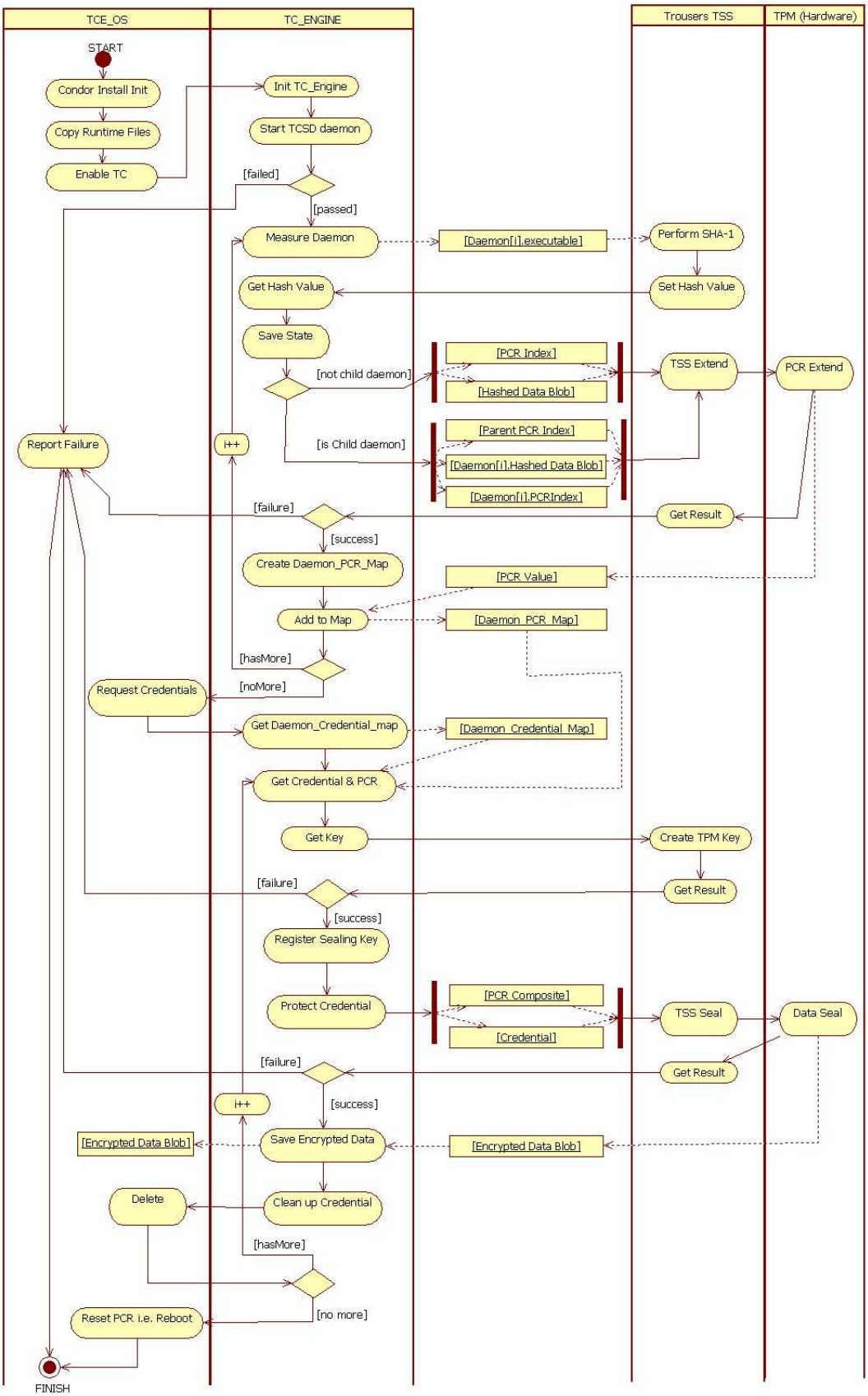


Figure 5.1: Trusted Condor installation

(choice of TSS implementation depends on the operating environment) should then be installed. This results in a platform with an SRK, 4.3.7, generated and all the features (integrity measurement, platform attestation and secure storage) necessary for the rest of the design steps enabled.

Condor Install

Installation of Condor involves copying the runtime files (daemon executable files and configuration files) into the installation directory and configuring the necessary settings [27]. The design adds TP, which is necessary for the other steps in the design. The various files will form the basis upon which integrity measurements can be taken. The following has been assumed:

ASMP-3: Person performing installation is trusted

This implies that the person installing Condor will not replace the genuine copy of the software with some malicious code before installation and will also not skip the necessary installation steps. This ensures that the files on the machine can be trusted to be correct.

However, this does not prevent malicious code already running on the system from replacing or corrupting the files.

Since the thesis aims at improving the security of the system once it has been installed and running, detecting changes to the system and evaluating its trustworthiness, the following has been assumed:

ASMP-4: The Platform has not been compromised at the time of installation.

This implies that the platform state can be trusted to be correct so that the installed files will not be corrupted before they are measured. Failure to observe this assumption would imply that the system that has just been installed cannot be trusted. This also helps concentrate efforts on defending a running Condor system rather than defending the platform, which can be done by applying other security defence mechanisms such as keeping the antivirus upto date or applying the necessary OS security patches.

Integrity Measurement of Condor Master Daemon

Integrity measurement provides the necessary parameters that can be used to evaluate the trustworthiness of a system. In this design, integrity measurements will be performed on components of the Condor software so that they can later be used to

both protect authentication credentials and to prove to a remote host that the platform's integrity has been maintained.

Because the Condor master daemon, see section 2.2.1, is the entry point into the condor software, it has to be measured first. The measurement of the master daemon is very important because it determines whether it will start the correct daemons and whether it will enforce the measurements of the other daemons as described in section 5.1.2. The master is measured by the trusted operating as assummed in ASMP-1, so that the measurement can be trusted to be correct. This however raises the question of why the master is being measured alone rather than Condor as a whole. The answer lies in the decision made as described in:

Design Decision 1 (DD1): Measuring Condor
Option 1: Measure all the files installed as Condor
This requires a definition to be agreed upon as to what constitutes the state of Condor. Because of Condor's flexibility, achieved through various configuration files, it presents a challenge of keeping track of the possible configuration settings. However, it might be possible to define it without the configuration files so that only the executable files are measure as described in option 2.
Option 2: Measure all the components seperately
Condor is comprosed of so many components, some of which only serve as commands and thus do not offer any service to the user other than pass instructions to the other daemons. These components can all be measured, however, there is no added advantage in measuring all of them except that it adds an overhead on the integrity measurement process.
Option 3: Measure only the necessary components
This option implies that only the components identified as essential for integrity are measured. It is the most desirable option and one that has been settled on in this design because it reduces the overhead on the measurement process and also ensures that we only concentrate on components whose measurement adds value to the security of the system. However, it has a disadvantage that one needs to decide which components add value and which do not. This process proceeds as follows:

1. The master daemon is read and hashed by passing the necessary parameters to the TSS hashing algorithm described in section 4.4.3. Other hashing algorithms, such as md5, can also be used but to avoid the complexity that somes with using

external libraries, the TSS hashing function is used [6].
2. The final step in the integrity measurement of the master involves storing the measured hash in the PCRs so that it can be used later. This is achieved by passing the hashed value to the extend operation discussed in section 4.4.3. The result is the hashed value stored in a given PCR and an SML 4.3.3 showing that the value has been extended in a particular PCR. The SML and the PCRs makes the integrity measurement complete [22]. To ensure uniqueness and create a link between the master daemon and the platform on which it is running, the extend operation is performed on the PCR index that contains integrity measurements taken upto the point the OS was loaded and assuming that:

ASMP-5: operating system integrity value is stored in a PCR
This implies that during the boot process of a trusted OS, integrity measurements were being recorded in the PCRs so that the trusted platform has a unique value (that indicates cryptographic identification of the platform) stored in some PCR.

This allows the chains of trust established through the boot process and upto the operating system to be extended to the master daemon and later to all other daemons and used to provide a basis for trusting a running daemon. It is also a means by which platform identification information can be passed on to the master daemon.

Measurement of Other Daemons

Once the master daemon has been measured, all the other daemons need to be measured. This operation requires that a relationship between the daemons to be defined because it defines what daemons are expected to be running to have a particular integrity measurement. This brings up the decision on how to relate the measurements:

Design Decision 2 (DD2): Measurement Relationships
Option 1: Measure each daemon independently
This option implies measuring each daemon independently without any kind of relationship to others. The problem with this option is firstly the lack of a relationship between the measurements suggest a lack of relationship between the daemons and secondly does not link to the master daemon which started it and therefore a given daemon cannot be cryptographically linked to the host on which it is running.
Option 2: Measure each child adding its parent's value
All daemons can be traced back to the master daemon that started it and the master

can be traced to operating system that invoked it (traced means being able to check how it started). It is therefore important to maintain this relationship for a number of reasons; firstly it would fit well when trust needs to be tranferred to another component, transitive trust described in section 4.3.5; secondly it provides a stronger protection mechanism because a compromise in one component can be detected by all related components; and finally it provides a cryptographic connection between a daemon and the platform on which it runs.

A guide to determining the relationship is to look at how they start. So that those that start independent of each would not be related. This is important because Condor can be configured to run any subset of the entire daemons available and relating those that exist together ensures that any subset of daemons can still run without the un-related daemons.

Measurement of each daemon is performed by first getting the value in the PCR that is being used by the parent and performing the extend operation together with the daemon hash value on a PCR assigned to the particular daemon. By so doing, each PCR used will inherit cryptographic identification information for the platform from its parent.

Obtaining Credentials from CA

In section 2.6.1, it was discussed that authentication credentials have to be placed in a directory pointed to by the settings GSI_DAEMON_CERT, GSI_DAEMON_KEY and GSI_DAEMON_TRUSTED_CA_DIR, when using GSI. This design does not make any difference to this process except that it needs to keep track of these locations for use in the next step, credential protection.

Protected Credential Storage

Trusted computing provide a secure storage of credentials by making it hard for malicious code to access or use them, as described in section 4.3.6. The preceeding sections described how to take integrity measurements of the platform and at this point, the TPM has a number of integrity measurements stored in the PCRs together with an accompaning SML. This steps makes use of this information to protect the credentials. It is achieved using the sealing process described in section 4.4.3 by first reading the credential file, generating a sealing key in the TPM and specifying the indices and digest values of the PCRs that should be checked at unseal. The following

has been assumed:

ASMP-6: Short time period between getting and protecting credentials
The time period between getting the credentials and protecting them may vary depending on so many factors, including the policies in place, the speed of the machines on which the process is taking place. This assumption implies that the time period between when the credentials are exposed un-protected to when they are protected is very short, so that no software can access the un-protected format. It ensures that the credentials being protected were not already compromised.

The result is a sealed data package which is saved in some user persistent storage. The next step is to clean up the un-encrypted credentials and clear the PCRs as described in the following section.

Credential Cleanup and PCR Reset

Now that the credentials have been sealed, the un-encrypted credentials need to be deleted from the system to prevent any software from using them and the state of the PCRs need to be reset so that the values stored there are cleared. Clearing the PCRs is essential because without doing so any software can unseal the sealed package by presenting it to the TPM. Clearing the PCRs forces the measurements to be re-taken before the package can be unsealed. ASMP-6 has also been assumed here, so that no malicious code will have access to the credentials that are still in an un-encrypted state. The easiest way of clearing the PCRs is a reboot of the machine which forces the machine to go through the boot process as described earlier to achieve a trusted platform. This results in a set of PCRs that do not have integrity measurements that can allow the unseal operation to succeed.

5.1.3 Trusted Condor Startup

Currently, to start running Condor, the master daemon is started which in turn starts each of the daemons listed in the DAEMONLIST as a child process, see section 2.3. This design changes this process to include integrity measurements before each daemon gets started. This is shown in figure 5.2 and is discussed in the sections that follow.

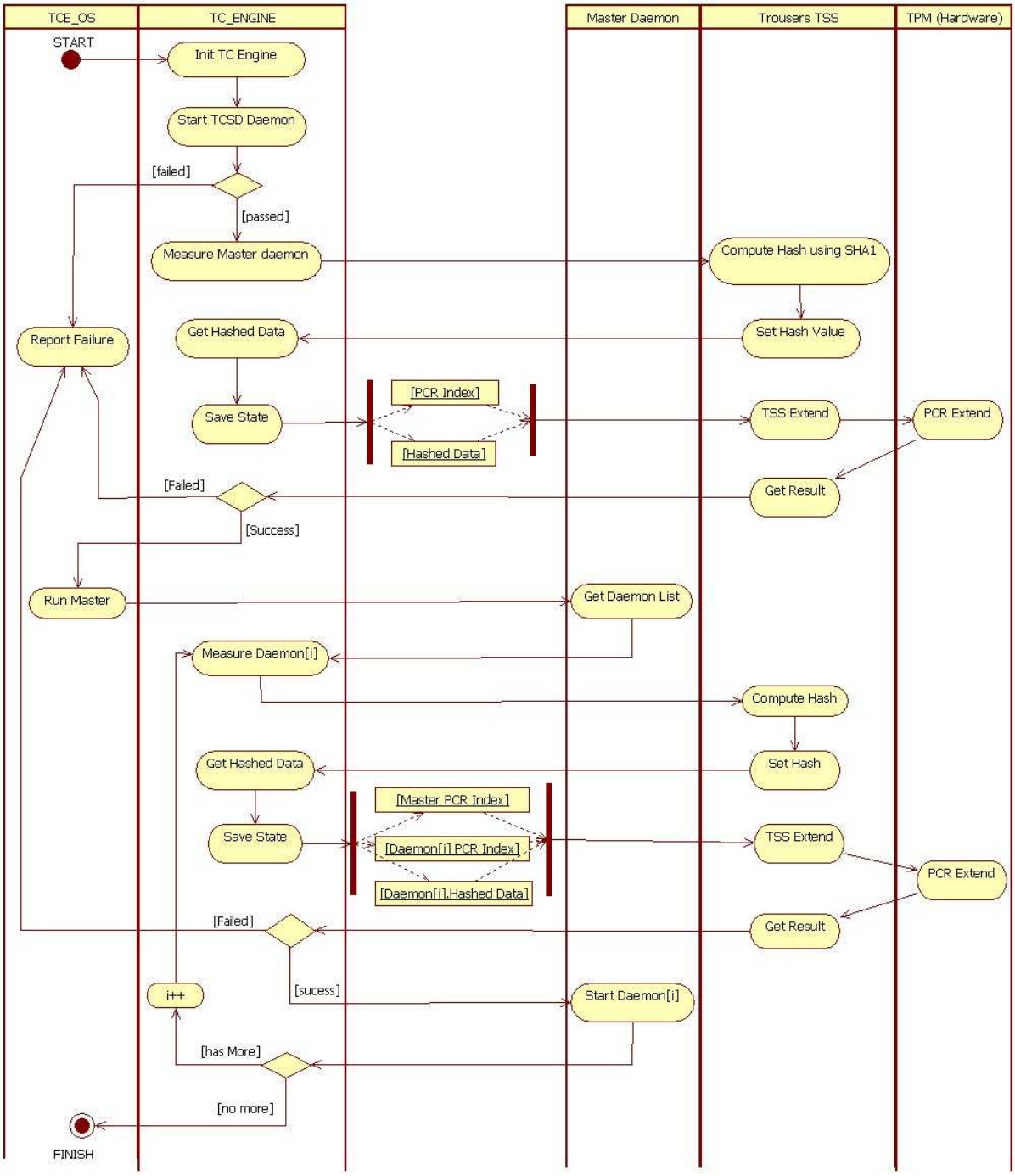


Figure 5.2: Trusted Condor startup

Trusted OS Measures and Loads Master Daemon

Before the Condor master is invoked, the operating system takes integrity measurement of the master daemon and stores it in the PCR. This is performed in exactly the same way as described in section 5.1.2 (get the hash of the master daemon and pass it to the extend operation) because the credentials were sealed with measurements performed following that particular process and since the extend operation, discussed in section 4.4.3, is not associative, the only way to obtain the same PCRs values used in the sealing process is to perform the exact operations that were performed at sealing. The difference from the process in section 5.1.2 is that the master gets control from the operating system once it has been measured.

When the OS measures the master daemon, and before passing control to it, two possible can happen: either the measurement equals that taken at installation time or different because the master daemon executable has changed. This presents the OS with a decision to either pass control to the daemon or not, depending on the outcome. To maintain Condor's flexibility (allowing different methods of authentication), this solution stores the measurements in the PCRs and allows the OS to pass control to the master daemon regardless of the outcome, leaving the duty of determining the correctness of the measurement either to a remote host or at the point when the platform requires to interact with others. However, if the measurement did not match, authentication will fail and the host would only be useful for communication that does not require authentication, ClaimToBe authentication [28].

Measurement of Each Daemon

This process follows the same steps as discussed in section 5.1.2 with an exception that only daemons specified in the DAEMONLIST are measured and started immediately. The rest of the daemons are measured at the point when they are started. Once the daemons have been measured, the PCRs will contain integrity measurements of all the running daemons which can operate in the same way they currently do, i.e. start waiting for commands.

5.1.4 Trusted Condor Authentication

At this point the daemons are running but the credentials are sealed. How does authentication take place if the credentials are sealed? This section describes how the daemons make use of the credentials and how they authenticate with each other. This

involves making the credentials available, using them in authentication and clearing them as illustrated in figure 5.3.

Prepare for Condor Authentication

Preparing for authentication involves exposing the credentials in a form that can be used by the authentication method in use, GSI or Kerberos in this case. When a daemon requires to authenticate itself to another daemon, it first gets the sealed credentials and presents them to the TPM to be unsealed. Unsealing, discussed in section 4.4.3, will create credentials in their original format, that is without protection from the TPM ready for use in authentication. This proceeds by the daemon instructing the TPM to load the key which was used for creating the sealed credentials and passing the sealed package to the unsealing process. The unseal process attempts to unseal and if the integrity measurements in the PCRs matches those specified during sealing, then the original credentials are obtained. If on the other hand, they differ then the process will fail and subsequently make authentication to fail.

Once the unsealed credentials are obtained, they need to be made accessible to the authentication method being used in Condor. To achieve this, the credentials are stored in a directory and the necessary Condor functions, such as setting environmental variables, performed to make the underlying authentication method use the credentials.

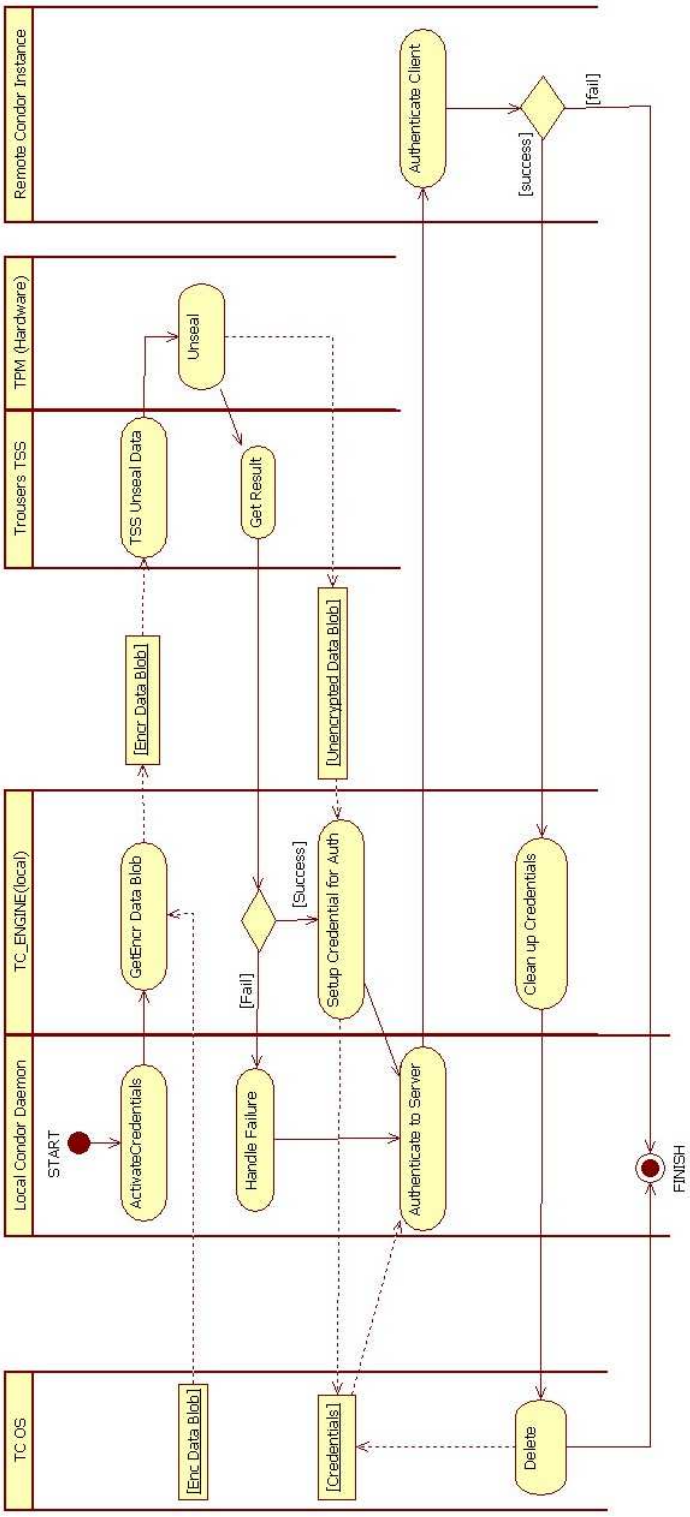


Figure 5.3: Trusted Condor authentication

Passage of Control to Authentication Method

Now that the credentials can be accessed by the underlying authentication method in use, control is passed back to Condor's authentication service to perform authentication. What the design would have achieved at this point is that the credentials have been protected up until the point when they are needed. This means that the period through which the credentials are exposed to the risk of being copied or used by malicious code is reduced.

Cleanup of Credentials After Authentication

Once the authentication process is complete, the credentials need to be cleaned up. This is done to prevent another daemon or some other un-authorised programs from accessing the credentials. This ensures that only the trusted daemons which unsealed the credentials uses them. Again this process relies on ASMP-6.

5.2 Design Phase 2: Attestable Platform

Attestation allows a platform to report to any host about its status in a manner that cannot be falsefied and thus provide a means of evaluating the trustworthiness of a platform. Phase 1 of the design discussed the installation of Condor in what was referred to as a trusted mode, which provided integrity measurements and protection of credentials. The implication was that a platform was considered to be in trustworthy state based on success of the unseal process, because unseal (and consequently authentication) would only work if that was the case. This does not however indicate a true picture because it is possible for malicious code to be running without affecting the PCRs required for unsealing. Therefore a remote host cannot be sure that a platform it is communicating with wasn't compromised.

Cooper and Martin [9] notes that attestation could be used to provide an assurance that a remote host can be trusted to operate correctly and thus guarantee that it will adequately protect user data. This design makes use of attestation to provide a mechanism by which a platform running Condor can report its state to a challenger and thus allow them to determine whether to trust the platform or not. This design makes the following decision:

Design Decision 3 (DD3): Host-Host Attestation Versus Host-CM attestation

Option 1. Host-Host Attestation

This option implies that given hosts A and B where A initiates a communication with

B, then A has to attest its platform state to B. This allows hosts within a pool to develop trust amongst each other and thus eliminating the need for trusting what a third party (CM in option 2) says. The disavatanges are that it places an overhead on the communication because of the high number of interactions that occur between any two hosts and it requires each host to keep a record of the trust metrics from all the other hosts on the pool, which is very difficult in terms of management.
Option 2. Host-CM attestation
This option implies that every host attests to the CM on its pool. The advantage is that the metrics required for attestation would only be exported to the CM and also avoids the need for the hosts to attest to each other when they communicate. It also means that the hosts will need to trust what the CM says about other hosts on the pool. This may be a problem if the CM gets compromised but could be solved by forcing the CM to attest to each of the host, achieving what can be referred to as mutual attestation between every host and the CM. This design settles on this option because it lessens on any privacy concerns and also works well with the current infrastructure where all hosts send information to the CM, which uses that to perform match-making.

5.2.1 Goals

1. provide a mechanism that allows a remote host to believe that cryptographic keys were not compromised

2. provide a mechanism that provides a guarantee to the CM that a remote host is running Condor

3. provide a mechanims that securely ensures that software required for running jobs exists on a remote platform

4. cryptographically identify the platform to the CM

5.2.2 Host Enrollment - Prepare for Attestation

This process prepares a host on the grid to attest its state to other hosts. It involves creating keys/certificates and exporting the attestation symbols/trust metrics.

Key & Certificate Creation

In section 4.3.8, it was discussed that attestation requires the attesting platform to be in possession of a valid private/public key pair and a certificate. This information is used to encrypt/decrypt data from the TPM in such a way that the challenger can

prove firstly that it was generated by a valid TPM and secondly that the platform is in a trustworthy state. The following describes how the keys and certificate are created.

1. Platform generates an AIK public/private key pair.
2. Platform sends public component of the AIK to a certification authority
3. Certification authority verifies that the key is non-migratable
4. CA issues and sends a certificate indicating the identity of the platform, the public key and the CA's signature

The advantage of this approach is that the private key never leaves the TPM. Alternatively, the CA can generate both the private and public keys and then force the platform to store the private key inside the TPM. An implication of this approach is that the CA keeps a copy of the private key and thus risks exposing it, if compromised.

Exportation and Storage of Attestation Symbols/Trust metrics

Having generated the credentials necessary for attestation, the next step is to export the symbols or trust metrics against which the state of the platform can be evaluated later. This includes the SML, see section 4.3.3, and the PCR values. This step would be carried out before the PCRs are reset during the trusted installation step discussed in section 5.1.2.

Once the symbols have been identified and exported from the platform, they need to be sent to the potential challengers. However, a carefull decision, DD3, has to be made to choose the hosts on the grid that would store this information to avoid privacy infringement. This results in metrics, against which trust can be evaluated, stored on some host(s) on the grid.

5.2.3 Attestation Service

The attestation service relies on assumption ASMP-1, and extends the integrity measurements to include the entire software stack running on a platform. It suggests an addition to the classad to include the results of the quote operation to allow a remote host to verify the state of a platform. This is depicted in figure 5.4.

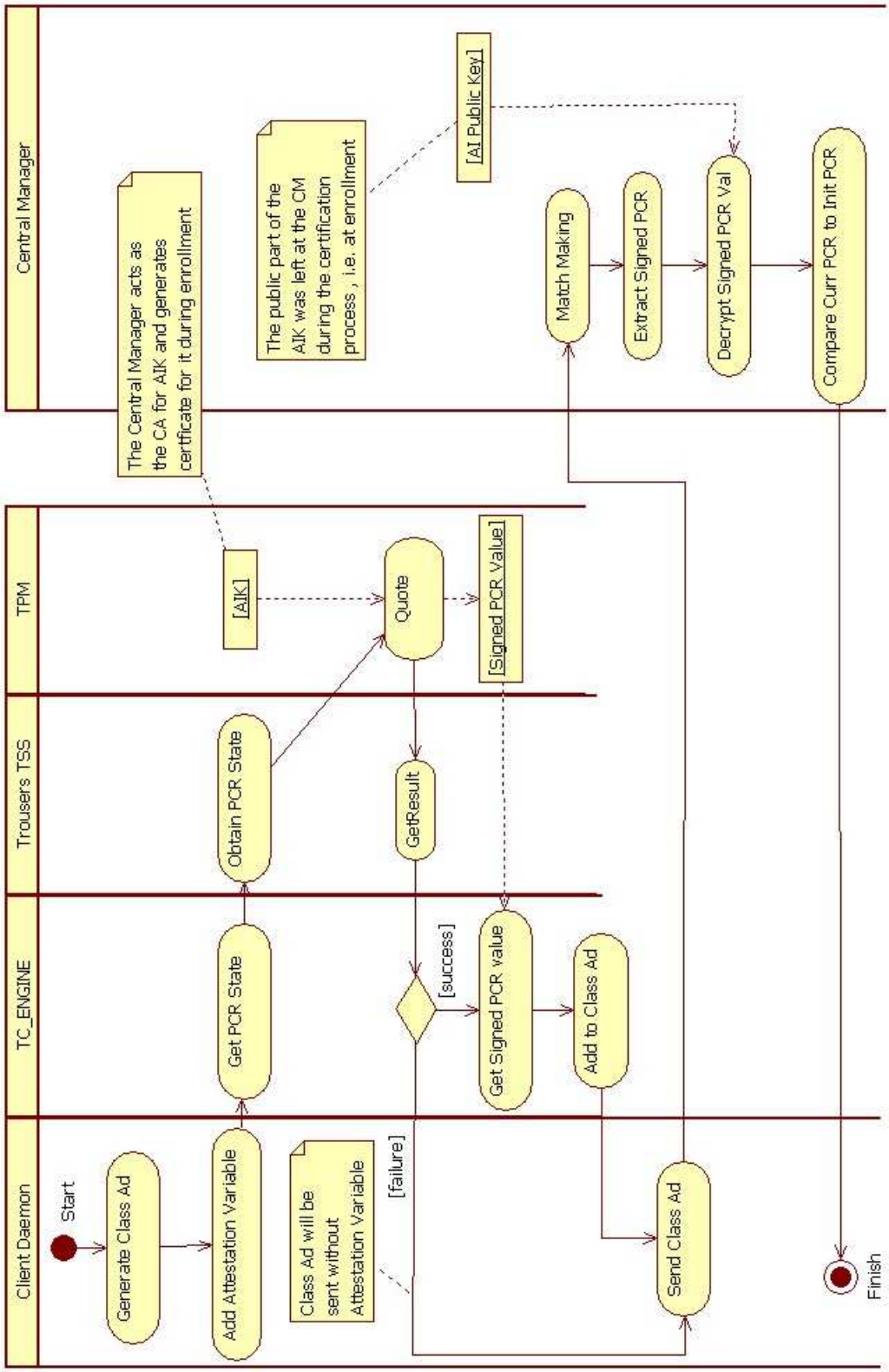


Figure 5.4: Platform Attestation

Attesting the State of Communicating Daemons

Classads, discussed in section 2.2.2 are a means by which Condor daemons send information about their status to other daemons, specifically to the collector daemon running on the central manager. In this design the classad is extended to include integrity measurements that can be used to attest the state of the daemons involved in the communication. Though the attestation protocol described in section 4.3.8 indicates that a remote host generates a nonce and compares it later, the nonce can also be generated by the attesting platform because it only ensures freshness of the message to avoid it being replayed, if stolen.

This extension to the classad requires the daemon to report the state of the PCR it is assigned to using the Quote operation described in section 4.4.3, which returns the digest values of the PCRs used by the particular daemon together with a nonce (generated on the platform to avoid communication overhead) signed with the private part of the AIK. This information is included in the classad together with the rest of the information, as described in section 2.2.2.

Platform Software Stack Attestation

The process described in the above section works well if the state of a particular daemon (or a set) is of interest. Sometimes, it might be necessary to take note of the state of the entire platform. For example to securely enforce software requirements, it is essential that the entire process that took a platform to a given configuration be stated.

The protocol described in section 4.3.8, shows that both the hardware of the platform and the software stack running on it can be reported. In order to report the entire software stack running on a platform, integrity measurements have to be taken on all the software before it is loaded. This is enforced by the trusted operating system assumed in ASMP-1. The result is an SML that shows all the software loaded on the platform and the order in which they were loaded. This information is sent together with the information from the qoute operation discussed in section 4.4.3 so that the challenger, in this case the CM, can use it in the verification process. This implies an addition of the SML to the second part of the protocol given in section 4.3.8 as follows.
$2 : A \rightarrow B : \{S, H, nb\}PrkA, cert, SML$

In this case, S implies the integrity measuremnets of the entire software stack running on the platform.

Verification Process

The verification process involves firstly checking that the data has been recieved from a valid TPM and secondly comparing the data sent with the one that was exported from the given platform. To verify the origin of the data and the validity of the TPM requires that the challenger determine if the key used to sign the data is stored in a TPM. Since the challenger is the CM, which also issued a certificate, it becomes easier for it to verify the authenticity of the certificate because it doesn't have to consult another party.

Once the data has been verified to have come from a valid TPM, it is decrypted using the public component of the AIK. This results in a set of PCR digest values as well as a nonce. These value are compared with the initially exported integrity measurements and if they match, the platform is considered to have successfully attested its state. This process, so far, compares the resulting values in the PCRs, which is ok for checking the state of the daemons. To verify the state of a complete platform, that is to determine the software stack running on it, requires that the digest values be compared to not only the exported values but to those stored in the SML as well. This is not an easy step because of the many possible configurations of the software stack, but can be performed by following a mechanism proposed by Bottoni et al [3], where each layer in the software stack is authenticated before authenticating the entire stack. This process is made as an extension to the match-making process described in section 2.5 so that hosts on the pool can be sure that the hosts they communicate with (send jobs to or recieve jobs from) are in a trustworthy state and also ensure that software requirements for their jobs will be met.

5.3 Putting it All Together

The preceeding sections described the components that ensure that integrity measurements for a platform are performed in a manner that cannot be subverted and that the credentials are protected from malicious code. The hosts can also attest to the CM about their state. So how does this whole process fit together? The section below describes the resulting pool and how that fits with the rest of the campus grid.

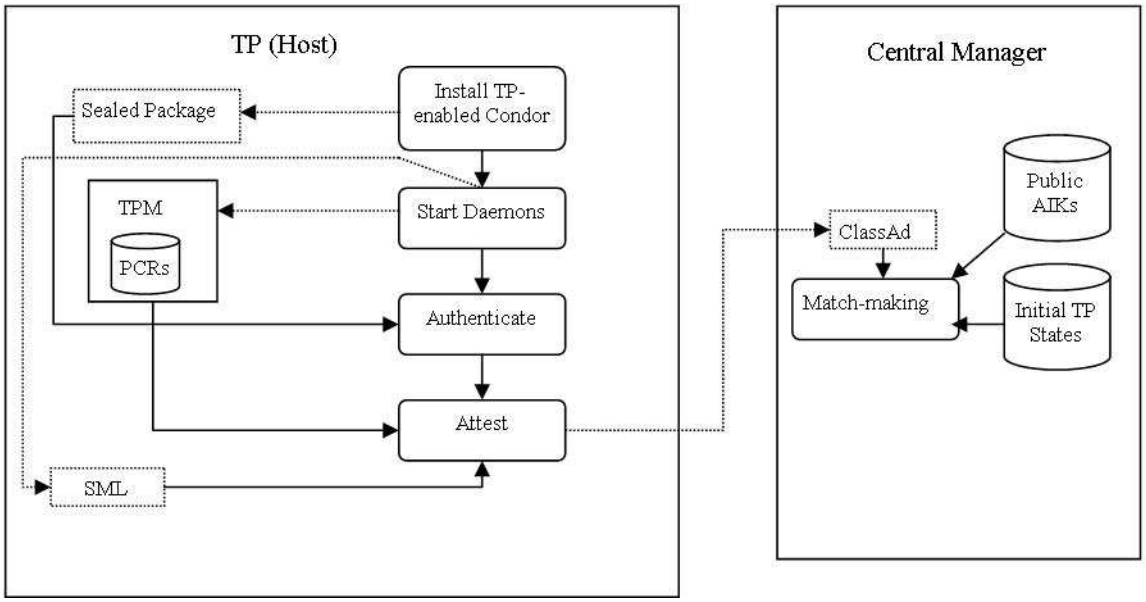


Figure 5.5: Interaction between a host and a CM in a trusted condor pool

5.3.1 A Trustable Condor Pool within OxGrid

A Condor pool will be composed of multiple hosts each running a TCG specified TPM, a TSS, a trusted operating system and Condor installed in a trusted mode. At enollment, each host also generates an AIK certified by the CM and exports its configuration to the CM. Trusted mode for Condor would have resulted in the credential sealed to the PCR values that have been reset. The diagram in figure 5.5 illustrates the interactions between one host and a CM within a pool. This solution suggests that trustworthiness is treated as boolean attribute so that a platform is either trustworthy or not. Meaning that the platforms evaluated not to be trustworthy will not be able to interact with other hosts on the pool. The resulting grid has a number of pools each with hosts that are able to attest their state to the CM on their particular pool. The CM on each pool can then inform other CMs about the state of a particular host on its pool, if needed. Therefore a trusted infrastructure for the campus grid is achieved by running hosts that protect their credentials from malicious code, are cryptographically identifiable to the CM and are able to report their platform state to the CM which can consequently report to other CMs on other pools.

Chapter 6

Implementation

This chapter describes the implementation of a prototype that demonstrates the practicallity of the design. It describes a library built using C++ that can be integrated with Condor, giving code snippets of the most important functions and showing how they can be used within Condor. The complete source code of the library can be obtained from the author. The class diagram in figure 6.1 shows the classes in the library and how they fit together. The library is divided into two parts, an abstraction layer that exposes TPM capabilities and the TC_ENGINE which serves an interface to an application wishing to operate in a TP environment, in this case Condor. The TC_ENGINE implements the essential algorithms required to complete a particular TP operation. For example, to seal credentials, the TC_ENGINE knows to first read the credentials before sending the bytes to an object in the abstraction layer and saves the sealed package to the appropriate directory. Details of the abstraction layer and the TC_ENGINE are presented in the following sections.

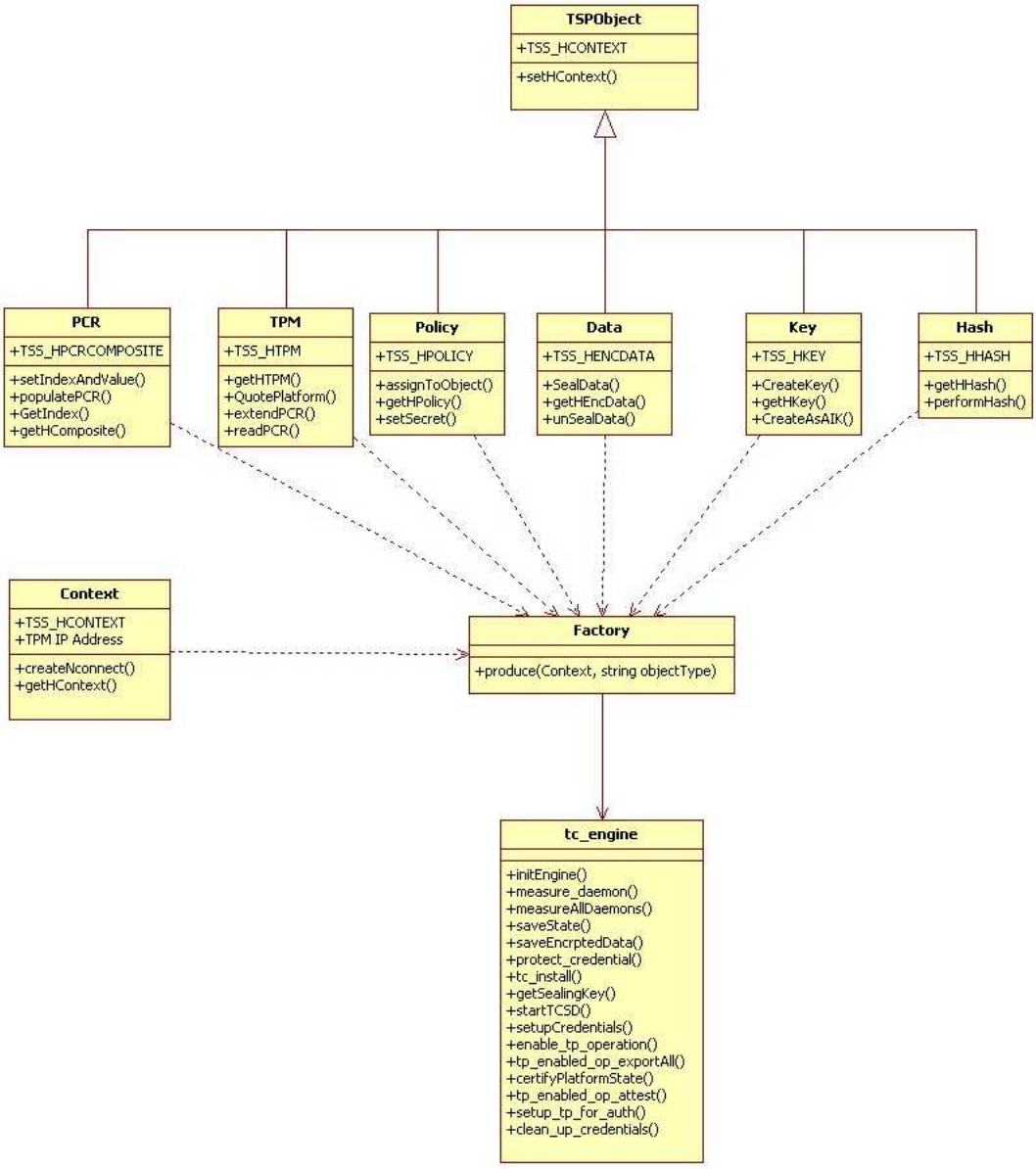


Figure 6.1: Class diagram for TC_ENGINE library

6.1 Wrapper classes for TSP Object: Abstraction layer

In section 4.4.3, it was discussed that the TSP library provides objects to the programmer through which trusted platform functionalities can be achieved. This implementation builds a number of wrapper classes for those objects to serve as an abstraction layer to Condor so that it does not become aware about the existence of the TPM, allowing it be usable in a different environment. The following sections

describe the wrapper classes.

6.1.1 Context

The Context class serves as a wrapper class for the TSS_HCONTEXT object, a TSP defined object that maintains a connection to a TPM on the local platform and provides mechanisms for creating any other objects that expose TPM functionalities. A TSS_HCONTEXT object is created as shown in listing 6.1.

Listing 6.1: Create and Connecting a Context

```
bool createNconnect()
{
        result = Tspi_Context_Create(&hContext);
        if(result != TSS_SUCCESS){
                Log("Context Create Failed With Code: " + ERROR_CODE(result));
                return false;
        }
        else{
                result = Tspi_Context_Connect(hContext, NULL);
                if(result != TSS_SUCCESS){
                        Log("Context Connect Failed With Code: " + ERROR_CODE(result));
                        return false;
                }
        }
        return true;
}
```

Listing also shows the error handling code which will be used in the same way throughout the library, that is, the result of the operation is checked, if not successful, the error is logged and the operation stopped. Once the context has been created, it is connected using the Tspi_Context_Connect also shown in listing 6.1.

6.1.2 PCR

The PCR class is a wrapper for the TSS_HPCRS, which holds indices and digest values of the PCRs, useful in sealing. It exposes the populatePCR function, shown in listing 6.2 which populates the TSS_HPCRS object with values to be used in the sealing process.

Listing 6.2: Preparing a PCR Composite for use in sealing

```
        /*
                Populates the hpcrs object with the values stored in the given PCR indices
                Algorithm:
                        Read the value of the pcr register with given index
                        for each index, set the index and the value of the register in the HPCRS object
        */
        bool
        populatePCR(UINT32 num_pcrs, UINT32 *pcrs, TPLM *tpmObj){
                UINT32 numPcrs, subcap, i, ulPcrValueLength;
                BYTE *rgbPcrValue, *rgbNumPcrs;

                TSS_HTPM htpm = tpmObj->getHTPM();
```

```
        subcap = TSS_TPMCAP_PROP_PCR;
        Tspi_TPM_GetCapability(htpm, TSS_TPMCAP_PROPERTY, sizeof(UINT32),
                                    (BYTE *)&subcap, &ulPcrValueLength, &rgbNumPcrs);
        numPcrs = *(UINT32 *)rgbNumPcrs;

        for(i = 0; i < num_pcrs; i++){
                if(pcrs[i] >= numPcrs){ return -1;}
                htpm->readPCR(pcrs[i], &ulPcrValueLength, &rgbPcrValue);
                result = Tspi_PcrComposite_SetPcrValue(hPcrComposite, pcrs[i], ulPcrValueLength, rgbPcrValue);
                if(result != TSS_SUCCESS){
                        Log("Setting Digest Value Failed With Code: " + ERROR_CODE(result));
                        return false;
                }
                PCR_INDEX = pcrs[i];
                hashValue = rgbPcrValue;
        }
        return true;
}
```

6.1.3 Key

The Key class provides functions for creating a TSS_HKEY object, loading a key into the TPM, registering the key and unloading the key. These functions work on the TSS_HKEY object wrapped within the class. To create a key, the SRK is first loaded, using Tspi_Context_LoadKeyByUUID and then a 2048-bit RSA key is generated as a child of the SRK. This class is used mainly during sealing, unsealing and at the key creation stage in attestation. Listing 6.3 shows how a key is created as a child of the SRK.

Listing 6.3: Creating and loading a key to be used by a TPM

```
/*
        Creates a key as child of the SRK, sets the necessary policies
        and loads into the TPM
 */
bool
CreateKey(Policy *usagePolicy){
        TSS_UUID SRK_UUID = TSS_UUID_SRK;
        result = Tspi_Context_LoadKeyByUUID( TSPObject::getContext(), TSS_PS_TYPE_SYSTEM,
                                                        SRK_UUID, &hSRK );
        if(result != TSS_SUCCESS){
                Log("SRK Load Failed with Code: " + ERROR_CODE(result));
                return false;
        }
        else{
                if(usagePolicy->assignPolicyToObject(this)){
                        result = Tspi_Key_CreateKey( hKey, hSRK, NULL );
                        if(result != TSS_SUCCESS) {
                                Log("Create Key Failed with Code: " + ERROR_CODE(result));
                                return false;
                        }
                        else{
                                result = Tspi_Key_LoadKey(hKey, hSRK);
                                if(result != TSS_SUCCESS) {
                                        Log("Create Key Failed with Code: " + ERROR_CODE(result));
                                        return false;
                                }
                        }
```

```
                }
                else return false;
        }
        return true;
}
```

6.1.4 TPLM

This class wraps a TSS_HTPM object (a TSP object that represents a particular TPM) and provides functionalities for reading the registers of a given TPM, performing the extend operation and quoting the status of the platform, shown in listing 6.4. hTPM represents the wrapped TSS_HTPM object.

Listing 6.4: TPLM class' read

```
        /*
         Reads the specified PCR index,
         ulPcrValueLength = length of the value read
         rgbPcrValue = data read
        */
        public: bool
        readPCR(int index, UINT32 *ulPcrValueLength, BYTE* rgbPcrValue){
                result = Tspi_TPM_PcrRead( hTPM, index, ulPcrValueLength, &rgbPcrValue );
                if(result != TSS_SUCCESS){
                        Log("PCR Read Failed With Code: " + ERROR_CODE(result));
                        return false;
                }
                return true;
        }
/*
         Extends the given PCR index with the given data
         It also creates an event which keeps track of the events on the PCR
        */
        public: bool
        extendPCR(int PCRIndex, BYTE *pcrValue, BYTE* NewPcrValue){
                TSS_PCR_EVENT event;
                memset(&event, 0, sizeof(TSS_PCR_EVENT));
                event.ulPcrIndex = PCRIndex;
                result = Tspi_TPM_PcrExtend(hTPM, PCRIndex, sizeof(pcrValue),
                        pcrValue, &event, &ulNewPcrValueLength, &NewPcrValue);

                if(result != TSS_SUCCESS){
                        Log("Extend Failed With Code: " + ERROR_CODE(result));
                        return false;
                }
                return true;
        }

        /*
          Quotes the state of the platform represented by the indices in
          the pcrObj
         */
        BYTE* QuotePlatform(BYTE* nonce, Key *key, PCR *pcrObj){
                BYTE* certifiedValue;
                TSS_RESULT result;
                TSS_HKEY AIK = key->getHKey();
                TSS_HPCRS hPcrComposite = pcrObj->getHPCRS();
                TSS_VALIDATION vData;
                vData.rgbExternalData = nonce;
                vData.ulExternalDataLength = 20;
                result = Tspi_TPM_Quote(hTPM, AIK, hPcrComposite, &vData);
                if (result != TSS_SUCCESS) {
```

```
                Log("Platform Quote Failed with Code: " + ERROR_CODE(result));
                return NULL;
        }
        else{
                certifiedValue = vData.rgbValidationData;
                return certifiedValue;
        }
}
```

6.1.5 Hash

The TSS provides the TSS_HHASH object which is required to hold the results of cryptographic hashing. The main function it provides, performHash shown in listing 6.5, computes the hash of the bytes passed to it and stores the result in the wrapped TSS_HHASH object. The wrapped object can be obtained by calling the getHHash function.

Listing 6.5: Cryptographic hashing of data

```
/*
        Hash the given data and store it in the wrapped TSS_HHASH object
 */
bool
performHash(BYTE* unHashed){
        UINT32 size = sizeof(unHashed);
        result = Tspi_Hash_UpdateHashValue(hHash, size, unHashed);
        if (result != TSS_SUCCESS) {
                Log("Hashing Failed with Code: " + ERROR_CODE(result));
                return false;
        }
        return true;
}
```

6.1.6 Policy

Policy wraps a TSS_HPOLICY object and provides functions for setting the secrets, that is the passwords, and assigning the policies to various objects, used mainly in the management of keys. The most function it offers is the assignPolicyToObject shown in listing 6.6, assigns a usage policy to a given object.

Listing 6.6: Assiging a usage policy to an object

```
/*
        Assigns the policy to a given Object

 */
bool
assignPolicyToObject(TSPObject objToAssign){
        result = Tspi_Policy_AssignToObject(hPolicy, objToAssign);
        if(result != TSS_SUCCESS){
                Log("Assign to object Failed with Code: " +ERROR_CODE(result));
                return false;
        }
        return true;
}
```

6.1.7 Data

The Data class is a wrapper class for the TSS_HENCDATA object. It exposes functionality that work on the TSS_HENCDATA such as sealing, and unsealing, both of which are very important in protecting the credentials, shown in listings 6.7. To perform sealing, pass the data, the key to be used for sealing (created using the functions from the Key class), the PCRs that should be used to seal the data (prepared using the PCR class functions) and the file path where the sealed data blob would be saved. To unseal, the sealed package is passed together with a key for unsealing to the unsealdata function. The wrapped TSS_HENCDATA will hold the encrypted data in both operations.

Listing 6.7: Data class Seal and Unsealfunctions

```
/*
        Unseal the given encrypted data blob
        return value of NULL implies platform state not correct
 */
BYTE*
UnSealData(BYTE *SealedDataBlob, UINT32 sealedDataLen, TSS_HKEY hKey){
        TSS_HKEY hKey = key->getHKey();
        BYTE *unsealedDataBlob;
        UINT32 unsealedDataLength;

        result = Tspi_SetAttribData(_hEncData, TSS_TSPATTRIB_ENCDATA_BLOB,
        TSS_TSPATTRIB_ENCDATABLOB_BLOB, sealedDataLen, SealedDataBlob);
        if(result != TSS_SUCCESS){
                Log("Set EncData Failed with Code: "+ ERROR_CODE(result));
        }
        else{
                result = Tspi_Data_Unseal(_hEncData, hKey, &unsealedDataLength, &unsealedDataBlob);
                if(result != TSS_SUCCESS) {
                        Log("Unseal Failed with Code: "+ ERROR_CODE(result));
                        return NULL:
                }
        }
        return unsealedDataBlob;
}

/*
        Seals the given data with the given key and the state of the platform
        specified in the pcr object and saves the encrypted to file pointed
        to by fullFilePath
 */
bool
SealData(BYTE  *rgbDataToSeal, int datasize, PCR *pcrObj, Key *key,
                 bool is, const char* fullFilePath ){

                TSS_HCONTEXT hContext = TSPObject::getContext();
                TSS_HKEY hKey = key->getHKey();
                UINT32 tmp_out_size;
                BYTE *tmp_out;
                UINT32  ulDataLength = ((2048/8)-40-2-65);//maximum allowed data size
                TSS_RESULT result;
                TSS_UUID SRK_UUID = TSS_UUID_SRK;
                ofstream outfile;

                result = Tspi_Data_Seal( _hEncData, hSRK, ulDataLength, rgbDataToSeal, pcrObj->getHPCRS());
                if ( result != TSS_SUCCESS )
                {
```

```
                Log("Seal Failed with Code: "+ ERROR_CODE(result));
                return false;
        }
        else{
                result = Tspi_GetAttribData(_hEncData, TSS_TSPATTRIB_ENCDATA_BLOB,
                TSS_TSPATTRIB_ENCDATABLOB_BLOB, &tmp_out_size, &tmp_out);
                if (result != TSS_SUCCESS) {
                        Log("Seal Failed with Code: "+ ERROR_CODE(result));
                        return false;
                }
                else{
                        outfile.open(fullFilePath,ios::out);
                        if( outfile.is_open() ) {
                                outfile.write ((char*)tmp_out,tmp_out_size);
                                outfile.close();
                        }
                        else{
                                Log("Seal Failed with Code: "+ ERROR_CODE(result));
                                return false;
                        }
                }
        }
        return true;
}
```

6.2 TC_ENGINE Module

The TC_ENGINE is a module that saves as an interface to the functionality offered by a TSS. It exposes the features of a trusted platform in a manner that does not make Condor aware about the underlying trusted platform operations. It does this by providing public functions that perform operations using the abstracted layer built by the wrapper classes described above. The following are the public functionality it exposes:

6.2.1 Measure Daemon

To measure a daemon, the daemon executable is read, using the standard C++ libraries, to get the bytes. The bytes are then passed to the TSS, using the Data class' performHash operation. The value in the Hash object is then obtained and passed to the TPM to the extendPCR of the TPLM object. Before the value is extended, the relationship of the daemon to other daemons is checked to see if it has a parent, if it has, the extend operation is performed twice, once with the parent SHA1 value and then with the child SHA1 value. This is done to create a link between the parent and child daemons so that compromising either of them would result in the whole process failing. This is defined in a public function called measure_daemon, listing 6.8, which can be called at any point by passing it the name of the daemon and the executable file path.

Listing 6.8: Measuring the state of daemon and extending into PCRs

```
/*
        Save the state of a given daemon
        Algorithm:
                Check if daemon is master or child of another daemon,
                        then get the parent and extend it to the childs pcr
                if not, then simply extend the daemon only,
*/
PCR*
saveState(const char* daemonName, BYTE* daemonHashBlob){
        bool isOK = false;
        bool hasParent = checkPaternity(daemonName);
        TPLM *tpmObj = (TPLM*)factory.produce(context, "TPLM");
        BYTE* parentPCRValue;
        BYTE* daemonPCRValue;
        int daemonPCRIndex = getDaemonPCRIndex(daemonName);

        if(hasParent){
                parentPCRValue = getParentPCR(daemonName);
                isOK = tpmObj->extendPCR(daemonName, daemonPCRIndex, parentPCRValue, daemonPCRValue);
            if(isOK){
                isOK = tpmObj->extendPCR(daemonName, daemonPCRIndex, daemonHashBlob, daemonPCRValue);
                if(!isOK) return NULL;
        }
        else{
                isOK = tpmObj->extendPCR(daemonName, daemonPCRIndex, daemonHashBlob, daemonPCRValue);
        }
        if(!isOK){
                Log("ERROR: EXTENDING DAEMON PCR:");
                return NULL;
        }

        //create a PCR object, populate it with the daemonPCR Value and return it
        PCR *pcrObj = (PCR*)factory.produce(context, "PCR");
        UINT32 num = 1;
        UINT32 rg[] = {daemonPCRIndex};
        pcrObj->populatePCR(num, rg, tpmObj);
        return pcrObj;
}

/*
        measures the state of a given daemon executable
        Algorithm:
                get the BYTEs of the executable given
                send to TSS to perform SHA1 operation
                call getSHA1 value to get the value
                then save the measurement using the extend operation
*/
bool TC_ENGINE::measure_daemon(const char* daemon, const char* executableFilePath){
        bool isOK = false;
        BYTE* contents = (BYTE*) readBinary(executableFilePath);
        executableFilePath = NULL;
        if(!contents){
                Log("ERROR: READING EXECUTABLE");
                return false;
        }

        //if we ever get here, then everything went well, so we are ready to perform hash
        const char* type = "HASH";
        Hash* hashingObj = (Hash*)factory.produce(context, type);
        isOK = hashingObj->performHash(contents);
        if(!isOK){
                Log("ERROR: HASH FAILED");
                return false;
        }
        else{
                BYTE* hashedDaemon = hashingObj->getSHA1Value();
```

```
                PCR *pcrObject = saveState(daemon, hashedDaemon);
                addDaemonPCR(daemon, pcrObject);
                Log(" MEASUREMENT FINISHED");
                delete hashedDaemon;
        }
        delete hashingObj;
        return true;
}
```

6.2.2 Protect Credential

Protection of credentials occur after the platform has been measured. This creates a link between the platform state, the credentials and a key generated by the TPM. This is implemented in a public function called protect_credentials, listing 6.9, which gets the name of the daemon, reads its credential file and sends to the Data object for sealing. Because sealing has a maximum data size determined by s/8 - (40-2-65), where s is the key size (in this implementation 2048), the credential file need to be split into sizes of 149bytes and each part sent for sealing. This function saves sealed packages in the same directory in which the original credentials were stored.

Listing 6.9: Protection credentials with digest values in PCRs

```
/* Protect credentials for the given daemon, with a Key
 * and the digest values in the PCRs
 * Algorithm:
                Get the credential and the pcr from the cred_pcr_map
                create a pcr object and specify its pcr values
                get the BYTEs of the given credential
                Call the tss seal operation which returns an encrypted credential
                save the credential in user persistant storage
                clean up the credentials
 */
bool TC_ENGINE::protect_credentials(const char* daemonName){
        map<const char*,BYTE*>::iterator iter;
        map<const char*,PCR*>::iterator pcriter;
        char* keyUUID = getKeyUUID(daemonName);
        Data *dataObj = (Data*)factory.produce(context, "DATA");
        Key *key = (Key*)factory.produce(context, "KEY");
        bool isOK = key->CreateKey();
        if(isOK){
                const char* credPath = getFullCredPath(daemonName);
                pcriter = daemon_pcr_map.find(daemonName);
                PCR *pcrObj = pcriter->second;
                ifstream myExecFile (credPath, ios::in | ios::ate);
                ifstream::pos_type size;

                if (myExecFile.is_open())
                {
                        size = myExecFile.tellg();
                        int partCount = 0, toread = 0;
                        if(size >= 149) toread = 149;
                        else toread = size;
                        myExecFile.seekg (0, ios::beg);
                        int remaining = size, readsofar = 0, sizeInt = size;
                        char contents[toread];

                        while(myExecFile.read (contents, toread) != 0){
                                string newpath = credPath;
```

```
                                newpath = newpath + "." + (partCount+1);
                                newpath = newpath + ".enc";
                                isOK = dataObj->SealData((BYTE*)contents, toread, pcrObj, key, true, newpath.c_str());
                                if(isOK){
                                    partCount++;
                                    myExecFile.seekg (partCount*149, ios::beg);
                                    readsofar = readsofar + toread;
                                    remaining = sizeInt - readsofar;
                                    if(remaining >= 149 )  toread = 149;
                                    else toread = remaining;
                                    saveSealInfo(daemonName, newpath.c_str(), pcrObj->GetIndex(), keyUUID);
                                   ::memset(contents,toread,sizeof(contents));
                                }
                                else{
                                        myExecFile.close();
                                        return false;
                                }
                                if(remaining == 0)break;
                        }
                        myExecFile.close();
                }
                else {
                        Log("ERROR: READING FILE >>");
                        return false;
                }
        }
        else{
                Log("ERROR: CREATING SEALING KEY");
                return false;
        }
        ok = cleanup_credentials(daemonName);
        if(!ok) return false;
        return true;
}
```

6.2.3 Setup Credential

Setting up credentials involves getting the sealed parts of a given credential (produced using the protect credentials) and presenting each of them to the Data class' UnsealData function and then placing the unsealed data in a directory accesible to the underlying authentication method. It is implemented in a public function setup_tp_for_auth, listing 6.10, which takes care of concatenating each of the unsealed parts of the credential.

Listing 6.10: Setting up credentials for authentication

```
/*
 * Gets the sealed credentials unseals then and puts them in the approproate directories
 * Algorithm:
 *    Read the sealinfo file for this deamon line by line,
 *    For each line, get the enc filename part for the daemon
 *    Read the bytes from the part
 *    send for unsealing
 *    if successful, append to the daemon cred, as specified in the cred_path_map
 *    else return failure
 */
bool
TC_ENGINE::setupCredentials(const char* daemonName){
        bool isOK = false;
        const char* sealfilepart;
```

```
        UINT32 unsealedDatasize = 0;
        BYTE* sealedData;
        BYTE* unsealedData;
        UINT32 sealedDatasize;

        string infopath = "../sealInfo.data";
        infopath = infopath + ".";
        infopath = infopath + daemonName;

        Data *dataObj = (Data*)factory.produce(context, "DATA");

        ifstream::pos_type size;
        ifstream mySealFile(infopath.c_str() );
        string temp;
        const char* daemonCredPath = getFullCredPath(daemonName);
        string newcredpath = daemonCredPath;
        newcredpath = newcredpath + ".unsealed";
        ofstream outfile (newcredpath.c_str(),ofstream::out);

        if (mySealFile.is_open()){
                while( getline( mySealFile, temp ) ){
                    sealfilepart = temp.c_str();
                        sealedData = (BYTE*)readFile(sealfilepart, true);//read the sealedpart
                        sealedDatasize = (UINT32)getFileSize(sealfilepart, true);

                        isOK = dataObj->UnSealData(sealedData, sealedDatasize, &unsealedData, &unsealedDatasize, NULL);//try unsea
                        if(isOK){
                                //the platform is correct, write unsealed data to file
                                outfile.write ((char*)unsealedData, unsealedDatasize);
                        }
            }
                mySealFile.close();
        }
        else{
                mySealFile.close();
                        return false;
        }
        outfile.close();
        return true;
}
/*
 * Starts the engine and then sets up the credentials
 */
bool TC_ENGINE::setup_tp_for_auth(const char* daemonName){
        bool ok = initEngine();
                if(ok){
                        ok = setupCredentials(daemonName);
                }
                return ok;
}
```

6.2.4 Cleanup Credential

Credential cleanup is implemented through a function called cleanup_credentials, listing 6.11, which simply deletes the unencrypted files from the file system.

Listing 6.11: Potential classad modification

```
/*
 * Delete the credentials for the given daemon
 * Get the defined path for the credential and call remove
 */
bool cleanup_credentials(const char* daemonName){
        const char* credPath = getFullCredPath(daemonName);
```

```
        if (remove(credPath.c_str( )) !=0)
                return false;
        else return true;
}
```

6.2.5 Enabling TP Operation

Enabling trusted platform involves starting the trousers daemon (tcsd), which enables communication with the TPM, measuring the state of each daemon (measure_daemon) and protecting the credentials (protect_credentials). The cleanup_credentials function is then called to remove the un-encrypted credentials. This is implemented in the TC_ENGINE.enable_tp_operation, listing 6.12.

Listing 6.12: Enabling TP operation

```
/*
 * Enables a TP environment by measuring the state of each daemon
 * And protecting the credentials with the measured values
 * System has to match the state before credetials can be given away
 * Algorithm:
 *    Initialise the engine to start the tcsd daemon and prepare a list
 *    of daemons to measure
 *    measure the master daemon
 *    measure the other daemons by appending the value of the master
 *    Protect the credentials for each daemon
 */
bool TC_ENGINE::enable_tp_operation(){
        bool ok = initEngine();
        if(ok){
                measure_daemon("master", getFullPath("master"));
                measureAllDaemons();

                /*************************************************************/
                /* NOTE: THIS IMPLEMENTATION ASSUMES THAT THE CREDENTIALS
                        HAVE ALREADY BEEN OBTAINED AND PLACED IN THE DEFAULT
                        DIRECTORIES. OTHERWISE, WE WOULD HAVE TO REQUEST THE
                        CREDENTIALS AT THIS POINT AND HOLD UNTIL WE HAVE THEM
                *************************************************************/
                ok = protect_credentials("master");
                ok = protect_credentials();//All other daemons
                if(ok){
                        //we are ready to clean up
                        cleanUpCreds();
                }
                else{
                        Log("Credential Protection Failed!");
                        return false;
                }
        }
        else{
                Log("TCSD Start Failed!");
                return false;
        }
        return true;
}
```

6.2.6 AIK Generation

The AIK generation was implemented following the method given by Challener et al [6], were a function recieves an openssl key from the CA, creates a TPM key from it by getting the public 'n' and the number of primes and then creates an identity request that is sent to the CA. The reader is referred to [6] for the source listing.

6.2.7 Exporting Initial Platform State

Exporting the platform state is implemented in the function tp_enabled_op_exportall, listing 6.13 which creates a TPLM object to reads all the PCRs and returns them in a human-readable format.

Listing 6.13: Exporting the state of the platform after installation

```
/*
 * Exports the values stored in the PCRs 7 to 16
 * Assumes that the 1-6 are reserved and only 7-16 are
 * available for the applications.
 */
string tp_enabled_op_exportall(){
        string state;
        TPLM *tpmObj = (TPLM*)factory.produce(context, "TPLM");
        UINT32 *ulPcrValueLength;
        BYTE* rgbPcrValue;
        for(int i=7; i <= 16; i++){
                tpmObj->readPCR(i, ulPcrValueLength, rgbPcrValue);
                state =+ (string)rgbPcrValue;
        }
        return state;
}
```

6.2.8 Platform Attestation

Platform attestation was implemented in the function tp_enabled_op_attest, listing 6.14, which first determines what indices to quote and creates and loads an AIK before passing this to the TPLM.QuotePlatform function. The resulting bytes represent a value that has been signed with the private component of the AIK and can be sent to a remote host which can compare with the measurements exported initially.

Listing 6.14: Functions to enable attestation

```
/*
 * Gets the state of the PCRs 7 - 16, by calling Quote Platform which
 * signs the state with AIK
 */
BYTE* TC_ENGINE::certifyPlatformState(){
        PCR *pcrObj = (PCR*)factory.produce(context,"PCR");
        for(int i=7; i <= 16; i++){
                pcrObj->selectIndex(i);
        }

        TPLM *tpmObj = (TPLM*)factory.produce(context, "TPLM");
        BYTE* nonce = generateRandomNonce();
```

```
        Key *AIKey = (Key*)factory.produce(context, "KEY");
        AIKey->CreateAsAIK();
        BYTE* signedval = tpmObj->QuotePlatform(nonce, AIK, pcrObj)
        return signedval;
}
/*
 * Gets the state of the PCR being used by the specified daemon
 */
BYTE* TC_ENGINE::certifyPlatformState(char const* daemonName){
        PCR *pcrObj = (PCR*)factory.produce(context,"PCR");
        int daemonPCRIndex = getDaemonPCRIndex(daemonName);
        pcrObj->selectIndex(daemonPCRIndex);

        TPLM *tpmObj = (TPLM*)factory.produce(context, "TPLM");
        BYTE* nonce = generateRandomNonce();
        Key *AIK = (Key*)factory.produce(context, "KEY");
        AIK->CreateAsAIK();
        BYTE* signedval = tpmObj->QuotePlatform(nonce, AIK, pcrObj)
        return signedval;
}
/*
 * Starts the Engine and gets the quote of the platform state
 */
BYTE* TC_ENGINE::tp_enabled_op_attest(char const* daemonName, bool qouteEntirePlatform){
        bool ok = initEngine();
        if(ok){
                if(qouteEntirePlatform)  return certifyPlatformState();
                else return certifyPlatformState(daemonName);
        }
        else{
                Log("Engine Start Failed");
                return NULL;
        }
}
```

6.3 Library Usage & Testing

This section describes how the library can be used with Condor and some of the tests that were carried out with the Condor daemons.

6.3.1 Performing Trusted Installation

Performing an installation involves measuring each of the Condor daemons and then protecting their credentials. To test this, a script, shown in listing 6.15, was developed that executes the TC_ENGINE.enable_tp_operation with the list of deamons (names and locations) and their relationships defined as shown in the listing.

Listing 6.15: Script to perform trusted installation

```
map <const char*, const char*, strCmp> daemon_child_parent_map;
map <const char*, const char*, strCmp> daemon_credpath_map;
map <const char*, const char*, strCmp> daemon_exepath_map;
map <const char*, int, strCmp> daemon_pcr_index_map;

int main(int args){
        /*
         * Setup the relationship among daemons
         */
```

```
        daemon_child_parent_map.insert (pair<const char*,const char*>("startd","master") );
        daemon_child_parent_map.insert (pair<const char*,const char*>("collector","master") );
        daemon_child_parent_map.insert (pair<const char*,const char*>("negotiator","master") );
        daemon_child_parent_map.insert (pair<const char*,const char*>("schedd","master") );
        daemon_child_parent_map.insert (pair<const char*,const char*>("starter","startd") );
        daemon_child_parent_map.insert (pair<const char*,const char*>("shadow","schedd") );
        /*
         * Paths to the daemon Executables
         */
        daemon_exepath_map.insert (pair<const char*,const char*>("master","/usr/local/sbin/condor_master") );
        daemon_exepath_map.insert (pair<const char*,const char*>("startd","/usr/local/sbin/condor_startd") );
        daemon_exepath_map.insert (pair<const char*,const char*>("collector","/usr/local/sbin/condor_collector") );
        daemon_exepath_map.insert (pair<const char*,const char*>("negotiator","/usr/local/sbin/condor_negotiator") );
        daemon_exepath_map.insert (pair<const char*,const char*>("schedd","/usr/local/sbin/condor_schedd") );
        daemon_exepath_map.insert (pair<const char*,const char*>("shadow","/usr/local/sbin/condor_shadow") );
        daemon_exepath_map.insert (pair<const char*,const char*>("starter","/usr/local/sbin/condor_starter") );
        /*
         * Paths to the daemon credentials, i.e. private key
         */
        daemon_credpath_map.insert (pair<const char*,const char*>("master","/etc/gridsec/hostkey_master.pem") );
        daemon_credpath_map.insert (pair<const char*,const char*>("startd","/etc/gridsec/hostkey_startd.pem") );
        daemon_credpath_map.insert (pair<const char*,const char*>("collector","/etc/gridsec/hostkey_collector.pem") );
        daemon_credpath_map.insert (pair<const char*,const char*>("negotiator","/etc/gridsec/hostkey_negotiator.pem") );
        daemon_credpath_map.insert (pair<const char*,const char*>("schedd","/etc/gridsec/hostkey_schedd.pem") );
        daemon_credpath_map.insert (pair<const char*,const char*>("shadow","/etc/gridsec/hostkey_shadow.pem") );
        daemon_credpath_map.insert (pair<const char*,const char*>("starter","/etc/gridsec/hostkey_starter.pem") );
        /*
         * PCR INDICES FOR EACH DAEMON
         */
        daemon_pcr_index_map.insert ( pair<const char*,int>("master",9) );
        daemon_pcr_index_map.insert ( pair<const char*,int>("startd",10) );
        daemon_pcr_index_map.insert ( pair<const char*,int>("schedd",11) );
        daemon_pcr_index_map.insert ( pair<const char*,int>("negotiator",12) );
        daemon_pcr_index_map.insert ( pair<const char*,int>("collector",13) );
        daemon_pcr_index_map.insert ( pair<const char*,int>("starter",14) );
        daemon_pcr_index_map.insert ( pair<const char*,int>("shadow",15) );

        TC_ENGINE tc_engine;
        if(tc_engine.initEngine()){
                tc_engine.setup_maps(&daemon_child_parent_map, &daemon_exepath_map,
                                                        &daemon_credpath_map, &daemon_pcr_index_map);
                bool ok = tc_engine.enable_tp_operation();
                if(!ok){
                        coutT<<"Enable TP operation failed";
                }
        else{
                cout<<"Could not Start Engine!";
        }
}
```

This script was tested using the private key file, where each daemon was given a copy of the private key file. This resulted in a list of encrypted files stored on the disk, (private key file size (887)/ maximum seal data size (149)) = 6 for each daemon, the original credentials deleted and the PCRs reset by forcing a restart of the machine.

6.3.2 Condor Startup

The investigation of Condor reviewed that the startup of the daemons in implemented in the DaemonCore class of the condor_daemon_core_V6 module. This can be changed to include measurement of the daemons as shown in listing 6.16. This could not be

tested due to the complexity in the Condor build process, there is not documentation that indicates how to make changes to it.

Listing 6.16: Daemon measurement before startup (daemon_core.c line 674)

```
{
        /************************************************ TCE:*******************************
         * We have the full path to the executable file here and we are about to folk
         * the process in UNIX, we can measure the executable at this point
         * However, we need to ensure that in the forkit object there is no point at which
         * the executable path can be changed, otherwise we might measure something and start
         * something else
         */
        bool engineStarted = tc_engine.initEngine();
        if(engineStarted){
                bool measurementPassed = tc_engine.measure_daemon(executable, executable_fullpath);
                if(measurementPassed){
                        // Create a "forkit" object to hold all the state that we need in the child.
                        // In some cases, the "fork" will actually be a clone() operation, which
                        // is why we have to package all the state into something we can pass to
                        // another function, rather than just doing it all inline here.
                CreateProcessForkit forkit(
                        errorpipe,
                        args,
                        job_opt_mask,
                        env,
                        inheritbuf,
                        forker_pid,
                        time_of_fork,
                        mii,
                        family_info,
                        cwd,
                        executable,
                        executable_fullpath,
                        std,
                        numInheritFds,
                        inheritFds,
                        nice_inc,
                        priv,
                        want_command_port,
                        sigmask,
                        core_hard_limit);

                newpid = forkit.fork_exec();
                }
                else{
                //log the error
                dprintf(D_ALWAYS, "TCE: Daemon Measurement Failed: %s (%d)\n",
                                                        strerror(errno), errno );
                //set the newpid to less than 0
                 newpid = -1;
                }
        }
        else{
                dprintf(D_ALWAYS, "TCE: Engine Could not be started: %s (%d)\n",
                                                                        strerror(errno), errno );
                newpid = -1;
        }
}
```

It shows that the TC_ENGINE is started (if already running simply returns true) and then each of the daemons is measured before a process for it is created.

6.3.3 Authentication

Authentication couldn't also be tested but was identified to fit as shown in listing 6.17. It makes a call to the TC_ENGINE.setup_tp_for_auth passing it the name of the daemon that is currently communicating and then sets up the credentials in the original directory. Once authentication is complete, the TC_ENGINE.cleanup_credentials is invoked to remove the un-encrypted credentials.

Listing 6.17: Credential setup before authentication on the client side (condor_daemon_client daemon.C startCommand line:556)

```
/*********************************************** TP OPERATION*****************************************/
/* he credentials need to be setup before calling the authentication method in Startcommand */
        TC_ENGINE tc_engine;
        bool setupSuccess = tc_engine.setup_tp_for_auth(_name);
        if(setupSuccess){
                start_command_result = sec_man->startCommand(cmd, sock, other_side_can_negotiate,
                                                             errstack, 0, callback_fn, misc_data,
                                                             nonblocking);

                if(callback_fn) {
                        // SecMan::startCommand() called the callback function, so we just return here
                        return start_command_result;
                }
                else {
                        // There is no callback function.

                        if(cb_errstack) {
                                // Print out the error stack.
                                bool success = start_command_result == StartCommandSucceeded;
                                cb_errstack->CallbackFn(success,sock,&cb_errstack->errstack,cb_errstack);
                        }

                        return start_command_result;
                }
        }
        else{
                dprintf(D_SECURITY,"Failed to setup credentials for Daemon %s.\n",_name);
                return StartCommandFailed;
        }
```

Listing 6.18: Credential setup before authentication on the daemon-side (daemon_core.C HandleReq line:3854)

```
/*********************************************** TP*****************************************/
/* The Startcommand calls execute, therefore the credentials need to be setup before
   calling the authentication method
 *********************************************** TP*****************************************/
TC_ENGINE tc_engine;
bool setupSuccess = tc_engine.setup_tp_for_auth(daemon);
if (setupSuccess && is_tcp && (will_authenticate == SecMan::SEC_FEAT_ACT_YES)) {
        // we are going to authenticate. this could one of two ways.
        // the "real" way or the "quick" way which is by presenting a
        // session ID. the fact that the private key matches on both
        // sides proves the authenticity. if the key does not match,
        // it will be detected as long as some crypto is used.

        // we know the ..METHODS_LIST attribute exists since it was put
        // in by us. pre 6.5.0 protocol does not put it in.
        char * auth_methods = NULL;
        the_policy->LookupString(ATTR_SEC_AUTHENTICATION_METHODS_LIST, &auth_methods);
```

```
if (!auth_methods) {
        dprintf (D_SECURITY, "DC_AUTHENTICATE: no auth methods in response ad, failing!\n");
        result = FALSE;
        goto finalize;
}

if (DebugFlags & D_FULLDEBUG) {
        dprintf (D_SECURITY, "DC_AUTHENTICATE: authenticating RIGHT NOW.\n");
}

if (!sock->authenticate(the_key, auth_methods, &errstack)) {//IMPORTANT
        free( auth_methods );
        dprintf( D_ALWAYS,
        "DC_AUTHENTICATE: authenticate failed: %s\n",
         errstack.getFullText() );
         result = FALSE;
         goto finalize;
}
else{
        tc_engine.cleanup_credentials(daemon);
}
```

6.3.4 Attestation Service

The attestation service could be used by making a call to the tp_enabled_op_attest passing it the name of the daemon and a flag indicating whether to quote the entire platform or measurements just for the given daemon. This returns a value signed with the private part of the AIK, which can be included in the classad as shown in listing 6.19. On the server side, the match-making process could be extended to check the signed value included in the classad, listing 6.20, against the values that were initially exported. The utility function getExportedValuesForMachine gets the initially exported values for the given machine while decryptValForMachine decrypts the given value using the public component (exported at key creation) of the AIK for that machine.

Listing 6.19: Extending classad with signed PCR values (daemon_core.c publish line:9025)

```
void
DaemonCore::publish(ClassAd *ad, const char* daemonName) {
        const char* tmp;

                // Every ClassAd needs the common attributes directly from the
                // config file:
        config_fill_ad(ad);

                // Include our local current time.
        ad->Assign(ATTR_MY_CURRENT_TIME, (int)time(NULL));

                // Every daemon wants ATTR_MACHINE to be the full hostname:
        ad->Assign(ATTR_MACHINE, my_full_hostname());

                // Publish our network identification attributes:
        tmp = privateNetworkName();
        if (tmp) {
                ad->Assign(ATTR_PRIVATE_NETWORK_NAME, tmp);
```

```
            tmp = privateNetworkIpAddr();
            ASSERT(tmp);
            ad->Assign(ATTR_PRIVATE_NETWORK_IP_ADDR, tmp);
        }
        tmp = publicNetworkIpAddr();
        ASSERT(tmp);
        ad->Assign(ATTR_PUBLIC_NETWORK_IP_ADDR, tmp);

        /********** TP OPERATION**********/
        /* The classad can be extended with values coming from the Quote Operation */
        TC_ENGINE tc_engine;
        bool engineInitialised = tc_engine.initEngine();
        if(engineInitialised){
            const *char signedVal = tc_engine.tp_enabled_op_attest(daemonName, false);
            if(signedVal != NULL){
                ad->Assign("SIGNED_PLATFORM_STATE", signedVal);
            }
            else{
                dprintf( D_ALWAYS, "TC_ENGINE: ERROR: Could not quote platform");
            }
        }
        else{
            dprintf( D_ALWAYS,"TC_ENGINE: ERROR: Could not start Engine");
        }
}
```

Listing 6.20: Checking platform state during match-making (condor_negotiator.MatchMaker.C matchmakingAlgorithm line:2251)

```
        // scan the offer ads
        startdAds.Open ();
        while ((candidate = startdAds.Next ())) {

            // the candidate offer and request must match
            if( !( *candidate == request ) ) {
                // they don't match; continue
                continue;
            }
            /********** TP OPERATION**********/
            /* Each candidate should be checked for the platform state specified in the
             * classad, if it does not match that exported at installation, then go to the
             * next candidate because it means that platform might have been compromised
             */
            char* signedVal;
            candidate->LookupString("SIGNED_PLATFORM_STATE", signedVal);
            char* machine;
            candidate->LookupString("ATTR_MACHINE", machine);
            string decryptedVal = decryptValForMachine(signedVal, machine);
            string exportedVals = getExportedValuesForMachine(machine);
            if( !(exportedVals.contain(decryptedVal) ) ) {
                //candidate not in right state, go to next candidate
                continue;
            }
```

Chapter 7

Evaluation, Discussion & Future Work

This chapter discusses the achievements of the proposed solution, its shortfalls and how it compares with the implementation. It also discusses the potential impact of the solution on the performance of the system and the challenges faced during the investigation process and implementation.

7.1 Intellectual Challenges

One of the main contributing factor towards the intellectual challenges faced while working on thesis was the lack of adequate documentation. This is because the initial stages of the thesis required a solid grasp of the design of the Condor software, which is not documented. The Condor project has done a good job on the administrator and user manuals, which explain how to use Condor or what its components do, but has not produced any design documentation for a developer wanting to understand how the various components fit together. This could only be done through source code review (firstly identifying which source files might be necessary and secondly reading the code and the comments in the file) and debugging (to see how the components work together). Both of which were very undesirable!

Though the thesis proves that it is possible to improve security of an existing (functional) system, the effort required in understanding their operation is very significant and require high intellectual capacity, especially if essential documentation is unavailable. This effort could have been channeled elsewhere had there been sufficient documentation.

7.2 Design Analysis

7.2.1 Sealing Protects Credentials from Malicious Code

In section 3.5, it was discussed that Condor lacks a secure way of protecting credentials, so that malicious code could have access to them. The first phase of the design greatly improves this problem in that it protects the credentials from malicious code by only exposing the credentials when the TPM determines that the platform is in the correct state. However, it does not completely eliminate the problem because the credentials are still exposed at the point of authentication. For example, malicious code running on the platform can wait for the Condor daemon to present the sealed package and once unsealed make a copy of it and use it later. Optionally, malicious code can run on the platform without affecting the values in the PCRs and present the sealed package to the TPM which will successfully unseal. The former can be improved by applying access control policies on the unsealed credentials while the latter was improved in the second phase of the design by measuring the entire software stack. This solution therefore reduces the chances of some malicious code running on a platform using the authentication credentials because it reduces their exposure time.

7.2.2 Integrity Measurements and Attestation Enable Cryptographic Identification of Hosts

By assuming a trusted operating system that records the steps taken during the boot process, so that cryptographic identification information for the platform is also included, the solution provides a means by which integrity measurements of the platform can be cryptographically linked to its identity. Each of the daemons measured would inherit this information from its parent so that any integrity measurement can be traced back to the platform on which it was taken. As a result each of the hosts can be cryptographically identified, through the measurements presented by the daemons during attestation, to the central manager and thus improving the problem described in section 3.4.

7.2.3 Integrity Measurements Guarantee Correct Behaviour of Hosts

Integrity measurements taken at installation and compared at authentication ensure that only hosts that have the correct Condor daemons interact. With the assump-

tions, ASMP-4 (platform not compromised at installation) and ASMP-2 (trusted Condor software source), made in the solution, the daemons that take part in communication can guarantee their correct behaviour because they would only be able to use the authentication credentials if that was the case. This helps to ensure that no malicious code can pretend to be a Condor daemon to either the user or the rest of the grid. Therefore, if a daemon authenticates successfully, it can be trusted to operate correctly, carrying out match-making and other policy enforcements as expected, helping improve the problem described in section 3.6 and cause no DDOS, described in section 3.7, to the user nor the grid. However, this does not prevent users from submitting malicious jobs that can carry out DDOS attacks.

7.2.4 Attestation Provides a Secure Identification of Software Required for Jobs

The attestation service, in the second design phase, allows a platform to securely report to a remote host (CM) the software it is running. In section 3.8, it was discussed that the current mechanism where the software requirements and the presence of the software was indicated using configuration files was inadequate in enforcing software requirements for a given job. The attestation service allows the CM to determine the presence of the required software by looking at the SML and use that information in the match-making process. However, this is not as easy as it sounds because the numerous configurations possible for any software are difficult to manage. Chen et al [7] also notes that measuring the binary state of software is difficult and proposes the use of property-based attestation, which uses properties of the platform rather than the cryptographic hash values of the binary code.

7.2.5 Trusted OS Enforces Measurement of Running Code

The solution heavily depend on integrity measurements. However, it is possible that a platform could be compromised without affecting stored integrity measurements. For example, it is possible that malicious code could run in the background, wait for a daemon to be measured and then replace the daemon before it gets started, as depicted in figure 7.1. This situation is helped by the assumption of a trusted operating system so that it will enforce a stronger access control policy of the component that has just been measured to ensure that the same one gets started.

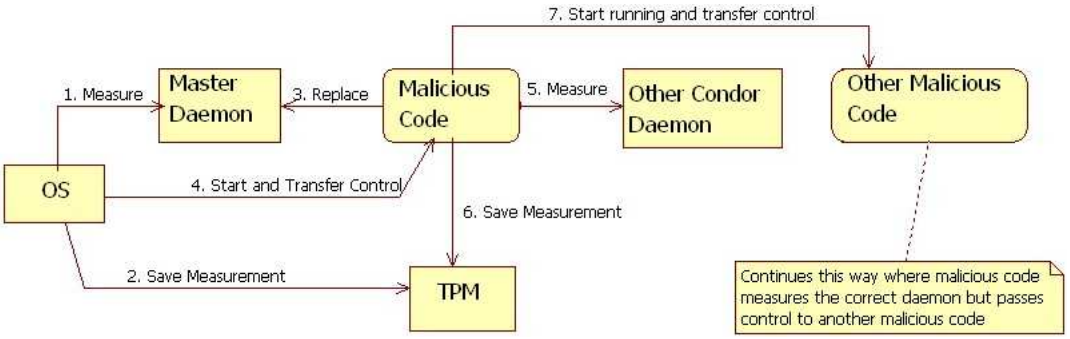


Figure 7.1: Malicious code replaces measured daemons

7.2.6 Lack of Adequate Protection Against Rogue Administrator

The solution given does not explicitly state the condition of a platform at the time it is joining the Grid. For example, a rogue administrator would install malicious code, as though it were Condor, and export the initial platform state to the CM. The CM which just compares the initial platform state to the later one would not detect this problem. The use of ASMP-3 in the solution implies that the solution would work well in cases where a platform gets compromised after its initial installation. However, it can be extended to fix the problem of a rogue administrator by keeping a database of the expected daemon measurements and comparing them with what is exported from a platform. This is possible because the cryptographic hash of an executable file would be the same regardless of the platform on which it is taken.

7.2.7 Integrity Measurements Represent Start-time Rather than Run-time State

The attestation service aims to provide a way for one system to query another system about its status at startup but does not provide a means of determining the running state of a system. This means that if a system is started and later modified, by either changing the contents of its memory address or by running some malicious code after the platform measurement, performing attestation will not indicate the modifications made to the system. This is an inherent limitation in attestation also pointed out by [5]. It can be improved by using a dynamic root of trust which can be re-evaluated at anytime rather than a static one that only gets evaluated at startup.

7.2.8 Key Management Challenges

The solution makes use of a number of keys which introduces problems of how to manage them. The sealing keys used are stored outside the TPM encrypted with a parent key that never leaves the TPM. The keys can only be used on the particular TPM on which they were created. However, the only barrier to using them is authorisation data in the form of key usage policies, specified using passwords. This implies that the solution could still be faced with a problem of managing passwords. Additionally, the entire functionality of the TPM relies on the security of owner authorisation data, which also uses passwords, and if exposed would render the whole solution less secure because the PCRs could be reset, or other authorisation data changed to make processes such as unsealing or attestation fail.

The solution also introduces the usage of AIKs in attestation, which relies on PKI. The consequence of this is twofold. Firstly it means that extra work on credential management and secondly, PKI is slower compared to symmetric encryption and therefore introduces a lug in the process. This implies that systems will have to keep track of more keys, in addition to other authentication and session keys already in use.

7.2.9 Frequency of Attestation Grows Exponentially to the Number of Daemons and the Frequency of Communication

Given that daemons within Condor send updates about their state and the platform on which they run every 5 minutes [36] to the collector, how many times will the daemons attest? The solution suggests that every time a daemon communicates, it needs to send its platform state, which means loading the AIK, getting a quote of the platform and the SML and sending to the CM. On the other end, the CM needs to check the information for each daemon even if two daemons run on the same platform.

The solution requires a modification that takes into account the fact that the platform state may not always change between consecutive communications. This requires an algorithm that can securely determine if the platform state has changed without making it an attractive attack point for potential attackers.

7.3 Implementation Challenges

7.3.1 TCG Seal Limitation

The usage of the library was tested by protecting the privake key file, with a size of 887-bytes, that is used in GSI authentication. Due to the limitation in the size of data that can be sealed by the TPM, given by (s/8 - (40-2-65)) [43], the private key file could only be sealed by first splitting it into 6 files each with size less or equal to 149-bytes. This not only brings in a problem of havnig to recombine the files after unsealing but increases the time required to completely protect the files and thereby increasing the exposure time of the private key file. An alternative would be to only seal a random portion of the private key and then recombining it at unseal. But this would mean keeping track of the point at which the split occurred. This limitation was not so much of a problem but could prove so in other environments.

7.3.2 Condor Source Code Modification

This process proved, by far, to be the most complex. It involved identifying the parts of the Condor source code that needed to be changed to make use of the library that was developed. The problem was that there were too many modules in the Condor source, all of which were undocumented and thus made it very difficult to identify where a given functionality culd be found. Even after the source files were identified, it was very difficult to identify the functions that were responsible for a given functionality because there were too many lines of code in each of the source files. This situation was even made worse when it came to integrating the library into Condor. This is because Condor uses a very complex build process that is made up of a number of "makefiles" which are also not documented and thus made the process very difficult.

7.4 Performance Analysis

One of the performance concerns with the solution is the number of keys that have to be created. The design indicates that the credentials are protected with a combination of the integrity measurements and a key created by the TPM. In the implementation, each daemon uses a different key to seal the credentials, meaning that the TPM has to create and load a key a number of times equivalent to the number of daemons measured. The good news is that this process, in this environment, would take place only during installation, which already takes sometime that is relatively large compared to

the time required for creating keys. In addition the jobs on the grid take very long, from a few hours to days, and so adding a few minutes to the authentication process cannot be considered an overhead. Therefore, in this environment, the creation of keys would not cause much of an effect on performance but could prove so in other situations.

Though the time it takes for a platform to collect information for use in attestation is very short, the freqeuncy at which this is done, dicussed in section 7.2.9, may enventaully have an impact on the performance of the system. Therefore, the implementation must carefully choose the exact points at which a platform should attest its state.

7.5 Generalisability of the Solution

This solution could be applied to any system that operates a number of components, such in service-oriented architectures. The general idea presented is that components can be measured, credentials necessary for their communication protected with TPM functions and used when the platform matches the state at the point of protecting the credentials. The components can also, using the attestation service, provide a guarantee of the correctness of the platform on which they are running. Therefore to apply this solution to another problem requires the identification of the components and deciding which of their properties can be used to determine their integrity, identifying components that can be used to restrict their communication (cryptographic credentials if they carry out authentication) and enabling the TPM. It is also important to analyse the limitations and assumptions in the solution before using it.

7.6 Future Work

7.6.1 Dynamic root of trust for measurement

The root of trust for measurement on which the proposed solution relies can be regarded as static because it is checked once, at boot up, and assumed that it never changes when the system runs. However, a lot of things can happen between when the system was started securely up and when its trustworthiness is evaluated [5]. For example, section 7.2.7 discussed the idea of a possible change in the state of a platform between start-up time and runtime. This can be solved by using a dynamic root-of-trust for measurement (DRTM), which allows the state of a platform to be

evaluated at runtime and as many times as necessary [5]. The solution can therefore be improved by using a DRTM rather than a SRTM because it can be re-evaluated at runtime and consequently force measurement of running code.

7.6.2 Trusted Virtualisation

The Condor software currently allows the use of a virtual machine to provide a separation between the running job and the machine on which it runs. This is mainly targeted at preventing user jobs, which may be malicious, from accessing the resources of the machine on which they run. Löhr et al [20] proposed a trusted grid architecture which uses trusted virtualisation to improve the security of grid computing. However, their solution still relies on a trusted grid software (Condor software), which is too complex to trust completely. Therefore, it would be worthwhile to investigate how the virtual machine usage provided within Condor can be improved to protect both the users and the resources without placing so much trust in the software or atleast be able to determine how trustworthy the software is.

7.6.3 Modifying the underlying Authentication Method to use TPM

The solution can be improved by making changes to the authentication methods so that they can use the sealed packages directly rather than having the program unseal and placing them in a directory. This is because, some malicious code might be running on the platform and wait for the credentials to be unsealed before copying them as discussed in section 7.2.5. This problem can be improved by changing the underlying authentication method, so that it communicates directly with the TPM to get the credentials and thus reducing the risk of exposing the credentials to malicious code. However, this does not completely solve the problem because the credentials may be compromised on their way to the TPM. This can further be improved by having the TPM generate the credentials so that the private key never leaves the TPM.

Chapter 8

Conclusion

The lack of a secure identification mechanism for hosts on the Grid, insecure storage of cryptographic credentials as well as the lack of a procedure for verifying that the host run the required software and not some malicious code exposes the campus Grid to a number of threats which prevents users from gaining confidence in the integrity and confidentiality of the data as well as the availability of the services on the Grid.

This thesis proposes the use of trusted computing technology to protect cryptographic credentials in such a way that it cannot be subverted by software running on the platform. It also proposes the use of remote attestation to allow one resource on the Grid to determine what software is running on a remote resource and thus evaluate whether to expect it to behave correctly. This allows users to gain confidence in the results of the jobs executed on the Grid. Attestation also provides a means by which software requirement policies can be securely enforced because the reported software stack can be checked to see if the specified software package is present on a platform.

The achievements made by this work show that it is feasible to improve the security of the campus Grid using trusted computing, despite it being enormously challenging. It also recognises some of the weaknesses in its proposals and hence not conclusive in itself but further work can be done to improve on it.

Appendix A

Abbreviations

AIK	Attestation Identity Key
API	Application Programming Interface
ASMP	Assumption
CA	Certification Authority
CBC	Cypher-Block Chaining
CM	Central Manager
CPU	Central Processing Unit
DD	Design Decision
DDOS	Distributed Denial of Service
DES	Data Encryption Standard
GSI	Grid Security Infrastructure
IP	Internet Protocol
NGS	National Grid Service
OS	Operation System
OSC	Oxford Supercomputing Centre
PCR	Platform Configuration Register
PKI	Public Key INfrastructure
RAM	Random Access Memory
RSA	Rivest, Shamir, and Adleman
RTM	Root of Trust for Measurement
RTR	Root of Trust for Reporting
RTS	Root of Trust for Storage
SML	Stored Measurement Log
SRK	Storage Rook Key
SSL	Secure Socket Layer
SSPI	Security Support Provider Interface

TCG	Trusted Computing Group
TGS	Ticket Granting Server
TP	Trusted Platform
TPM	Trusted Platform Module
TSP	TCG Service Provider
TSS	TCG Software Stack

Bibliography

[1] Atmel. Trusted platform module. www.atmel.com/products/Embedded, July 2008.

[2] Bruce Beckles. Building a secure condor pool in an open academic environment. 2005.

[3] Andrea Bottoni, Gianluca Dini, and Evangelos Kranakis. Credentials and beliefs in remote trusted platforms attestation. In *WOWMOM '06: Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks*, pages 662–667, Washington, DC, USA, 2006. IEEE Computer Society.

[4] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch. Design and deployment of a national-scale authentication infrastructure, 1999.

[5] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. V. Doorn. *A Practical Guide to Trusted Computing*. Pearson plc, 2008.

[6] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. V. Doorn. *A Practical Guide to Trusted Computing*, chapter 8, pages 104–116. Pearson plc, 2008.

[7] Liqun Chen, Rainer Landfermann, Hans Löhr, Markus Rohe, Ahmad-Reza Sadeghi, and Christian Stüble. A protocol for property-based attestation. In *STC '06: Proceedings of the first ACM workshop on Scalable trusted computing*, pages 7–16, New York, NY, USA, 2006. ACM.

[8] ClimatePrediction. Climate prediction project official website. http://www.climateprediction.net/, July 2008.

[9] A. Cooper and A. Martin. Towards a secure, tamper-proof grid platform. *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, 1:8, May 2006.

[10] Anthony J.G. Hey Fran Berman, Geoffrey Fox. *Grid Computing: Making the Global Infrastructure a Reality*, pages 42, 52. John Wiley Sons, May 2003.

[11] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, October 2003.

[12] Tal Garfinkel, Mendel Rosenblum, and Dan Boneh. Flexible os support and applications for trusted computing. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 25–25, Berkeley, CA, USA, 2003. USENIX Association.

[13] N. Nagaratnam T. Ebringer S. Munetho H. Maruyama, F. Seliger and S. Yoshihama. Trusted platform on demand (tpod). Technical report, 2004.

[14] Marty Humphrey and Mary R. Thompson. Security implications of typical grid computing usage scenarios. *MIT Press*, October 2001.

[15] IAIK/OpenTC. Trusted computing for the java(tm) platform. http://trustedjava.sourceforge.net/index.php?item=jtt/readme, July 2008.

[16] IETF. The internet engineering task force. July 2008.

[17] J. Armstrong M. Kendzierski A. Neukoetter MasanobuTakagi R Bing-Wo A Amir R Murakawa O Hernandez J Magowan N Bieberstein L. Ferreira, V. Berstis. *Introduction to Grid Computing with Globus*. IBM, December 2002.

[18] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor-a hunter of idle workstations. *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, June 1988.

[19] Miron Livny and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *In Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, pages 28–31, 1998.

[20] Hans Löhr, HariGovind V. Ramasamy, Ahmad-Reza Sadeghi, Stefan Schulz, Matthias Schunter, and Christian Stüble. Enhancing grid security using trusted virtualization. In Bin Xiao, Laurence Tianruo Yang, Jianhua Ma, Christian Mller-Schloer, and Yu Hua, editors, *ATC*, volume 4610 of *Lecture Notes in Computer Science*, pages 372–384. Springer, 2007.

[21] Barton P. Miller, Mihai Christodorescu, Robert Iverson, Tevfik Kosar, Alexander Mirgorodskii, and Florentina Popovici. Playing inside the black box: Using dynamic instrumentation to create security holes. parallel processing letters. *Letters*, 11:267–280, 2001.

[22] C. Mitchell. *Trusted Computing*. Institue of Elctrical Engineers, 2005.

[23] C. Mitchell. *Trusted Computing*, chapter 3, pages 45–47. Institue of Elctrical Engineers, 2005.

[24] NGS. National grid service official website. http://www.ngs.org.uk.

[25] National Bureau of Standards. Data encryption standard, January 1997.

[26] OSC. Oxford supercomputing centre technologies. www.osc.ox.ac.uk, July 2008.

[27] Condor Project. Condor administrator manual v7.0: Installation. http://www.cs.wisc.edu/condor/manual/v7.0/3_2Installation.html.

[28] Condor Project. Condor project official website. http://www.cs.wisc.edu/condor/.

[29] Condor Project. Condor administrator manual v7.0: Security. http://www.cs.wisc.edu/condor/manual/v7.0/security.html, July 2008.

[30] Condor Project. Condor project official website. http://www.cs.wisc.edu/condor, July 2008.

[31] Condor Project. Condor versus condor-g - what's the difference? http://www.cs.wisc.edu/condor/condorg/versusG.html, July 2008.

[32] Condor Project. Condor's classad mechanism. http://www.cs.wisc.edu/condor/manual/v7.0/4_1Condor_s_ClassAd.html, July 2008.

[33] Globus Project. Globus toolkit homepage. http://www.globus.org/toolkit/, July 2008.

[34] Globus Project. Overview of the grid security infrastructure. http://www.globus.org/security/overview.html, July 2008.

[35] Trousers Project. Trousers project. http://www.trousers.sourceforge.com, July 2008.

[36] Alain Roy and Miron Livny. Condor and preemptive resume scheduling. pages 135–144, 2004.

[37] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[38] National Semiconductor. National semiconductor home page. www.national.com.

[39] SETI@Home. Seti@home official website. http://setiathome.berkeley.edu/sah_about.php.

[40] Jennifer G. Steiner, B. Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter 1988 Technical Conference*, pages 191–202, Berkeley, CA, 1988. USENIX Association.

[41] STMicroelectronics. Stmicroelectronics home page. www.st.com.

[42] TCG. Tcg official website. http://www.trustedcomputinggroup.org.

[43] TCG. *TCG Software Stack (TSS) Specification*. TCG, March 2007.

[44] TCG. *Trusted Platform Module (TPM) Specifiation*, 2008.

[45] Infineon Technologies. Infineon technologies home page. www.infineon.com.

[46] Douglas Thain and Miron Livny. Building reliable clients and servers. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[47] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[48] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[49] Karen Miller Todd Tannenbaum, Derek Wright and Miron Livny. Condor a distributed job scheduler. in thomas sterling, editor, beowulf cluster computing with linux. *MIT Press*, October 2001.

[50] Karen Miller Todd Tannenbaum, Derek Wright and Miron Livny. Condor a distributed job scheduler. in thomas sterling, editor, beowulf cluster computing with windows. *MIT Press*, October 2001.

[51] D. C. H. Wallom and A. E. Trefetchen. Oxgrid: A campus grid for the university of oxford. In S. J Cox, editor, Proceedings of the UK e-Science All Hands Meeting 2006, National e-Science, National e-Science Centre, September 2006.