Decentralized Trust Management and Accountability in Federated Systems

Brent N. Chun
Intel Research Berkeley

Andy Bavier
Princeton University

Abstract

In this paper, we describe three key problems for trust management in federated systems and present a layered architecture for addressing them. The three problems we address include how to express and verify trust in a flexible and scalable manner, how to monitor the use of trust relationships over time, and how to manage and reevaluate trust relationships based on historical traces of past behavior. While previous work provides the basis for expressing and verifying trust, it does not address the concurrent problems of how to continuously monitor and manage trust relationships over time. These problems close the loop on trust management and are especially relevant in the context of federated systems where remote resources can be acquired across multiple administrative domains and used in potentially undesirable ways (e.g., to launch denial-of-service attacks).

1 Introduction

Emerging federated computing systems, including wide-area network testbeds [7, 22, 31] and computational Grids [13, 14], are significantly impacting the quality of networking and systems research in wide-area computing systems. These systems are characterized by multiple diverse sites contributing computational and networking resources and sharing them amongst principals spanning multiple administrative domains. In such systems, trust management is a key challenge, as trust relationships between principals can be complex and must be managed in a flexible and scalable manner. While previous work [2, 4, 5, 12, 23, 25, 28] provides the basis for expressing, delegating, and verifying trust, it does not address the concurrent problems of how to continuously monitor and manage trust relationships over time. These problems close the loop on trust management and are especially relevant in the context of federated systems where remote resources may be acquired across multiple administrative domains and used in potentially undesirable ways.

In federated systems, trust management must also be accompanied with *accountability*. Whether intentional or not, misuse of shared resources is inevitable, in particular in research settings where experimental network services and measurement studies often use the network in unusual ways. While the intent is rarely malicious, anomalous use of the network is often irresponsible and naive. For example, while a naive researcher may feel it is perfectly acceptable to map the topology of the Internet by walking the IP address space and performing `traceroute` probes to random nodes on port 80, such probing is typically viewed by an ISP as a hostile attack with the signature of a scanning worm [19, 20]. While we certainly do not want to constrain innovation, at the same time we must recognize the realities of the Internet and disallow blatant misuse of resources that is likely to cause problems. Towards this end, trust to use resources must be accompanied with full accountability to allow anomalous resource use to be rapidly identified, handled, and traced back to responsible parties.

In this paper, we present a layered architecture for addressing the end-to-end trust management and accountability problem. In this context, the three subproblems faced are: (i) expressing and verifying trust in a flexible, scalable, and accountable manner, (ii) monitoring trust relationships over time so that misuse of trust can be detected and (iii) managing and reevaluating trust relationships based on automatic detection of misuse of trust. In wide-area network testbeds, for example, these subproblems can be cast as (i) how are principals authorized to use resources in the system, (ii) how do we monitor use of these resources so that abusive behavior (e.g., scanning a remote network for valid IP addresses) can be tracked down, and (iii) how is abusive behavior automatically detected and handled before it escalates to a point where formal complaints are made (e.g., from external ISPs). Addressing these three problems will be key in sustaining long-term growth and to avoid large amounts of traffic filtering by disgruntled ISPs.

In Section 2, we motivate the end-to-end trust management and accountability problem based on experience to date on PlanetLab [7, 22], an open, wide-area network testbed. In Section 3, we describe the authentication and

authorization layer which addresses how trust is expressed, delegated, and verified and how authentication and authorization is performed. In Section 4, we describe the accountability layer which addresses how trust relationships are monitored over time to detect misuse of resources. In Section 5, we describe the anomaly detection layer which addresses how misuse of resources can be automatically detected and acted upon based on local events on individual nodes and correlation of events across nodes. In Section 6, we discuss related work and in Section 7, we conclude the paper.

2 Motivation

In this section, we motivate the end-to-end trust management and accountability problem in the context of PlanetLab [7, 22], an open, shared overlay for developing and deploying planetary-scale network services. There are a number of instances in PlanetLab where trust relationships need to be established. The most basic example of this is authorization to create a *slice*, a network of virtual machines, and to deploy a wide-area network service in it. Because users in PlanetLab span multiple administrative domains [1], it is infeasible to have a central entity authorize every possible user. In order to scale, delegation is needed. For example, one natural way to delegate in this case is to authorize principle investigators (PIs) at each site then have PIs, in turn, authorize graduate students. Graduate students might, in turn, authorize undergraduate students, and so on.

Once users have been authorized to create slices and deploy services, we then need to monitor resource usage in each slice to ensure that resources are not being misused. Table 1 lists a sample of resource misuse incidents on PlanetLab that have resulted in external complaints from ISPs, operational staff at PlanetLab sites, operational staff at universities, external sites (e.g., companies hosting web servers), external users, or some combination. In all cases so far, misuse appears to have been largely unintentional and due to either naive service design and analysis, programmer errors, or plain irresponsible use of the resources. Examples of resource misuse include network mapping experiments that probe large numbers of random IP addresses (looks like a scanning worm), services aggressively `traceroute`'ing to certain target sites on different ports (looks like a portscan), services performing distributed measurement to a target site (looks like a DDoS attack), services sending large numbers of ICMP packets (not a bandwidth problem, but renders low-end routers unusable), etc.

[1] As of May 2003, PlanetLab consists of 151 nodes at 69 sites spread across 13 countries

When anomalous resource usage occurs, we need a way to quickly determine accountability and to resolve the problem in a timely manner before it escalates to external sites generating complaints. Being able to determine accountability means being able to identify the responsible user, the user who authorized that user, and so on along a chain of trust. Resolving the problem means being able to quickly localize where the anomalous resource use is occurring and to suspend or kill the associated service. The consequences of not addressing these problems are significant and real. First, ISPs receiving network traffic they deem to be hostile could stop routing packets to and from our system. Second, sites contributing resources could pull the plug on their nodes if the operational overhead of handling external complaints is too high. Finally, if external complaints are too frequent, this could result in additional legal and administrative pressure to stop contributing resources.

Detecting anomalous resource usage in the first place is also a problem. Even if we have mechanisms to determine accountability and to resolve problems, it is still undesirable to wait until external complaints arise since such complaints still carry the risks of the consequences previously mentioned. Instead, what we would like is to proactively identify anomalous resource usage and perform appropriate actions in an automated fashion. In general, using automation to catch all types of resource misuse that might generate an external complaint is impossible. On the other hand, we do not need to eliminate all false positives and negatives to be effective. Approximate anomaly detection and response on even human timescales, at least for PlanetLab, is likely to be both possible and effective for two reasons: (i) the signatures associated with past incidents appear to be easy to detect and (ii) background noise caused by Internet worms, scanning, and probing by real attackers creates enough of a diversion that network operations cannot devote human resources to respond to all anomalous events, only the most significant and persistent ones.

3 Authentication and Authorization

The first layer in our architecture is the authentication and authorization layer. In large federated systems, it is both infeasible and undesirable to have a central entity vouch for the identities of all principals and to make associated authorization decisions regarding access control. In order for a system to scale, trust management must be decentralized to allow trust to be flexibly delegated between principals. At the same time, while delegation of trust is key for scaling, trust also needs to be fully accountable to allow delegation to be traced along paths of delegated trust. Accountable delegation is key in managing trust relationships and to trace misuse of the system back to the responsible

Date	Incident	Complaint From
Oct 2002	High UDP packet rate (on port 0), many non-existent IPs	Multiple ISPs and PlanetLab sites
Dec 2002	Running Gnutella, violates local AUP	PlanetLab site
Dec 2002	Large number of packets to non-existent IPs	Multiple ISPs
Dec 2002	Large number of non-standard ICMP packets	Campus net admins
Dec 2002	High UDP packet rate and bandwidth usage	Multiple PlanetLab sites
Dec 2002	UDP packet traffic spikes	Campus net admins
Dec 2002	High ICMP packet rate	Multiple ISPs and PlanetLab sites
Dec 2002	Pinging many external WWW servers	PlanetLab site
Feb 2003	High UDP packet rate	PlanetLab site
Mar 2003	High ICMP packet rate	Campus net admins
Apr 2003	Mapping Internet on port 80, many non-existent IPs	Multiple PlanetLab sites and campuses
Apr 2003	DDoS on multiple WWW servers (measurement exp)	Multiple companies
Apr 2003	High UDP packet rate	PlanetLab site
May 2003	DoS on campus web server	PlanetLab site
May 2003	Portscan on external network	External site
May 2003	Periodic (20 min) portscans	Multiple external sites
May 2003	Port scan	External user

Table 1: A sample of resource misuse on PlanetLab during the period starting October 2002 and ending May 2003. Each row in the table corresponds to a service generating external network traffic which resulted in complaints from ISPs, operational staff at PlanetLab sites, operational staff at universities, external sites (e.g., companies hosting web servers), external users, or some combination. In all cases, incidents were resolved through a combination of email and phone conversations (usually over a period of days).

principals, the principals who vouched for their identities, and the principals who authorized them to use the system. The authentication and authorization layer provides the mechanisms to express, delegate, and verify trust relationships and to perform authentication and authorization using public key cryptography based on fully accountable paths of trust.

3.1 Delegation of Trust

Previous work in distributed security infrastructures has addressed the trust delegation problem. Distributed capability systems, used in distributed operating systems such as Amoeba [28] and Chorus [25], use unforgeable tokens to represent access control rights to perform privileged actions. Such tokens are typically constructed using large random numbers which are hard for an adversary to guess. A capability granting access to an object includes a large random number, an identifier for the object, and a set of actions that can be performed on the object. The principal who issues a capability on an object grants access on that object to any requesting principal who presents the associated capability. Using capabilities, delegation of trust is natural. A principal P_1 delegates trust to P_2 simply by sending P_2 the associated capability. P_2, in turn, can further delegate trust to a principal P_3 in the same manner, and so on. A key advantage of this approach is that no additional mechanisms are needed to delegate trust between principals.

In large federated systems, there are three key problems with distributed capabilities. First, capabilities are not fully accountable. If trust is delegated across a path of principals $P_1 \rightarrow P_2 \rightarrow \cdots \rightarrow P_{n-1} \rightarrow P_n$, there is no easy way to trace the delegation across the path of trust. Should P_n use the delegated capability to abuse the system in some manner, one thing we would like to know is that P_{n-1} was the principal who authorized P_n. Knowing who authorized the responsible party is important since we envision that chains of trust will frequently mirror the natural hierarchy of privilege associated with organizations. For example, in PlanetLab, a common chain of trust might be $P_{pi} \rightarrow P_{grad} \rightarrow P_{ugrad}$, where P_{pi} is a principle investigator associated with a contributing site and P_{grad} and P_{ugrad} are graduate and undergraduate students, respectively, at that site working on a related project. Accountable paths of trust are a key element for implementing full accountability (Section 4).

Second, distributed capability systems presume the existence of secure, authenticated channels on which capabilities can be granted and transferred without fear of interception by malicious third parties. While such an assumption might hold in distributed systems in certain set-

tings (e.g., with specialized trusted hardware), it does not hold in a large federated system where the set of principals is large, dynamic, and crosses multiple administrative domains. Without authentication, should P want to grant a new capability to Q, P cannot know whether the principal it is talking to is indeed Q or not. If the principal turns out not to be Q, the capability might be used in some undesirable and abusive manner (e.g., scanning remote hosts and launching scripted attacks). The same authentication problem occurs when capabilities are delegated by transferring them between principals; the principal delegating the capability must ensure it is talking to the intended principal.

Third, distributed capability systems make certain types of trust relationships awkward, if not impossible, to express. For example, a set of principals $P_1, P_2, \ldots, P_n$ might trust a specific principal Q to make certain access control decisions on their behalf by issuing capabilities (e.g., a set of nodes trusting some entity to make resource management decisions on their behalf). However, since the capabilities are issued by Q, if a principal R subsequently obtains and presents a capability issued by Q to P_i, P_i has no knowledge of the capability, since the capability was issued locally by Q, not P_i. For P_i to accept the capability, P_i would have to know about the capability either a priori or be able to dynamically determine that the capability is valid. Another example of where expressing trust is awkward is partial delegation of trust. For example, suppose P issues a capability to Q to perform actions a_1 and a_2 at P. If Q subsequently wants to delegate to R the ability to only perform a_1, expressing and validating this type of partial delegation using distributed capabilities in a general purpose manner with an arbitrary number of actions can be difficult.

Trust management systems such as PolicyMaker [5] and KeyNote [4] also provide mechanisms for delegation of trust. In contrast to distributed capability systems, though, delegation is fully accountable along paths of trust. In trust management systems, principals are represented as public keys. Trust is delegated between principals through signed statements of the form P asserts Q may perform some action a, where a is a predicate expressed in a trust management language. Authorization to perform an action is then done using a compliance checker which, in essence, searches for a path of signed statements starting with a trusted principal and ending with the requesting principal, where each statement has a predicate that evaluates to true, has not expired, and has a valid signature. Since actions are represented simply as predicates in a trust management language, trust management systems are capable of expressing complex trust relationships in a far simpler manner than would be possible using a capability system. On the other hand, like distributed capability systems, they also presume the existence of secure, authenticated channels (e.g., as established using a preexisting public key infrastructure).

3.2 Implementation

For full accountability, decisions on both authentication and authorization should be traceable along paths of delegated trust whenever a privileged action is performed. In our current implementation, we support full accountability for both authentication and authorization. For authentication, we rely on SSL authentication based on chains of X.509 certificates. As pointed out in [5], we can view X.509 certificates as simply being a special case of trust management, where the action being delegated is the assertion of identity. For example, whenever a certificate authority such as Verisign issues an X.509 certificate to some principal, it is effectively asserting that principal's identity for some period of time. Given a chain of X.509 certificates and a request to perform some privileged action, we can trace the assertion of identity from some trusted entity (e.g., a trusted certificate authority) across a chain of principals to determine the path by which the requesting principal's identity was asserted.

For authorization, we use trust management certificates to assert a delegation of trust from one principal to another for some period of time. Trust management certificates are represented as signed XML statements. Each statement includes the issuing principal's public key, the subject's public key, a valid time interval, the action being delegated, an application-specific language in which the action is expressed, and the issuer's signature. In contrast to a number of previous systems which prescribe a specific language to express actions, our implementation treats actions as opaque strings. This allows actions to be expressed in a language most natural to the application. Given a chain of trust management certificates delegating trust across a chain of principals, we use a simple compliance checker to verify the validity of the chain. The compliance checker does this by first verifying the validity of signatures on each hop of the chain and second by checking that each hop in the chain is delegating a subset of the trust in the previous hop both in time and in "space" using an application-supplied callback function. As with chains of X.509 certificates for authentication, a chain of trust management certificates allows for full traceability along a path of delegated trust for authorization.

4 Accountability

The second layer in our architecture is the accountability layer. The accountability layer provides two key services. First, it provides continuous monitoring on how trust relationships are being used by the principals in the system.

Second, it provides periodic logging of monitoring data to create historical traces of how principals behave over time. Together, monitoring and logging allow trust relationships to be continuously scrutinized, such that should a misuse of trust occur, it is a straightforward exercise to search the log data and identify the responsible principal along with (i) the chain of trust that asserted that principal's identity and (ii) the chain of trust that authorized the principal to perform the trusted action. Appropriate actions can then be performed either in-band or out-of-band (e.g., the user could be reprimanded or have his/her privileges revoked) along the appropriate paths of trust.

4.1 Resource Usage

The key class of trust relationships we are concerned with in federated systems are ones that either directly or indirectly authorize resource usage on remote machines and networks. In the PlanetLab network testbed, for example, users have the ability to dynamically create a *slice*, a network of virtual machines, and to deploy a wide-area network service in it. Such services are capable of consuming both local resources as well as remote resources over the network. While virtual machines [1, 30], sandboxing [17, 27] and resource management mechanisms [11, 15, 26, 29] can ensure secure, bounded consumption of local resources, network services are still capable of affecting remote systems over the network. For example, there is nothing to prevent a wide-area service from using its multiple vantage points of the network to launch a distributed denial-of-service attack. In the event that such abuse occurs (either intentionally or not), we need to be able to identify the responsible principal and who authorized that principal to use the system.

4.2 Monitoring

Monitoring for resource usage accountability can be implemented using a combination of existing OS accounting and monitoring mechanisms, additional fine-grain monitoring primitives, and indirection. Existing OS accounting and monitoring mechanisms already provide methods to associate CPU, memory, network, and disk usage with a specific process and local principal (e.g., a login in Linux). In Linux, for example, a significant amount of information about a process can be gleaned simply by inspecting the `/proc` filesystem or by using programs which interpret the contents of `/proc`. For example, the `fuser` command maps use of files, TCP ports, and UDP ports to a local principal. Using existing OS accounting and monitoring mechanisms is desirable since such mechanisms are already well-supported. On the other hand, such mechanisms alone are insufficient to provide monitoring at the level of detail required to provide full accountability.

To provide full accountability, fine-grain monitoring primitives are needed that allow resource usage in the system to be monitored at various levels of aggregation. As mentioned, existing resource management mechanisms such as proportional-share schedulers can ensure bounded consumption of local resources on each node. On the other hand, short of severely constraining the types of applications that can be supported, applications and network services generally have free use of the network in terms of where data can be sent and the type of data that can be sent. Existing network monitoring primitives on modern operating systems do not provide monitoring information at the necessary granularity. For example, in Linux, network statistics exposed through the `/proc` filesystem are aggregate statistics for the entire node and do not allow network statistics for a specific local principal to be collected. Furthermore, even aggregate network statistics in Linux do not capture network traffic statistics at the level of detail needed for full accountability (e.g., per TCP port, per destination IP address, etc.).

Fine-grain monitoring primitives on network usage can be provided by logging additional information in the kernel and exposing this information at user-level (e.g., using the `/proc` filesystem, a pseudo device and appropriate `ioctl` commands, etc.). On each node in the system, we would like to be able to observe how each local principal is using the network at a fine level of detail. For systems such as wide-area network testbeds, the overhead of such monitoring is both warranted and likely to be negligible anyway given the speed of modern processors and network interfaces in comparison to end-to-end wide-area network bandwidth. For each local principal, we would like to collect statistics such as TCP ports used, UDP ports used, ICMP packets sent, and, importantly, destination IP addresses (along with packet and byte counts). In PlanetLab, we have already begun the implementation and deployment of early versions of mechanisms that collect these types of statistics and expose them to user-level programs.

With resource usage statistics being collected on local principals, the last piece needed is the ability to map local principals back to global principals. Examples of local principal names include `uid`'s, Linux vserver security contexts, and virtual machine IDs. Examples of global principal names include global usernames or, more practically, public keys (and associated identifying information through an identity chain of trust) as described in Section 3. Because federated systems span multiple administrative domains and are composed of nodes, each with their own namespace of local principals, a natural way for a global principal to make use of remote resources is for the global principal to obtain a set of distributed resources

and execute as local principals while maintaining global to local principal mappings. In Grids based on the Globus toolkit [13], for example, a file (`/etc/grid_mapfile`) maps global names from X.509 certificates to local Unix logins. In providing full accountability, a federated system must maintain enough information to map from resource usage to a local principal to a public key to identifying user information associated with that public key.

4.3 Logging

Logging creates historical traces of how principals behave over time and can be implemented using existing logging and database technology. Given appropriate resource monitoring primitives, resource usage information on each node can be periodically collected and mapped back to associated principals along chains of trust. Such information might then be stored in standard logfiles using existing mechanisms such as `syslog`. Alternatively, resource usage information might be stored and indexed in local relational databases to allow complex queries on logfile data to be performed in a timely manner.

In a federated system, resource usage information needs to be collected and logged in persistent storage for at least two reasons. First, because misuse of resources can require correlating events over time and aggregating resource usage behavior (e.g., probing of a large number of IP addresses over a few hours), the system needs to maintain information over a window of time as opposed to just triggering on observation of specific events. Second, because nodes can crash and we do not know in advance how much information we may need to perform postmortem analysis in the event that resource misuse is detected, logging extensive monitoring information to persistent storage is a prudent decision.

4.4 Implementation

Our implementation currently consists of resource monitoring and logging of CPU, memory, network, and disk usage statistics for all active slices on all nodes of PlanetLab. Resource monitoring is continuous and is reported through a variety of different services, a number of which are maintained by other users and offered as shared services. Two examples of resource monitoring include the `/proc/scout` interface, which provides fine-grain network monitoring statistics, and the `slicestat` sensor server interface, which provides current per-slice resource usage statistics. Both examples rely on Linux kernel extensions to export relevant resource monitoring information. `/proc/scout`'s monitoring interface is implemented as part of SILK, a loadable kernel module which provides resource isolation and scheduling mechanisms for slices. Network monitoring in SILK includes extensive per-slice network information such as total bytes/packets sent and received for TCP, UDP, and ICMP traffic for each slice and the set of destination IP addresses each slice has sent to. The `slicestat` sensor server also relies on kernel extensions. It uses information exported by Vservers [27] (a kernel patch) in combination with existing per-process information contained in `/proc` in order to provide current per-slice resource usage information for CPU, physical and virtual memory, and send/receive network bandwidth over the last 1, 5, and 15 minutes.

5 Anomaly Detection

The third layer in our architecture is the anomaly detection layer. The goal of this layer is to perform automatic detection of anomalous resource usage and to perform appropriate actions in response. As previously mentioned, while isolation and resource management mechanisms can bound local resource consumption, network applications and services can still affect remote systems over the network in significant ways, even with low to moderate bit rates. In response to such misuse of resources, we need a way to automatically identify anomalous resource usage and to resolve problems in a timely manner before they escalate to a point where external sites issue formal complaints (e.g., by automatically suspending the associated application/service, contacting the relevant principals, etc.). Implementing automated anomaly detection requires a combination of both local anomaly detection on each node and distributed anomaly detection that correlates events across nodes. In this section, we describe the key issues and sketch a potential path towards a solution.

5.1 Local Anomaly Detection

Local anomaly detection allows the system to flag resource usage on individual nodes that is deemed suspicious or likely to cause external problems and to perform appropriate actions. We use the monitoring and logging primitives from the accountability layer to observe resource usage information for each principal on each node. We then apply rules that map resource usage information over time to anomalous events along with a warning level. For each type of anomalous event and each warning level, there is a set of associated actions that are performed. For example, if the anomaly detection layer detects that an application is performing systematic portscans on a target network, the actions associated with this detection could be filtering of all external network traffic for the application and emailing operational support staff about the application, the anomaly, the responsible principal and associated chains of trust.

Local anomaly detection focuses mainly on how applications and services use the network, since use of local computational resources can be bound and controlled through use of appropriate VMM and resource management techniques. In essence, network anomaly detection in a federated system can be viewed as being the converse of the intrusion detection problem. In traditional network intrusion detection systems (NIDS) [3, 18, 21, 24], incoming network traffic is scrutinized and rules are applied which generate warnings on potentially hostile traffic from the outside world. In contrast, in a federated system, what we want is to scrutinize outgoing network traffic and to generate warnings (and take actions, such as suspending the offending application of service) on potentially hostile traffic *to* the outside world. In summary, what is needed is a *reverse anomaly detection* system.

In the local node case, reverse anomaly detection can be implemented by using monitoring and logging information from the accountability layer and applying appropriate rules and actions. One natural starting point towards such an implementation is to start with an existing, open NIDS (e.g., `snort` [24], Bro [21], etc.) and simply run the NIDS in reverse. That is, view all traffic being generated by the node as the potentially hostile traffic and view all external destinations as being potential targets. Running an existing NIDS in reverse has the benefit of leveraging existing rules for various types of anomaly detection. On the other hand, this only allows anomaly detection at a a single level of aggregation, the entire node. The key challenge for implementing effective local anomaly detection will be figuring out how to perform anomaly detection at multiple levels of aggregation (e.g., per-principal, per-process, etc.) and tie that information back to responsible principals along chains of trust in an efficient manner.

5.2 Distributed Anomaly Detection

Distributed anomaly detection identifies anomalous network behavior in the aggregate by correlating network events across multiple nodes, performing distributed aggregation, and applying rules in either an incremental or aggregate manner. In wide-area network testbeds, for example, network services (e.g., content distributed networks, global storage systems, etc.) run on large numbers of geographically distributed nodes. In such settings, while a service's behavior on a single node may be acceptable, its behavior in the aggregate may not be. For example, a service running on 1000 nodes in the wide-area could portscan a target machine by having a process running on each node probe the target on just a small number of ports, thereby avoiding local detection. To detect these types of anomalies, network events need to be correlated across nodes and aggregation needs to be performed at multiple levels (e.g., per-application, per-slice, entire system, etc.) while allowing responsible principals to be held fully accountable through paths of trust.

The key challenge in implementing distributed anomaly detection is keeping overhead low while detecting anomalies in a timely manner. Given the monitoring and logging from the accountability layer, for example, one brute force approach for detection is to simply replicate all outgoing network traffic from all nodes to a central site where a traditional NIDS is running in reverse. Each replicated packet would also be tagged with additional metadata (e.g., responsible principal, application, process IDs, etc.) to allow network anomalies to be handled and traced back to a responsible principal along a path of trust. The brute force approach, while simple, is untenable in large systems due to overhead and scalability problems. To make distributed anomaly detection practical, nodes will have to use substantially less bandwidth compared to the brute-force approach.

One possible approach to solving the distributed anomaly detection problem in a scalable manner is to leverage location-independent, key-based routing (KBR) services [10]. If all data is processed at a single site, centralized bandwidth and computation is required which clearly presents a bottleneck. On the other hand, if processing is distributed, then each node in the system collects and processes a fraction of the total data, thereby improving scalability. Using KBR with PlanetLab, for example, we might perform per-slice anomaly detection by hashing slice names to keys, routing anomaly detection data to nodes responsible for associated keys, and applying anomaly detection rules. Of course, if the data being routed to the nodes is raw packets, then aggregate bandwidth usage in this scheme is no better than the centralized, brute-force approach. Thus, a key challenge in implementing effective distributed anomaly detection will be figuring out how to perform appropriate summarization when aggregating data while still detecting significant anomalies.

6 Related Work

Previous work in trust management systems [4, 5], distributed capability systems [25, 28], and distributed security infrastructures [2, 12, 23] provides the basis for expressing, delegating, and verifying trust in federated systems. These systems enable scalability by allowing trust relationships to be expressed and verified in a completely decentralized fashion through delegation. Our work builds on these efforts by addressing the concurrent problems of how to continuously monitor and manage trust relationships over time, once trust relationships have already been established. These problems close the loop on trust management and are especially relevant in the context

of federated systems where remote resources can be acquired across multiple administrative domains and potentially used in undesirable ways (e.g., to launch denial-of-service attacks).

Network intrusion detection systems (NIDS) such as `snort` [24] and Bro [21] use traffic analysis to detect and respond to anomalous network traffic. The key problem these systems address is how to identify anomalous incoming traffic from the outside world and to perform appropriate actions. In contrast, federated systems are primarily concerned with the converse problem, namely how to identify anomalous outgoing traffic *to* the outside world and to perform appropriate actions. Running a NIDS in reverse on a single node provides a starting point towards addressing this problem. Such an arrangement, however, only provides anomaly detection at a single level of aggregation, the entire node. To provide full accountability, federated systems require aggregation at multiple levels both on individual nodes and correlated across nodes for distributed anomaly detection. Addressing this problem in an efficient and effective manner is still an open research problem.

Previous work on limiting the capabilities of compromised machines also relies on reverse anomaly detection. There, anomaly detection is applied on outgoing network traffic and hostile packets are either dropped or throttled to limit the potential damage an attacker (or worm) might cause using a compromised machine. AngeL [6] is a Linux kernel module that implements this idea by intercepting and dropping network packets which match well-known signatures of over 70 attacks. Williamson's [32] work is similarly motivated but focuses exclusively on self-replicating computer worms and, importantly, does not require a priori attack signatures. Here, the assumption is that normal network traffic does not involve making large numbers of connections to distinct machines (as a scanning worm does). Thus, by limiting the rate of connections to new hosts, worm propagation can be significantly slowed. Our work also utilizes reverse anomaly detection but for a different purpose, namely to protect the Internet against legitimate distributed applications gone awry.

Work on building accountability into electronic commerce protocols also bears some similarity to our work. There, the key challenge is the design of protocols which have provable properties with respect to accountability. For example, a user purchasing service from a service provider ought to be capable of proving to a third party that the provider agreed to provide service if the provider reneges on a transaction. Representative work in this area includes frameworks [9, 16] for analyzing accountability in communication protocols and the design of protocols [8] for delegation of trust with full accountability for electronic commerce applications. Compared to our work, these efforts are complementary in that our work is focused mainly on determining accountability as opposed to *proving* accountability through cryptographic mechanisms. How to implement and integrate efficient mechanisms for proving accountability for fine-grain actions in federated systems remains an interesting research question.

7 Conclusion

Emerging federated systems are significantly impacting the quality of networking and systems research in wide-area computing systems. In order to maintain this momentum, we argue that better facilities for decentralized trust management and accountability are needed. Such facilities should allow us to express and verify trust in a flexible and scalable manner, monitor the use of trust relationships over time, and manage and reevaluate trust relationships based on historical traces of past behavior in a fully accountable manner. Towards this end, we are currently in the process of implementing and deploying infrastructure to address the first two problems in the context of the PlanetLab network testbed. This infrastructure includes authentication and authorization based on a new decentralized trust management system along with fine-grain monitoring primitives and extensive logging of resource usage. We are also investigating initial steps towards rudimentary reverse anomaly detection to address the third problem.

References

[1] P. R. Barham, B. Dragovic, K. A. Fraser, S. M. Hand, T. L. Harris, A. C. Ho, E. Kotsovinos, A. V. S. Madhavapeddy, R. Neugebauer, I. A. Pratt, and A. K. Warfield. Xen 2002. Technical Report UCAM-CL-TR-553, University of Cambridge, Computer Laboratory, January 2003.

[2] E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. The crisis wide area security architecture. In *Proceedings of the 1998 USENIX Security Symposium*, January 1998.

[3] BlackIce. Blackice defender (`http://blackice.iss.net`).

[4] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The role of trust management in distributed systems security. In *Proceedings of Secure Internet Programming*, 1999.

[5] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Conference on Privacy and Security*, May 1996.

[6] D. Bruschi and E. Rosti. Angel: A tool to disarm computer systems. In *Proceedings of the 2001 New Security Paradigms Workshop*, pages 63–69, September 2001.

[7] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: An overlay

testbed for broad-coverage services, January 2003. PlanetLab PDN-03-009.

[8] B. Crispo. Delegation protocols for electronic commerce. In *Proceedings of the 6th IEEE Symposium on Computers and Communications*, July 2001.

[9] B. Crispo and G. Ruffo. Reasoning about accountability within delegation. In *Proceedings of the 3rd International Conference on Information and Communications Security*, November 2001.

[10] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[11] A. Demers, S. Keshav, and S. Shenker. Anaylsis and simulation of a fair queueing algorithm. In *Proceedings of the 35th IEEE Computer Society International Conference (COMPCON)*, pages 380–386, March 1990.

[12] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. Spki certificate theory, September 1999. RFC 2693.

[13] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[14] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, June 2002. Open Grid Service Infrastructure WG.

[15] S. Hand. Self-paging in the nemesis operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, February 1999.

[16] R. Kailar. Reasoning about accountability in protocols for electronic commerce. In *Proceedings of the 14th IEEE Symposium on Security and Privacy*, pages 236–250, 1995.

[17] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, Maastricht, The Netherlands, May 2000.

[18] Z. Labs. Zone alarm (`http://www.zonelabs.com`).

[19] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the sapphire/slammer worm, February 2003.

[20] D. Moore, C. Shannon, and J. Brown. Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of the SIGCOMM Internet Measurement Workshop 2002*, August 2002.

[21] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 1998 USENIX Security Symposium*, January 1998.

[22] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of HotNets-I*, October 2002.

[23] R. L. Rivest and B. Lampson. Sdsi – a simple distributed security infrastructure. Available from: `http://theory.lcs.mit.edu/~rivest/sdsi10.html`, 1996.

[24] M. Roesch. Snort – lightweight intrusion detection for networks. In *Proceedings of the 13th Systems Administration Conference*, November 1999.

[25] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the chorus distributed operating systems. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, April 1992.

[26] P. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of the 1998 ACM SIGMETRICS Conference*, pages 44–55, June 1998.

[27] Solucorp. Whitepaper: Virtual private servers and security contexts. Available from: `http://www.solucorp.qc.ca/miscprj/s_context.hc`, September 2002.

[28] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 558–563, 1986.

[29] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 1–11, 1994.

[30] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, December 2002.

[31] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, December 2002.

[32] M. M. Williamson. Throttling viruses: Restricting propagation to defeat malicious code. In *Proceedings of the 2002 Annual Computer Security Applications Conference*, December 2002.