# Dynamic Mandatory Access Control for Multiple Stakeholders

Vikhyath Rao
Systems and Internet Infrastructure Security Laboratory
Pennsylvania State University
University Park, PA 16802
vrao@cse.psu.edu

Trent Jaeger
Systems and Internet Infrastructure Security Laboratory
Pennsylvania State University
University Park, PA 16802
tjaeger@cse.psu.edu

## ABSTRACT

In this paper, we present a mandatory access control system that uses input from multiple stakeholders to compose policies based on runtime information. In the emerging ubiquitous environment, many devices run software whose access permissions depends on multiple stakeholders, such as the device owner, the service provider, the application owner, etc., rather than a single system administrator. However, current access control administration remains as either discretionary, allowing the running and perhaps compromised process to administer, or mandatory, requiring all permissions to be known by load-time. A key problem is that users may download arbitrary programs to their devices, requiring that the system contain such programs while allowing some reasonable functionality. However, such programs may need access to resources that can lead to attacks, such as implementing voice-over-IP calls, but that may also be needed for benign operations. In our approach, we use a "soft" sandboxing mechanism to first contain such processes, request the stakeholder to authorize operations outside the sandbox that are not prohibited by policy, and maintain a runtime *execution role* for the process to identify its access state to the stakeholders. We define a *proxy policy server* that caches and combines stakeholder policies to make such access decisions. Our framework was implemented by modifying the SELinux module and using a remote proxy policy server, although a local proxy policy server is also possible. We incur a 0.288 $\mu$s performance overhead only when stakeholders need to be consulted, and new permissions are cached.

## Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access Control*

## General Terms

Security

## Keywords

Mobile Phones, Distributed Access Control, SELinux

## 1. INTRODUCTION

The emergence of a wide range of applications for ubiquitous devices, such as cell phones, is causing such devices to evolve toward general-purpose computing systems. Unlike PCs, however, such ubiquitous devices have a history of being closed systems that implement the needs of multiple stakeholders, such as the device manufacturers and telecommunications service providers for cell phones. The introduction of downloadable applications to such an environment opens these formerly closed systems, but these stakeholders wish to retain some control over these systems, unlike PCs. Thus, when an application is downloaded, it is imperative that the application fall under the administration of these stakeholders. However, for previously unknown applications or for untrusted applications that may leverage some sensitive permissions, the stakeholders need a way to make runtime administrative decisions to control such software, while providing the desired user experience. In this paper, we propose a dynamic administrative mechanism that provides scalable and stateful administrative decisions for multiple stakeholders.

In general, there are primarily two approaches for administering access control policies, discretionary access control (DAC) and mandatory access control (MAC). DAC permits administration by the owners of system objects, often users. The central problem with DAC administration is that users or processes on behalf of users, accidentally or intentionally, may override important system permissions, thereby compromising security. In MAC, administration is restricted to trusted subjects, such as system administrators, which can ensure that a specific goal is enforced on the system.

While MAC administration appears to provide the control necessary to ensure system security, it is too restrictive for the dynamic environment of these ubiquitous devices. For MAC administration, it is imperative that system administrators define accurate security policies for all possible application deployments and executions, but that is difficult for the following reasons. First, each deployment may have a different set of stakeholders with different security requirements that must be composed. Even the set of stakeholders may not be known until runtime. Second, some applications may want to use some sensitive permissions. While it is easy to deny permissions that lead directly to attack, in many cases it is a combination of mutually conflicting permissions (e.g., access to voice input and the WiFi network that would enable VoIP, circumventing telephony charges) that may lead to attack, rather than a single permission. Third, some applications may be previously unknown to the stakeholders, so it is necessary to derive permissions on the fly. However, strict sandboxing may be too restrictive, so the mechanism needs to be able to authorize limited permissions, again preventing attack vectors. A traditional MAC policy is inadequate in this case, because it does not capture the requirements of all stakeholders nor express the dynamic requirements for determining policy.

Existing approaches do not balance the security with the dynamic requirements of such systems accounting for governance by multiple stakeholders. In phone systems, Google Android uses static manifests to load the policy for new applications at install time [15], and Symbian uses certificate-based permissions [11, 12]. These approaches assume that the policy is sufficiently static to be pushed to the client at install timeand importantly, that the program is already known to a particular stakeholder responsible for the system security, and finally, only static policy needs to be enforced by a Client Reference Monitor (CRM). Traditional MAC systems presume either that all programs are known ahead of time (e.g., SELinux [10]) or that the mapping between users and labels is determined at login (e.g., MLS [1] and RBAC [3]). None of these systems support the download of programs that may need sensitive permissions whose use may preclude other permissions (e.g., Chinese Wall [2]) nor do these approaches support the input of multiple stakeholders.

Our approach implements dynamic administrative decisions in a distributed environment consisting of multiple stakeholders. For each application process, we provide a base policy that only provides access to those operations granted for all runs. For previously unknown applications, this policy provides a sandbox. When permission is requested that is not authorized by the base policy, but not strictly prohibited, a policy server is consulted to determine whether this operation should be authorized based on the input of the application's stakeholders. The policy server implements a hierarchical mechanism to determine the application's stakeholders and their administrative decisions regarding this request. We say that the mechanism is hierarchical because the stakeholder's decisions may be cached at the policy server on the device, a proxy policy server (e.g., in the telecommunications network), or on the stakeholders themselves. In best case, the stakeholders are already known and the administrative policy is already cached on the device, so it is a matter of updating the MAC policy policy. A variety of policies are permitted for combining stakeholder decisions. At present, we only consider a Chinese Wall-style selection of one set of permissions from a conflict set, although others are possible. We support both transient updates (e.g., limited use or temporary permissions) and persistent updates (e.g., policy modules in SELinux).

Our major contribution in this paper is an administrative policy mechanism for MAC in dynamic environments with support for multiple stakeholders. This mechanism provides the functionality described above, as well as supporting revocation, since we believe in any dynamic environment, policy may also be removed. A key insight in implementing such a mechanism is that it will not be possible to push the state of each devices MAC policy around the network (e.g., to the proxy or stakeholders), so we identify that subjects correspond to dynamic "roles" that, like typical RBAC, imply that the subject possesses a set of permissions, but unlike RBAC, is used by policy servers and stakeholder to make administrative decisions. We have implemented our administrative mechanism and find that the performance overhead for each access request is minimal at 0.288 $\mu$ s. Of course, there may be additional overhead to talk to the proxy server and stakeholders. We expect that such delays will be similar to registering the device on the network, and much administrative policy may be captured in that task.

The rest of the paper is structured as follows. In section 2 we define the problem and challenges our framework is trying to solve, followed by an elaborate explanation of our solution in section 3. We then discuss the implementation details of our framework in section 4. Section 5 deals with the performance and overhead involved in our solution along with some additional features. Discussion of our framework, and other existing approaches with regards to the enveloping skeleton of Usage CONtrol (UCON) is presented in related work in section 6. We conclude the paper with section 7 which also summarizes our future work.

## 2. PROBLEM

In order to help define our problem clearly we present an example scenario. We consider the telecommunications system whose core network is owned by the network operator, and provides services to millions of distributed clients. Early telecommunications systems did not allow users to install any software obtained without the service providers permission. Even basic features like ring tones could only be downloaded Over-The-Air (OTA). This is unlike personal computers, where the lines are demarcated more in favor of increased user privileges to the detriment of security. In recent years, however, the telecommunications network has opened up, as consumers demand more services, and advanced cellphones with support for installation of third party applications were introduced. Following an Internet-like model where "anything goes" is not feasible in a closed network like the telecommunication network. Many researchers already warn us about the effects of end-users installing third party applications [13, 4].

Further, the more restrictive the service providers are, the more alienated the users will become, leading to a situation where the users rebel against the very same providers who are trying to protect them and their systems. For example, the initial iPhones were introduced which locked users into a certain service provider. This quickly resulted in mass "Jail-breaking" of these phones [9, 7]. "Jail-breaking" is a term for removing locks that the service providers implement. The users who installed these hacks, many from questionable sources, further exposed themselves and their systems to more threats. This creates a security model where the users are adversaries. MAC is a good solution as can implement verifiable security, but the demand for increased services and flexibility from consumers means a static MAC policy is not flexible enough.

Thus we define our problem statement as follows: *Enabling dynamic policy administration in MAC systems by multiple stakeholders in a distributed environment.* We need to perform the following steps in any effective framework.

- Identify when to ask - Here we define the conditions under which we ask for new permissions. For this, we use prior work in *access control spaces* [5] as guidance. The entire request space can be divided into *prohibited*, *permissible*, *specified* and *unknown* subspaces. The base policy on the client defined by the manufacturer corresponds to prohibited and permissible subspaces. Core applications that have already been granted permissions to run, including the default phone dialing, SMS, and phonebook applications come under permissible subspace. Critical resources like SIM card secrets are permanently denied access are part of the prohibited subspace. Unknown subspace is the remaining set of permissions that have not been defined locally. When an access request for a permission residing in the unknown subspace is encountered, the decision is sent to the stakeholders, and the response becomes part of the phone policy, and this transitions the request from unknown to the specified subspace [1] (if authorized), and may extend the prohibited subspace. A revocation of permission results in the access request associated with the revocation returning to the unknown subspace.

- Identify what to ask - In order for the stakeholders to make a well-informed decision based on their policy, the following information needs to be transmitted by the Client Reference Monitor (CRM); the access request comprising of subject, object, task being performed, unique application identifier, and finally state information. State information refers to the set of permissions already existing on the phone. Maintaining a separate state unique to every set of permissions is unfeasible. Hence, in our framework we define sets of permissions as roles. Possible roles may include, WI-FI enabled, GPRS-enabled, handset microphone and speaker-enabled, and GPS-enabled, to name a few.

- Response from Stakeholder - The stakeholders provide administrative policy that may be cached at the proxy server (in the network) and policy server (on the device). The administrative policy associates applications and roles with rules for administering the MAC policy. The mechanism does not limit the types of rules, but in our initial approach, stakeholders may state no interest or conflict sets among roles (and the permissions associated with those roles). The policy server will find the role associated with a permission request, determine if this role is in conflict with one of the application's current roles, and, if not, return the new role and its specified permissions (including positive and negative permissions).

[1]The permission is added to the *specified* rather than the permissible or prohibited subspaces because such assignments are transient. In general, a stakeholder may have a choice of whether to grant or revoke the permission unconditionally, but at present, we only consider temporary assignments.

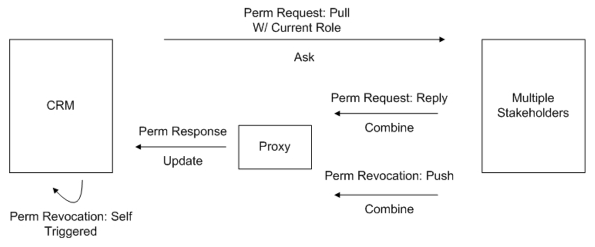

**Figure 1: Problem Overview**

- Response from Policy Server - There are two possible forms of policy update responses. The first is a transient update that is request-specific. The permissions are added, but they may expire or otherwise be revoked. This approach is better suited to high risk, downloaded applications with individual permissions that may need to be revoked. The second type updates permissions at the scale of the application or the application-run, where a batch of permissions analogous to a policy module are updated. It is not intended that such permissions be revoked individually or perhaps at all.

- Update client - Finally, after obtaining the permission response, we either make a transient change or a module change to policy. The roles of the application also may be updated.

Let us analyze the installation of a Internet based telephony application similar to Skype, on a handset implementing MAC in a telecommunications system. With the advent of Wi-Fi on handsets, such applications can be used to make free phone calls over the Internet bypassing the telecommunications infrastructure and costs associated with it. The first restriction for such applications comes from the service provider. Network operators do not want to allow end users to make calls through the Internet as this creates a loss of revenue for them. On the other hand, the phone manufacturer would have no problem allowing installation of such applications if its the telecommunications systems, as it increases the phone's appeal to consumers and may result in increased sales. Finally, the application provider might want to limit the user to only 20 executions of the application since it is a trial version and the user has not paid the requisite fee. Here, the phone manufacturer, network operator, applications provider, and the end user are all stakeholders in the system, and have specific access control restrictions that need to be enforced. Let us see how this plays out in our scenario.

When the Internet telephony application first runs on the MAC enabled phone, it requests permission for access to the handset speaker and microphone. The service provider stakeholder grants this permission since it is not a violation of policy. The phone changes state into the "handset speaker and microphone-enabled" role. When the application then attempts to access Wi-Fi to stream the voice, another request is made to the stakeholder, and this time, based on the stateful information that handset speaker and microphone is already permitted, the stakeholder can identify that an internet telephony application may be running and either deny the new request, or revoke the previously granted permissions and permit the WI-FI request. This would also change the state from "handset speaker and microphone-enabled" role to the Wi-Fi role. We note that other complexities in state might be involved in making a real decision on a working system, but we limit the example to easily understand the problem at hand.

Even if the service provider stakeholder decides through a change in policy that Internet telephony is allowed, the applications provider would still want to enforce pay-per-use applications or trialware, where application access is limited. In this case, the permission granted to the telephony application must be revoked after the requisite usage. One of the main challenges of policy revocations in distributed environments is the overhead associated with frequent policy downloads. This problem is compounded in an environment where users may install new applications regularly. Hence scalability must be carefully accounted for in any framework solution. Finally, as mentioned, one of our framework requirements is that depending on if the telephony application is recognized by a stakeholder, or is an unknown third party application, we either obtain an application specific policy, or a request specific response respectively and this affects ease of revocation which ultimately impacts scalability.

Figure 1 presents the interaction of various components in our example scenario. The CRM is responsible for policy enforcement on the client, and for transmitting permission requests with the information required for various stakeholders to make a decision. By using a CRM to identify the permission requests dynamically, we avoid the problem of requiring an external entity to create a list of static permissions that is required at install time. The proxy refers to an intermediate stage where information from multiple stakeholders is consolidated and staged.

## 3. APPROACH

In this section we present an architectural overview of our framework, followed by the details of our main functional mechanisms.

### 3.1 Architecture Overview

The main components of our framework as shown in Figure 2 are the Proxy Server, local policy server and MAC sandbox. We use a trusted channel like IPsec, SSL, or any other end-to-end encryption from the client to the proxy server to securely transmit information. The proxy server contains the various stakeholders' consolidated decisions. It serves as a staging point before permission responses are transferred to the local policy server. The MAC Sandbox is the client reference monitor that enforces policy on the end device.

Viewed differently, our framework components provides an interesting hierarchical structure as shown in figure 3. Top to bottom, the responsibilities gradually change from deciding the policy to enforcing the decision. Similarly, the policy representation becomes more specific, from policy subspaces and abstractions to binary "yes" or "no" decisions. The top tier consists of the various stakeholders and their individual policies. These are consolidated and staged at the proxy policy server which acts as a proxy and an abstraction to

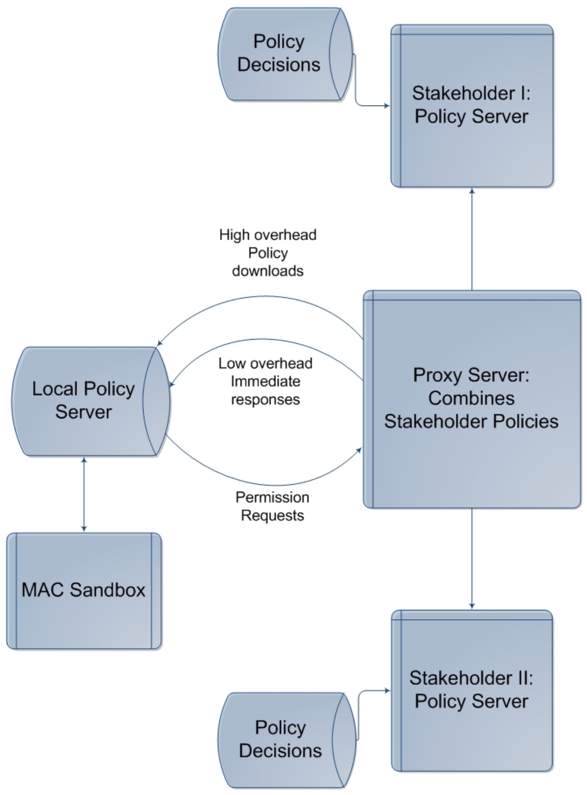
Policy Decisions
Stakeholder I: Policy Server
High overhead Policy downloads
Low overhead Immediate responses
Local Policy Server
Proxy Server: Combines Stakeholder Policies
Permission Requests
MAC Sandbox
Stakeholder II: Policy Server
Policy Decisions

**Figure 2: Framework Overview**

the lower tiers, simplifying their interaction to a single entity. Then we have the local policy server that processes permission responses as request specific (directly inserted into the Access Vector Cache (AVC)) and application specific which are processed through access vector table before insertion into the AVC. The AVC enables high performance, easy revocation and contains the final access decision.

### 3.2 MAC Sandbox

In order to identify permissions for an application, we use a strict MAC policy tailored to the unspecified subspace. We term this as a "soft" sandboxing solution or a *MAC Sandbox* since unspecified application requests get denied initially as if running in a sandbox. Thus when newly installed applications like our example Internet telephony application attempt to use cell phone functionality in the unspecified subspace, the MAC policy will isolate these requests. Conversely, specified subspace requests like core permissions and restrictions, are directly defined in the local MAC policy and do not require further processing.

In order for the stakeholders to make stateful decisions, permission requests are transmitted with a predefined subset of possible conflict sets abstracted as the current role. This role is intialized at boot up, and varies according to the set of permissions currently in the client's possession. We define the request consisting of the source label, target label, requested operation, and role as a tuple that uniquely defines an access request.

We transmit this access request tuple to the proxy policy

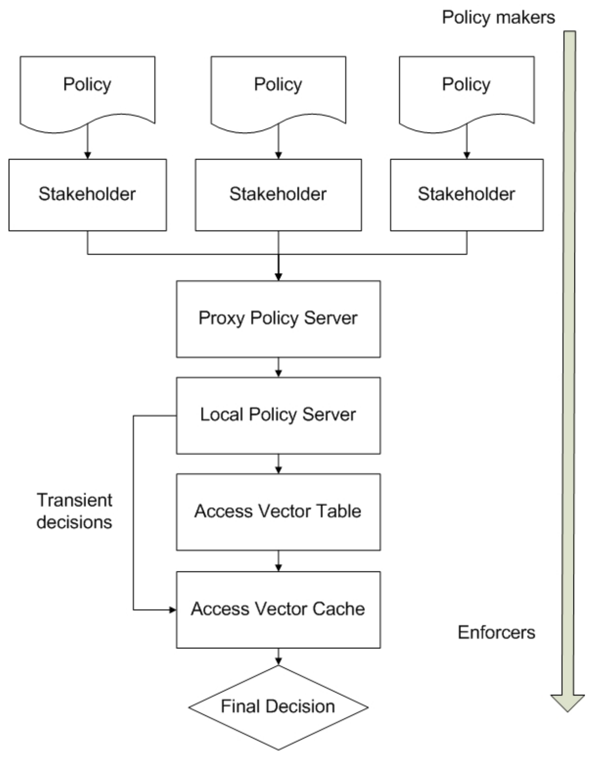

**Figure 3: Hierarchical flow diagram from the policy creators to enforcers**

server before the denial is final. We term this run-time permission request generation as a *dynamic manifest* (e.g., like a Google manifest which provides permissions for an application). We note that this is only a manifest in principle, since these requests are generated on resource access and they are not consolidated.

An important advantage of our dynamic manifest solution is that an application's unused privileged functionality does not affect its operation. This is unlike an install time certification system like Symbian where static policy directly prevents installation regardless of usage.

Another advantage of our framework is that it ensures applications run with least privilege. This is because permission requests are only sent for accesses that are denied, and the application is only granted the minimum permissions it requests and nothing more.

## 3.3 Proxy/Policy Server

Request tuples need to be resolved according the multiple stakeholders. We may use a staging point called the proxy server to support the same. Various policy servers corresponding to the different stakeholders are connected to the proxy server. If bypassed, the client needs to query every stakeholder's policy server individually and cache responses in the local policy server. The remote proxy server design avoids this by handing policy consolidation externally and thus minimizes the complexity and delay overhead for the client. The trade-off is the overhead of an additional component in the framework. We note that although the proxy server component is optional, the rest of this paper assumes it is part of the framework. Implementations without the proxy server are straighforward as its functionality is just pushed to the local policy server.

Policy rules are generally defined as explicit "allow" with default deny to support no interest. Explicit "deny" rules may also exist, that are overarching restrictions on request tuples. This is termed as strict-policy. In contrast, a targeted-policy scheme is where all undefined requests are allowed and the policy only consists of "deny" rules. However this is less secure as we then have to define every malicious scenario.

In the case of multiple stakeholders, policy consolidation can be handled in the following semantically different ways.

- All allow - All policy servers must explicitly have an allow ruleset. Implemented by anding all access bit vectors. Extremely conservative.
- Any allow - At least one policy server must have an allow ruleset results. Implemented by oring access bits vectors. Extremely generous.
- Consensus - At least one policy server must have an allow ruleset and none with an explicit deny ruleset. No fault tolerance nor conflict resolution.
- Priority - Sum of priorities of allow ruleset policy servers greater than those with explicit deny. Implemented by weighted sum of bit vectors with priorities as weights. Higher ranked stakeholders can override lower ranked ones. Provides some fault tolerance and conflict resolution.

While consensus provides us with a basic implementation that is elegant and simple, it does not take into account complexity in treating all stakeholders equal. For example, a secondary stakeholder say, the manufacturer, should not override a primary stakeholder, say the telecommunications provider, in the context of a particular access decision. We can implement these kinds of complexities using a priority based approach with different priorities associated with each tuple and stakeholder combination. A complete identification and mapping of the intersection and unions of different stakeholders and permissions is out of scope and left as future work.

We now look at how state information, i.e. role transmitted with the dynamic manifest helps us make permission decisions at the proxy server. In our example case, we assume that the initial Wi-Fi Internet access request is first granted and is inserted into the cache. Following this, the application then attempts to access the handset microphone and speaker. Now this request will have the existing state/role of Wi-Fi enabled. Thus the proxy server can make a stateful decision denying access and preventing end users from accessing Internet Telephony over Wi-Fi. If Wi-Fi was not already enabled, the proxy server would have permitted access to the handset microphone and speaker as the end-user cannot use Internet Telephony unless both are active.

### 3.4 Incremental Policy Addition

The decision payload returned from the stakeholders or proxy server, is transferred to the local policy server. Here depending on the category of response, i.e. permission or application specific, local policy updates are performed.

Permission specific responses are termed as incremental policy addition. We explain its operation through our example scenario. The Wi-Fi access request from our Internet telephony application is in the unspecified subspace and correspondingly the local policy server transmits a request tuple to the proxy server, which replies with an instant request-specific response. The local policy server receives this response, identifies it to be request specific, and inserts into the local client cache. Since the cache is always checked before consulting the local policy, we exploit this layer of indirection to enforce system policy. We minimize the performance overhead as once the cache is populated, subsequent permission responses are returned directly through cache hits.

The main advantage of this online approach is twofold. First, downloading complete policy modules involves significant overhead for every permission reply received. Modifying a monolithic policy involves downloading the new binary policy to the phone, which is in megabytes, and inserting it into the kernel. Even in modular form, binary policy modules for meaningful applications like firefox or thunderbird are about half a megabyte in size and need to be inserted into the kernel. Unknown high risk applications conceivably result in frequent permission requests and revocations exacerbating the problem. Second, and more importantly, it greatly simplifies revocation, which is critical in dynamic environments. Since these transient decisions are only inserted into the cache, revocation becomes as easy as invalidating the cache either incrementally or completely. Thus in this mechanism, we exploit cache poisoning as a fundamental feature to provide low overhead policy enforcement and easy revocation.

### 3.5 Batch Policy Module Insertion

The application specific solution is utilized if the application making requests is recognized by the stakeholder and it has a module available that encompasses resource requirements at the proxy server. We assume our Internet telephony application is a store application bought from a stakeholder. Known applications are recognized by Application IDentifiers (AppID) that are provided by stakeholders to developers as a premium service for money. In this case, the application module is provided directly in response to the access request, and the local policy server inserts the module into the client MAC policy. This mode of batch operation where a complete permissions manifest is downloaded one-time to the client, is more akin to traditional policy servers that provide on-demand policy modules. Revocations are a little challenging, as policy changes will require a new module download to override existing policy. However as this option is used only for recognized applications, the frequency of revocations should be low and thus easily handled. In this mechanism, we exchange the advantage of easy revocation for a more persistent solution that alleviates per-access response delay.

### 3.6 Revocation

The telecommunication system was initially built to be a closed network, but with the advent of smart phones, we see a constant evolution of permissions in an dynamic environment, where millions of end users routinely install and uninstall new applications. In order for the service provider to effectively control the end points of this network, revocation of permissions is a high priority. Our framework has been carefully designed with revocation in mind, so it supports both mass revocation and functionality-specific revocation.

We first look at revocation of the request specific, incremental decisions. The permission replies from the policy server are inserted into the cache instead of modifying the phones local policy directly. Now resetting the phone to its original state and revoking all permissions can be performed by a simple cache invalidation. Apart from mass revocation, we can also invalidate cache entries line by line by simply removing them individually from cache. Invalidation can be triggered by requests from the policy server as and when required.

Revocation of the application specific, policy insertions need to be handled a little differently. In this case, in order to invalidate permissions the old policy module will require removal and a new module is reinstalled at the client. However as policy insertions are performed only for stable recognized applications, there is less need for revocation in this case.

Another interesting feature of our framework is the ability to support pay-per-use applications. We associate a counter with each cache entry that is intialized with the desired number of permitted accesses. Then decrement this counter evertime a service is accessed by an application. When the counter reaches zero, we simply invalidate the cache entry denying further access to the service. Thus, a user may be allowed to access internet telephony services for say, up to 50 times a month or send 100 SMS's a month.

## 4. IMPLEMENTATION

In this section we discuss the implementation details of our framework. We use the SELinux LSM module in the Linux kernel version 2.6.27.2 to implement the base MAC system for our framework, and modify it according to our requirements. Our experiment shows that it is relatively simple to introduce this functionality in the kernel with minimum overhead and modular changes. A thorough study of the SELinux module architecture was required. Establishment of a secure channel for communication like IPSEC or TLS is a requirement for our approach. In the telecommunications network, it is not unrealistic to assume a secure channel between the user and the core network. The hardware used was a Dell Optiplex GX620 machine with Intel Pentium Dual Core processor @ 3.20 GHz. In order to enable basic SELinux support the kernel needs to be recompiled after enabling the SELinux options like CONFIG_SECURITY_SELINUX and CONFIG_SECURITY_SELINUX_BOOT SELinux has two modes of operation namely, Enforcing mode and Permissive mode. In enforcing mode, SELinux will actually prevent those operations that are not permitted by the policy, whereas in permissive mode, a violation is just logged for audit purposes and the operation is allowed to continue. We boot the kernel in permissive mode for testing purposes.

## 4.1 SELinux

SELinux uses a reference monitor to mediate all access on a system. It is predominantly a type enforcement system (TE). All entities are labeled with contexts, and when a subject tries to access an object, both the source contexts (scontext) and target contexts (tcontext) are checked for access control in the security policy. SELinux was modified to operate as part of the Linux Security Module (LSM) framework. While the LSM uses TSIDs (Target Security Identifier) and SSIDs (Source Security Identifier) to identify the source and target objects, these get translated to source and target contexts in the SELinux module. These contexts stay consistent between machines as long as the same policy is loaded and this enables us to maintain consistency between the policy server and local system. We have identified that transmitting the source and target contexts, along with a class identifier and the access control bitmask is enough to uniquely identify a permission request and its response. The context refers to the label of the entity, and the class identifier corresponds to the various entity types like file, socket, directory etc. The access control bitmask, avc->allow, refers to a 32 bit identifier where each bit refers to either an allow or deny decision for various operation requests. When a subject tries to access an object, this bitmask is identified for the source-object context pair and corresponding class, in the Access Vector Cache (AVC). If not present in the cache, the bitmask is looked up from the Access Vector Table (AVT) and then inserted into the AVC. The access permissions requested are represented in the same format as the bitmask permissions. Except, only the requested permission bit is set, all other bits are zeroed. To identify if access should be granted or not, a bitwise "and" is performed between the requested and permissions bit mask, and if it is non zero, access is granted. Thus a tuple consisting of scontext, tcontext, class, and bitmask are sufficient for access decisions. Apart from the avc->allow bit access vector described above, there are two other access bit vectors namely avc->auditallow, and avc->deny. The auditallow vector corresponds to one that grants permission and has its output logged in the audit report. This is useful for troubleshooting and logging purposes. The deny access vector is similar to the allow vector, except it is initialized as ones and zero corresponds to allow. These access bit vectors used together allow us to obtain a fine grained policy.

## 4.2 MAC Sandbox

The MAC sandbox is the primary component in the implementation of the dynamic manifest concept. The implementation framework is outlined in figure 4. We define the handset user *sysadm_t* as our subject and a new object type *untrusted_t*, which corresponds to an unknown third party application. The phone's pre-configured local MAC policy is limited to only core applications and this becomes our specified subspace. The specified subspace is inserted into the cache as part of normal SELinux operation when an access request causes a lookup from the access vector table. However, when a user of type sysadm_t attempts to execute untrusted_app, it is not allowed as this permission is not enabled in the local policy (unspecified subspace). This simulates running unknown applications in a sandbox, and this becomes our dynamic manifest.

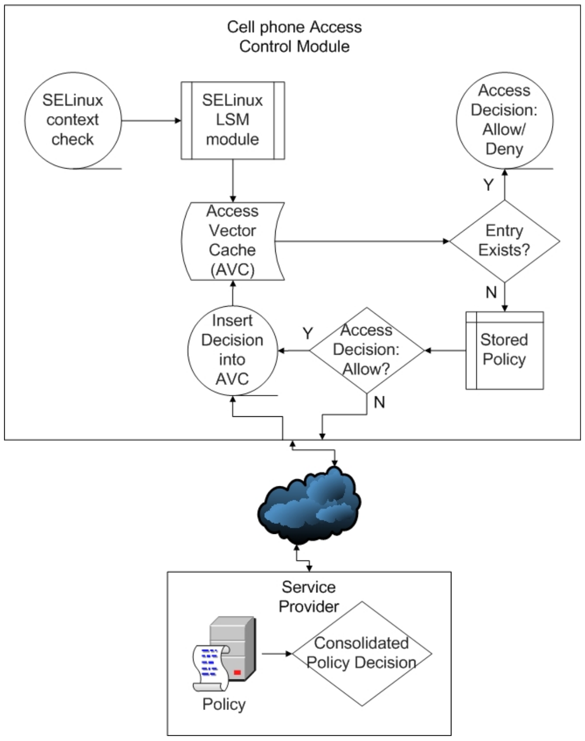

**Figure 4: Implementation Framework for MAC Security Model**

Once we obtain the unspecified access request, we insert a hook and instead of the default deny message being logged, an independent kernel thread is spawned to communicate with the proxy policy server. The security tuple consisting of the source security context, the target security context, target class, the requested permission mask and the current role, if one exists is sent to the proxy server via a secure network socket. The proxy server on receipt of this message checks its consolidated policy database and makes an access decision. This decision is communicated back to the local policy server over the network. It is important to note here that the client does not busy wait for an access vector decision from the policy server. The kernel thread created earlier blocks while waiting for the access vector decision. On receipt of the packet from the proxy server, the local policy server performs a bitwise "or" between the local bitmask and the proxy server bitmask. This corresponds to a union of the local and remote decision vectors. This new bitmask is then "anded" with the requested permission vector to generate the allowed/denied decision. the local policy server inserts this access vector decision in the AVC. This functionality is verified by viewing the kernel log messages. Once a decision is inserted into the cache, the next time the same functionality is accessed, the access decision is returned as a cache hit. Thus the unspecified request, becomes part of the specified subspace and exists in the cache. It will once again be downgraded to unspecified only in the event of revocation.

Depending on the network traffic as well as the load at the policy server, the client might have to wait momentarily before it gets the correct access control decision from the policy server. This one time transient wait at the client before it can run the application is an example where system security is given higher priority over a minimal performance increase.

Revocation of granted permissions in case of an attack is also well facilitated. This is done by allowing cache invalidation of a single-entry or the entire cache by sending an explicit invalidate request from the server. The ability to perform single entry cache invalidation after specific number of accesses is supported by invalidating cache entries that exceed a specific number of cache hits. The only modification to support this feature was insertion of a counter variable in the cache entry. On every cache hit, the counter variable is decremented and when it reaches zero, the entry is deleted from the cache, and this revokes its permission.

## 4.3 Policy insertion

In order to support the policy insertion part of our framework, we wrote an SELinux policy module "trustedm.te" for our test application. The module creates a new object type called *trusted_t* that corresponds to a known phone application that is trusted and given access permission by the local policy server. The policy also allows the user of type *sysadm_t* to read and execute a file of type *trusted_t*. This precompiled modular policy can be inserted into the kernel through the *semodule -i <module name>* command dynamically without recompilation of the entire policy. When a user of type sysadm_t attempts to execute trusted_app, it is now granted permission to execute by phone's local SELinux policy. This is confirmed by the AVC logs. Revocation is achieved by using the *semodule -r <module name>* command to remove the corresponding policy module.

## 4.4 Proxy Policy Server

Policy consolidation is performed at the proxy server using any of the techniques mentioned in section 3.3. When the MAC sandbox sends a permissions request to the proxy server, it checks the transmitted AppID to confirm if it is a known application. If it is unknown, an incremental access decision vector is returned to the local policy server according to the consolidated policy. Instead, if the request is for a known application with a policy module already existing at the proxy server, we just send the compiled binary module to the local policy server for insertion. Similarly revocation requests can be initiated from the server whether it is a cache invalidation or module removal request.

## 5. DISCUSSION

In this section we discuss performance of our implementation and some additional features that can be easily integrated into the framework.

## 5.1 Performance Overhead

Our framework relies on a modified SELinux module that sends access requests to the proxy server after lookup in the cache and local policy. Since SELinux has now been standardized in Linux kernels, introduction of our framework into an existing system should be relatively easy. In terms of performance our framework does not add much overhead to the users handset operations. This is because although the first access for a new application will be delayed at least by round trip time to the policy server, subsequent decisions are cached and returned locally with minimal overhead. By increasing the size of the cache, before entries are replaced, we can further ensure the performance overhead is limited only to first time access and all subsequent accesses are returned locally through the cache. The advantages we gain in exchange for this initial overhead include many novel features. These include easy revocation and reduced overhead compared to SELinux policy insertions.

**Table 1: Implementation delays for a single access request**

| | |
|---|---|
| Data Payload Size | 28 bytes |
| Processor Speed | 3.2 GHz |
| Plain SELinux Kernel | 946 cycles = 0.295 $\mu$s |
| W/Local Policy Server | 1485 cycles = 0.464 $\mu$s |
| W/remote Policy Server | 1870 cycles = 0.584 $\mu$s |
| Single Request Overhead | 924 cycles = 0.288 $\mu$s |
| Network Round Trip Delay | 3 ms (approx) |

The kernel patch has not been completely optimized as yet and currently consists of about 256 lines of kernel code. However, performance testing even without optimization only showed 0.288 microseconds overhead in executing our framework. Of course, this is independent of round trip time which varies depending on the network used and is probably a bigger bottleneck. However, we mitigate its influence by intelligent caching design and further, core applications and stakeholder recognized applications will have their security vectors in the local policy itself and only newly installed, unidentified third party applications will have this overhead, limited to the first time functionality is accessed.

As we see, local network roundtrip time domintates the framework overhead. The alternative to our framework in distributed environments is to download a new binary policy module each time a permission request is sent to the policy server. An average application policy module like firefox or thunderbird is approximately half a megabyte in size. Each time we want to update the local policy, no matter how trivial the change, the entire binary module needs to be transferred and installed into the handset. Consider a 1.5 MBps connection transferring 0.5 MB for each policy update. This scenario becomes readily unscalable with total overhead reaching magnitudes of 33 seconds as the number of policy updates approaches 100. By comparison, our framework scales efficiently in the same scenario to just 300 ms.

We also present our motivation for using the access vector cache to store transient decisions, in figure 5. From this graph, we see that after the system stabilizes, all security decisions are returned from the cache directly. The number of cache misses does not increase after 100s while the cache hits continue to increase steadily. Thus after the system converges to a stable state, communication between the local policy server and proxy server will be minimized and round trip delay will not impact the system significantly over time. Only in the event of a revocation, the cache gets reset and the system will need to re-insert the new security decisions into the cache. However, the alternative of downloading entire policy modules instead of cache insertion, will

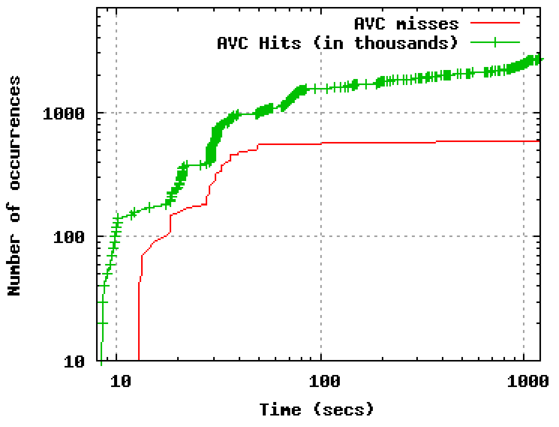

**Figure 5: AVC statistics vs Time**

result in much greater overhead. We leave further research regarding optimization of cache size, and dealing with cache replacement as future work.

### 5.2 Flagging

Having a tightly coupled security framework between the handset owner and the service provider allows both parties to maintain control over content and services. For example, permissions requested that are not granted by the handset local policy have to be granted by the service provider. This means that all access requests can be logged and maintained at the proxy server per user. This kind of flagging allows the service provider to have greater visibility over the services accessed by third party applications. For example, a third party SMS application installed by the user will require permission to access SMS services. When the user requests this permission from the proxy server, the service provider can flag the user as one with greater risk of misusing SMS resources. Thus if the user's handset appears to perform an anomalous operations like sending hundreds of SMS's shortly after installation of a third party application, the service provider can identify the software at runtime. This is in contrast to a framework like "Symbian Signed" which performs extensive source code analysis on every signed application to provide the same feature. This also allows the service provider to maintain higher vigilance on users who install high risk content on their handsets. For example, an elderly couple might never install the latest third party games from an online site. Hence they will not have many requests for permission and will not require careful monitoring. Whereas, a user that installs many third party applications with access to critical services can be flagged as high risk, and monitored closely for any anomalous behaviour. Intensive fine grained security flagging is often very useful in identifying security holes in a system quickly but there is a corresponding tradeoff in terms of privacy. However, not all loss of privacy infringes upon the user rights and many may be willing to balance their privacy requirements for security. However a thorough study of privacy issues and user behaviour is out of scope for this paper.

### 5.3 DySEL as IDS

Since we already have the complete policy module for premium and core applications, any denied requests for the same can be assumed to be a deviation from its normal profiled behaviour. For example, through a code injection attack like a buffer overflow, a highly privileged phone banking application attempts to access the Bluetooth device to offload user information. In such a scenario owning to the least privilege permissions constraints, the phone banking application should not have any access to the Bluetooth device, and any such resource request can be identified as a deviation from normal profiled behaviour and classified as an attack. While we note that the MAC framework will prevent such an attack, we can additionally flag these deviations to provide basic Intrusion Detection System (IDS) capabilities.

### 5.4 AppID

The AppID assigned for each premium application that is recognized by the proxy policy server is universally unique and it provided as a wrapper along with the installation package. This AppID is transmitted as part of the access request tuple and enables easy identification of the premium application and its corresponding tailored policy. Unknown applications are assigned an AppID that is locally unique on installation, but may have duplicates globally. However during analysis and identification of rogue applications this locally unique AppID can be combined with a unique identifier for the phone like IMSI (International Mobile Subscriber Identity) or IMEI (International Mobile Equipment Identity).

## 6. RELATED WORK

### 6.1 Multiple Stakeholders and Access Control

Certificate based techniques on the topic of multiple stakeholders and distributed policy have been explored in Akenti, however the authors deal more with access to distributed resources, which is different from our goal of a tightly coupled local CRM enforcing policy provided by multiple stakeholders [8]. Tresys has implemented an SELinux policy server that allows remote administration of SELinux policies, however, it is limited to single administrator, pre-computed policy modules that allow remote installation [14]. Static manifests techniques like kirin are not suitable for scenarios where the manifest itself is highly dynamic [15]. Certificate based techniques have been already used in practical systems like Symbian [12], but have weaknesses similar to static manifests. Further more, external certification leads to increased costs for third party developers, as they must get their applications certified for a fee, in case of any modification.

Tools like audit2allow, have previously used the set of all denied messages obtained from an installed application to identify the permissions it needed [14]. However this was performed off-line using log files to generate new policy files which allow the application to operate. This approach is not dynamic since we would need to obtain the appropriate log files, and run the tool each time to generate the new policy. Further, the newly generated policy needs to be compiled and inserted into the kernel. Finally, we would still need to identify the permissions that should be allowed or denied, since we cannot perform a blanket guarantee on the entire log file and allow all requests.

## 6.2 UCON

Usage control (UCON) which is a generalization of traditional access controls, trust management, and digital rights management is a systematic approach for next generation access controls [6]. Traditional access control is limited to a closed system with a server-side reference monitor where all the users are known. Trust management covers authorization for strangers in an open environment, and digital rights management deals with client side control of information usage. UCON scope can be mapped in terms of different kinds of reference monitors, and payment options. In terms of UCON scope traditional access control and trust management is performed using a server side reference monitor (SRM) while usage control is performed using a client side reference monitor (CRM). For our discussion we only focus on access control in UCON. While UCON in general can include both SRM's and CRM's we compare our framework to the generalized domain of UCON. We use SELinux as a standard CRM but instead of using an SRM, which enforces the policy, we just identify the decision and pass it to the client for policy enforcement. This prevents the overhead of enforcement at the server and client, under the assumption of a trusted computed base (TCB) that includes the server, client and communication channel between them. Although we provide some resource control in the sense, we can limit the number of execution attempts on functionality, our main focus in this paper is restricted to access control. Further our framework is flexible enough to incorporate more usage control, and increase the amount of state involved in policy decisions in the future.

## 7. CONCLUSIONS AND FUTURE WORK

We have presented a novel security framework using Mandatory Access Control (MAC) in a distributed environment. The main contribution of this paper was to provide a security framework with support for new features like multiple stakeholders, dynamic permission derivation, and extensive revocation support. Our framework was implemented by modifying the SELinux module and using a policy server database that can validate security permission requests in real time. The performance overhead introduced in the kernel was minimal at 0.288 $\mu$s. Our model provides an end-to-end solution using SELinux deployment and is beneficial to providing system security in a distributed environment.

In our future work, we plan to study the following: Impact of having policy servers outside the TCB and possibility of collusion among policy servers. A more in depth study of consolidation techniques of different stakeholders and compliance testing to ensure conflicts are resolved. Further research regarding optimization of cache size, and dealing with cache replacement.

## 8. ACKNOWLEDGMENTS

The authors would like to thank Sudeep Dutt for his help in implementation of this framework, and Dr Patrick McDaniel for his early guidance.

## 9. REFERENCES

[1] D. Bell, L. L. Padula, M. Ben-Ari, G. Benson, and G. . . . . Secure computer system unified exposition and multics interpretation.

[2] D. Brewer and M. Nash. The chinese wall security policy. *Security and Privacy, 1989. Proceedings., 1989 IEEE . . .*, 1989.

[3] D. Ferraiolo, J. Cugini, and D. Kuhn. Role-based access control (rbac): Features and motivations. *Proceedings of the Eleventh Annual Computer Security . . .*, 1995.

[4] C. Guo, H. J. Wang, and W. Zhu. Smart Phone Attacks and Defenses. In *Proceedings of Third ACM Workshop on Hot Topics in Networks (HotNets-III)*, 2004.

[5] T. Jaeger, X. Zhang, and A. Edwards. Policy management using access control spaces. *ACM Transactions on Information and System Security (TISSEC)*, 2003.

[6] R. S. Jaehong Park. The UCON usage control model. In *Proceedings of ACM Trans. Inf. Syst. Secur., vol 7*, pages 128–174, 2004.

[7] S. Kelby and T. White. The iphone book: how to do the things you want to do with your iphone. 2007.

[8] Mary Thompson, William Johnston, Srilekha Mudumbai, Gary Hoo, Keith Jackson, Abdelilah Essiari. Certificate-based Access Control for Widely Distributed Resources. In *Proceedings of the 8th USENIX Security Symposium*, pages 215–228, August 1999.

[9] C. Miller, J. Honoroff, and J. Mason. Security evaluation of apple's iphone. 2007.

[10] National Security Agency. Security Enhanced Linux. `http://www.nsa.gov/selinux`.

[11] Symbian Limited. Symbian OS - the mobile operating system. `http://www.symbian.com`, 2006.

[12] Symbian Limited. Symbian Signed. `http://www.symbiansigned.com`, 2006.

[13] P. Traynor, V. Rao, T. Jaeger, P. McDaniel, and T. La Porta. From Mobile Phones to Responsible Devices. Technical Report NAS-TR-0059-2006, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, 2007.

[14] Tresys technology, SETools policy tools for SELinux. `http://www.tresys.com/selinux/selinux\_policy\_tools.shtml`.

[15] William Enck, Machigar Ongtang, and Patrick McDaniel. Automated Cellphone Application Certification in Android (or) Mitigating Phone Software Misuse Before It Happens. Technical report, Pennsylvania State University, 2008.