High Assurance Policy-Based Key Management at Low Cost

Tolga Acar, Lan Nguyen
Microsoft Research, Microsoft
One Microsoft Way, Redmond, WA 98052, USA
{tolga,languyen}@microsoft.com
http://research.microsoft.com/en-us/people/{tolga,languyen}

***Abstract*—Past decade has witnessed the availability of Trusted Platform Modules (TPM) on commodity computers. While the most common use of TPM appears to be BitLocker on Windows OS, server class motherboards have not yet enjoyed a similar TPM deployment base. Recent research and products show that the TPM can provide a level of trust on locally executing software. Nonetheless, TPMs haven't been utilized in data center cryptographic key management for higher levels of security assurance than software-only techniques. Hardware-based key management has so far been constrained to higher cost add-on hardware.**

We present a large scale policy-driven cryptographic key manager built with TPM security assurances. We describe our design principles and axioms, architecture and abstractions, security policy, and implementation. We create a role-based security model and express the model with SecPal security policy assertions. We describe our implementation of three roles, actions, resources, SecPal policies and tokens that combine them. Finally, we present our implementation results with SecPal proof graphs.

***Keywords*-TPM; cryptography; key management; SecPal; policy language; data center**

I. Introduction

An enterprise or a user need some level of assurance before entrusting a service from a hosted service provider. With multiple benefits of what is collectively called "the cloud", more data and processing are moving to the cloud sush as e-mails, company documents with various forms of Intellectual Property, family documents and media files, and financial transaction data and services [1].

Under the cloud paradigm, suppose you are one of the CxOs in a company or in your family as the family IT person responsible for technology decisions and implementing them. You have already made the decision to move to a hosted provider to reduce costs and improve IT quality for your company's staff. You now have to choose a storage and processing provider for your Human Resources (your spouse), Legal (your spouse), R&D (your children), and Finance (you) departments. One of the providers runs third-party certified data centers in a physically controlled environment at premium price. Another one is running software of the month with the latest feature updates from its R&D department at a low price. Which provider would you choose?

None of the service level agreements from the providers satisfies your stringent requirements, and you decided to setup hardware and software in order to provide some of the services on-premise. Now you face another decision similar to the one with hosted service providers with a twist. Some of the software and hardware is third-party certified. They should be operated in a physically controlled environment by trained personnel at a high price point. The competing set of lower-priced software runs on commodity hardware and requires common IT administration skills, but does not offer comparable security assurances. Which combination would you choose?

But, wait! Some researchers claimed that it is possible to securely bootstrap in a "trusted" platform on commodity hardware [2], [3]. Other researchers claim they have a method to provide high-assurance hosted and on-premise services on commodity hardware. Do you really trust applied researchers for what you really wish existed in a production environment?

A. Problem Description

We consider this high-level problem: the lack of low-cost, automated cryptographic key management in large-scale data centers. We skip the motivation for catering cryptography and cloud key management, and assume both are necessary. We justify our assumption with publicly disclosed storage compromises and private communications with people who expressed a strong need for higher assurance key management and data protection for SaaS, PaaS, and other * *as a Service* offerings.

We recognize that one can build a high assurance key management and data protection with special purpose cryptographic hardware across many servers [4], [5]. These solutions require special-purpose hardware plugged in to each participating machine where high assurance cryptographic storage and processing is deemed necessary, such as top-level CAs. We want to come up with a low-cost key management alternative substituting special purpose cryptographic hardware with TPMs. It is also possible to create a hybrid approach with TPMs and Hardware Security Modules (HSM): that is not our focus.

In addition to cryptographic and TPM details, there is the *Trust Management* question in distributed and decentralized systems [6], [7]. Trust management is an access control approach in decentralized distributed systems with authorization decisions based on policy statements made by multiple principals [8], [9]. We focus on trust management for cryptographic key management and data protection to deploy in real-life distributed systems. Examples of such systems are Microsoft Azure and Amazon EC2.

B. Background

We look at previous trust management, key management, and public key certification systems. We embrace some of the public key certificate principles, reject others, integrate public key certification with SecPal policy language, and present a policy-based integrated key manager. Our system binds explicit authorizations to keys similar to SDSI [10]. We use SecPAL instead of SDSI's canonical S-expressions, but the authorization anchors remain similar. We qualify "trust" to mean carrying out actions granted to a principal, and nothing else. This will make more sense when we define roles in Section IV.

We don't associate public keys with identifiers (names) as in X.509 and PGP; public keys are identifiers themselves [11]. We represent a small set of mutable trust anchors, called `Root`, by their public keys through decentralized policies. Our policy allows automatic and manual adjustments to the trust anchor public keys, and avoids the single-instance X.509 trust anchor frailty. We achieve this through SecPal delegation.

The public key revocation story is simple: there isn't any. We stop granting access instead of revoking existing access. When a public key (a TPM) is no longer trusted for key management, we generate new keys, but don't grant access to the undesired public key access to the new keys. Existing keys are marked "Read-Only": decryption and verification, only.

SDSI can bind names and keys, but names are meaningless in our arhcitecture, perhaps except for auditing purposes: one might want to put the server name in an audit log in addition to a TPM key. That approach may prove to be less useful since names may change in a data center deployment. A more appropriate audit strategy may utilize the physical or logical server location. But, we separate human-usable audit from authorization, and treat it as a seperate mapping system. We follow the SPKI observation that a TPM public key is a globally unique identifier [10], which makes this mapping a bijection. We assume that unalterable TPM public keys can easily map to more meaningful audit information than X.509 style name assignment.

We are not going to describe logic based policy languages, role-base trust management languages, but refer to existing literature [12], [9], [13]. We describe how we used role-based access control with TPM keys to create a distributed key management and data protection system.

We have to comment on the most widely used security mechanism: Access Control Lists (ACL). It is almost customary for anyone to ask this question: "Why can't you just put an ACL on it?". We give a modified list inspired from Blaze, et.al, in [6],

- *Centralized Identity.* An ACL requires that the identities be globally recognized in the system. We have a distributed and decentralized topology where existing identities vanish due to failures or reinstallation, and new identities are introduced frequently. We can't rely on a centrally-managed identity system with ACLs attached to objects.
- *Authentication.* We can't rely on an existing authentication system: our system is designed to form a basis of trust for authentication. Servicing a key management request requires some sort of authentication, and a public key-based authentication fits naturally with TPMs.
- *Delegation* is required for fault tolerance and scalability in a distributed system. One of the tenets is the automatic trust adjustment by shifting within the same role. For example, in a failover cluster, the same request can be serviced by any role instance (assuming a role based cluster service model). Another aspect is the automatic role assignment to assure a minimum number of role instances in the distributed network or a subnetwork. Delegation makes the automation easy and allows unattended operation.
- *Expressive Policy.* ACLs are traditionally live out their expressiveness, and policies are hard-coded frequently in code. Our system requires dynamic and frequent changes to the authorization policy.

C. Contributions

Our cryptographic high-assurance key management uses commodity TPM hardware as tamper-resistant key storage and processing units. We give the role-based security model of our network key management. Our authorization approach combines previous decentralized and distributed trust management concepts with a security policy assertion language, and makes role-based access control decisions.

We define mappings between cryptographic keys and roles, objects for cryptographic key management, crypto-agile data protection, and object access permissions for roles. We also present a network service architecture based on these roles and observe principle of least privilege to reduce exposure to security threats.

Additionally, we achieve all of this with commodity hardware at significantly lower cost than special cryptographic hardware. As a demonstration, we discuss two sets of implementations. The first is a Windows C/C++ implementation that implements TPM interfacing, cryptography, network services and protocols, and data protection parts. The second software component implements security policy language,

claims and policies, policy decision and enforcement points in C#. We present the software architecture and design, SecPal base policies, SecPal assertions in authorization tokens for roles, and give SecPal authorization proof graphs.

II. TPM Background

The main output of the Trusted Computing Group (TCG) is a set of specifications of a hardware chip: Trusted Platform Module (TPM) [14]. A TPM is a security hardware with a public-private key-pair, secure memory locations for *Platform Configuration Registers* (PCR), a secure cryptography engine, and a programmatic interface to carry out predefined functions. The TPM v1.2 specification requires that TPM hardware to prevent disclosure of the private key against software attacks. Recent results showed that TPM hardware security measures aren't on par with assurances against software attacks [15]. As a design principle, we assume that TPM provides adequate protection for private keys within their bounds. In other words, our security value propositions are limited by what TPMs offer.

Each TPM is equipped with a special key-pair, an RSA key-pair, *Endorsement Key* (EK), that is either generated by the TPM manufacturer, or by the user when taking ownership of the TPM. We use $K_p^{(i)}$ and $K_r^{(i)}$ to represent the public and private Endorsement Keys of TPM i. We use $K_p^{(i)}$ to build multiple trust rings.

TPM can generate additional key-pairs for different purposes. Every key in the TPM has a parent key. The private portion of a key-pair can't be exported from the TPM in cleartext form; but it can be encrypted with its parent key when stored externally to the TPM. The root of the parental tree of keys sits the *Storage Root Key* (SRK). The SRK never leaves the TPM, and forms the root of trust for externally stored TPM keys.

All of the keys mentioned above are local to one TPM. Any data encrypted with these keys can only be decrypted by that TPM. We extend TPM security assurances to a network, and create a large-scale key management architecture. As we elaborate below, we require that each computer be equipped with a TPM.

Assume that a set of TPM public keys $\mathcal{T} = \cup_i \{K_p^{(i)}\}$ exist. One can establish secure channels between the TPMs represented by their public keys $K_p^{(i)}$ and $K_p^{(j)}, i \neq j$. If i is the requestor and j is the responder, we encrypt requests with $K_p^{(j)}$ and sign with $K_r^{(i)}$. We encrypt and sign responses with $K_p^{(i)}$ and $K_r^{(j)}$, respectively.

We considered using TPM keys to establish a TLS session. Unfortunately, non-legacy TPM keys can't be used with PKCS#1 formatting, and makes it impossible to do a TLS handshake. We considered defining a custom TLS ciphersuite, but saved it for a later project. Since out communication protocol is stateless, we did not create a stateful session, but instead relied on message security (both encryption and signature) with TPM keys.

III. Principles and Axioms

A data center forms the baseline for hosted services, sometimes collectively known as "the cloud". This project exploits gaps in both hardware and software utilization to increase the security of cryptographic keys in the cloud, and provides a distributed cryptographic key management and arbitrary data protection. The hardware gap is the underutilized TPM (Trusted Platform Module), and the software gap is the distributed cryptographic key management and data protection.

Our trust management approach is based on he following principles:

- **Cryptographic Processing and Storage Security.** We previously utilized an existing secure repository and communications channel for cryptographic key management and data protection [16]. Long-term secrets and policies can be stored in an access-controlled repository in cleartext, and a network protocol can protect them while in transit. We want to further strengthen cryptographic storage and processing security to address misplaced or stolen storage, minimize adverse impacts due to configuration problems, reduce liabilities of repository administrators, and minimize damage from rogue software. We must eliminate cleartext storage of secrets and release encrypted secrets only to authorized parties.
  While TPM encryption and storage of TPM-encrypted secrets address the plaintext *storage* threats, cryptographic *processing* with long-terms secrets occurs in software outside the TPM boundary. This is in contrast with an HSM that provides both: secure *storage* and *processing*. We strive to reduce HSM costs in large data centers by selectively replacing them with TPMs. Runtime code integrity can be provided by TPM style measurements or shielding key management and data protection code in a secure execution environment. We avoid discussing secure execution environments, and focus on trust management, cryptography, security policy, and crypto-agile data protection.
- **Fault Tolerance.** We assume that at any time, any machine may fail, network may fail and partition, and any TPM may irrepairably fail. Despite such failures, the system should eventually recover. We don't assume a globally consistent state. The state of a machine may differ from others, and inconsistencies may result in intermittent decryption failures. One of our goals is to minimize decryption failures without manual intervention.
  Despite the lack of a global and decentralized topology, an authorization decision must not change from "Allowed" to "Denied". We recognize that an entity may be granted access to a resource, but we don't want reversion in the other direction. We make two

observations on revocation and monotonicity [17]. We don't have the equivalent of "certificate revocation lists". Access removal is equivalent to removal of a role assignment to a TPM public key.

- **Commodity Hardware-Based Trust.** We aim at providing higher assurance security properties than a software-only solution at a much lower cost. We don't intend to offer equivalent security properties of Hardware Security Modules (HSM). On the contrary, once can view this as an untried disruptive technology.
- **Decentralized Trust Management.** Each machine in the network must be able to make an authorization decision whether to accept an incoming request. For scalability and to expand beyond single hierarchy of trust, we avoid the need of a globally known, monolithic hierarchy of certification authorities.
- **Flexibility.** Our policies are rich enough to express complex network trust relationships, but simple enough to design, implement, and evaluate for higher security assurance. We don't want comprehensiveness-through-complexity: we want KISS (Keep It Simple Stupid) with enough expressiveness for sound engineering. For instance, we have no ambitions to translate existing X.509, PGP, SDSI/SPKI certificates for our use.
- **Cryptographic Agility.** Cryptographic properties for data protection must be configurable and separable from those employed in authorization. We don't spend as much time on cryptographic agility details in this paper, but carry forward existing research into this architecture [18], [19].

IV. Architecture and Abstractions

Our topology is comprised of multiple, possibly thousands of, computers in one or more data centers. In particular, we target large data centers with many racks of computers.

This project utilizes TPMs to encrypt long-term secrets. As a stretch goal, we also use DRTM to determine authorized parties to release TPM-encrypted secrets.

The distributed network topology contains machines each equipped with a TPM chip. The cryptographic algorithms and key sizes are constrained by TPM's cryptographic capabilities. We expand the set of supported cryptographic algortihms at the data protection level to use contemporary strong cryptographic algorithms, and also provide cryptographic agility.

We use configurable cryptographic algorithms for arbitrary data protection. TPM keys protect data protection keys, and cryptographic processing for data protection is carried out on the CPU. We don't directly use TPM's cryptographic keys and processing for data protection due to (a) lack of cryptographic algorithm agility, (b) limited processing latency and throughput. We make this seemingly futile compromise, but describe a TPM-provided integrity mechanism to alleviate some of the concerns.

A. Authorization

We employ a role-based access control mechanism expressed in a logic-based security policy language that is suitable for distributed environments. We used a policy language that allows delegation and fine-grained expression of capabilities: SecPAL [12], [20]. This apporoach removes the need to have access control information attached to objects. It is a claims-based approach and removes the need to maintain and replicate authorization lists.

Our architecture requires that each request and response is validated for proper authorization before an action is taken on them. In this sense, we have a *reference monitor* with a few subcomponents: TPM public keys, roles, actions, and policies. We use the term *action* instead of *permission* in role-to-permission mapping [21]. Our authorization does not use names or location.

Let $\mathcal{T}$ represent the set of authorized TPM public keys $K_p^{(i)}$ for TPM i, $\mathcal{R}$ be the role set, $\mathcal{P}$ be the policy set, and $\mathcal{A}$ be the action set. In order to grant permissions to TPM keys, or allow TPM keys to carry out certain actions, we assign actions to roles, and assign roles to TPM keys. A surjective function $\phi_A : \mathcal{A} \rightarrow \mathcal{R}$ maps permissions (actions) to a role, and a surjective function $\phi_R : \mathcal{R} \rightarrow \mathcal{T}$ maps a role to a TPM public key. The ϕ_R and ϕ_A mappings are the authorization policy. Section V describes the SecPAL statements to express the authorization tokens.

In RFC2693 style, we define an authorization mapping as follows where the first mapping is ϕ_A and the second mapping is ϕ_R [10].

$$\text{Authorization} \rightarrow \text{Roles} \rightarrow \text{Key}$$

If a name is needed for audit purposes or punishment for some reason, a key may be mapped to a location (MAC address, rack location, etc.) to have:

$$\text{Authorization} \rightarrow \text{Roles} \rightarrow \text{Key} \rightarrow \text{Location}$$

We only use the first mapping, and rely on operational procedures for secure location discovery. In SecPAL parlance, an authorization decision is the result of a query to the datalog engine with the authorization tokens from the incoming request and authorization policies $\mathcal{P}$.

B. Roles

Our TPM-based key management architecture defines three roles: $\mathcal{R} = \{\texttt{Root}, \texttt{Store}, \texttt{Node}\}$ (Figure 1). Table I tabulates role descriptions.

- **`Root`.** Root role establishes the root of trust, similar to the Certificate Authorities in traditional PKI. Administrators can create and modify configuration data, setup and operation policies, lists of machines, and their TPM keys and roles on a machine in Root role. In order to do that, Administrators sign the updated data with the TPM Signing Key and distribute the signed

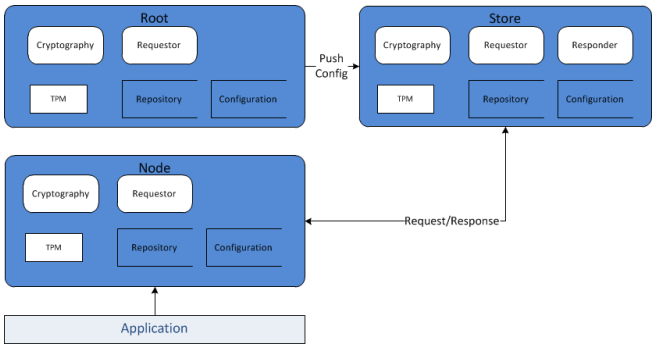

Figure 1. Roles and data flow between role implementations.

Role	Description
Root	Root of Trust for TPM public keys Role assignment to TPM public keys Push configuration to Stores Sync between Roots
Store	Repository: keys, policies, and configuration Network responder (HTTP) Synchronization and replication between stores
Node	Cryptographic operations Client API Sends requests to Stores In-memory encrypted key caching

Table I
DESCRIPTION OF ROLES.

manifests to machines in Server role. Other machines in root role retrieve the latest configuration from machines in Server role with an automated background daemon, or an administrator can push the new configuration to the machines in root role.

- **`Store`.** Machines in `Store` role provides policy, key, and configuration storage, synchronize them among other machines in store role, and respond to queries from machines in root, store, and node roles. The Store class machines form the distributed repository and automated key lifecycle management backbone. They are not directly callable by applications: that role belongs to the node role discussed next.
- **`Node`.** Nodes are the application class machines and expose data protection APIs to applications. Node class machines retrieve keys, policies, and configuration from Stores, protect them with local TPMs, and store them locally for caching purposes. Nodes can send various requests to Stores tabulated on Table II. They also get updated configuration data from Stores.

C. Actions and Resources

Table II defines SecPal verbs and resources. The verbs define actions $\mathcal{A}$ on resources $\mathcal{S}$: a group identifier ID_g a key identifier ID_k, and configuration data `file:///Config`. From the table, $\mathcal{A} = \{$`create`, `delete`, `read`, `send`, `write`, `update`$\}$, and $\mathcal{S}=\{$`file:///Config`, `[policy:`ID_g`]`, `[keyId:`ID_k`]`, `[groupName:`ID_g`]`$\}$.

Without loss of generalization, we use `[policy:Global]` to refer to the global policy instead of the group-specific policy `[policy:`ID_g`]`. The latter indicates a policy resource of group ID_g. For brevity, we use the resource name `[policy:Global]`, `[keyId:`ID_k`]`, and `[groupName:`ID_g`]`in our SecPal statements instead of adding a possess constraint claim with an identifier ID_g, ID_k, and ID_g, respectively.

One can think of resources as function parameters, and the verb as the name of a function in the traditional sense. For instance, a `Node` would call the `create` function with parameters ID_g and ID_k, the function would packet the request, attach authorization claims to the request, and send the signed request to a `Store` machine. After authorization checks, the `Store` machine would process the request and respond back.

Our verbs don't require a state; an important protocol design principle for scalability. All requests are simple request-response type calls and can easily be encoded in popular formats: SOAP, JSON, and others. The requestor split a large semantic process into multiple requests, and sends each one in order. For instance, when deploying a new group, the requestor first creates the group container ID_g, creates a default policy in the group, and creates a new key with ID_k in group ID_g, and updates the group policy with the new ID_k.

D. Permission Assignment

Permission assignment is one of the core tenets in a role-based access control system [21]. We allow ourselves a great deal of flexibility with the claim-based SecPal approach instead of deciding on a fixed and globally-recognized set of permissions. An ACL-based approach would not offer this level of extensibility, should we have chosen a more traditional approach.

Permissions are not directly assigned to principals; we assign permissions to roles by way of granting certain actions to roles. Another interesting aspect is the resource the permission is granted: that is not encoded in principal SecPal claims, either. Instead, our policies grant a role in $\mathcal{R}$ one or more permissions to resources via actions $\mathcal{A}$. Our SecPal assign permissions by "r CanDo a on s", and roles by "t possess r", where $r \in \mathcal{R}, a \in \mathcal{A}, s \in \mathcal{S}, t \in \mathcal{T}$. The permission assignment are in the base policy assertions, and the role assignment claims are in tokens issued to principals.

E. TPM Key Hierarchy

We define a fairly flat key hierarchy that ultimately anchors in a root of trust, a node in a `Root` role. This trust begins with a small configuration file deployed with software to each machine. The initial configuration file contains the DNS names and TPM public keys of at least one `Root` and

Table II
VERBS AND RESOURCES THE KEY MANAGEMENT SCHEME RECOGNIZES.

Verb	Resource	Description
`create`	ID_g	Create a new group `[groupName:`ID_g`]`
	ID_g, ID_k	Create a new key ID_k in group `[groupName:`ID_g`]`
	ID_g	Create a new default policy in group `[groupName:`ID_g`]`
`delete`	ID_g	Delete group `[groupName:`ID_g`]`
	ID_g, ID_k	Delete key ID_k in group `[groupName:`ID_g`]`
`read`	ID_g, ID_k	Read key ID_k in group `[groupName:`ID_g`]`
	ID_g	Read policy and group metadata in group `[groupName:`ID_g`]`
	`file:///Config`	Read configuration
`send`	ID_g, ID_k	Send key ID_k in group `[groupName:`ID_g`]`
	ID_g	Send policy in group `[groupName:`ID_g`]`
	`file:///Config`	Send configuration
`write`	ID_g, ID_k	Write key ID_k in group `[groupName:`ID_g`]`
	ID_g, `[policy:`ID_g`]`	Write policy in group `[groupName:`ID_g`]`
	`file:///Config`	Write configuration
`update`	ID_g	Update key in group `[groupName:`ID_g`]`

at least one `Store` machine. Rest of the configuration is automated and is pushed out from one of the `Root` machines to `Store` nodes declared in the initial configuration file. This approach allows minimal configuration at deployment time, and hands-off and self-adjusting configuration at run-time. However, setting up an automated roots of trust environment doesn't say anything about data protection. For that, we define a TPM key hierarchy.

The Endorsement Key identifies a TPM and forms the basis of trust. But, EK is of limited use for network request authentication and data encryption: we need additional keys. Figure 2 depicts our key hierarchy. Keys in red are unique in our architecture. There is one signing key, one key wrapping key, and one TLS key per TPM; although the last key is not strictly required. The signing key signs all outgoing requests, and wrapping key protects keys and other sensitive data coming in and going out of TPM. The signing key, wrapping key, and TLS key are asymmetric native TPM keys. There is one or more "DKMK" symmetric keys protected with the wrapping key. The "DKMK" keys are symmemtric keys and are processed in computer memory once decrypted by the TPM. The "DKMK" keys are cached in memory, and software cryptographic libraries provide high-performance data protection. We also create a TLS key, but that key is not used due to incompatibilities of TPM v1.2 RSA operations and the TLS algorithm.

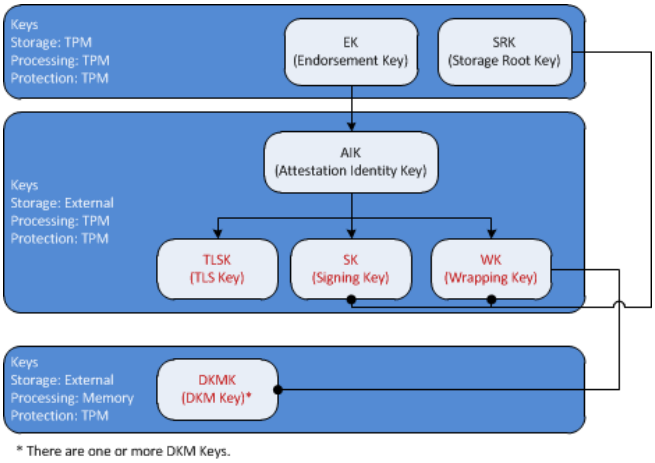

Figure 2. TPM Key Hierarchy.

V. SECURITY POLICY

In the policy descriptions below, we assume that the organization running the key management service is the trust anchor, and delegates trust to one or more administrators. The flow of trust continues with a trusted person authorizing one or more TPM public keys as trusted, and delegating trust machinations to those set of TPM hardware. Later on, we remove the key management trust anchor from the organization and transfer to a customer.

We express security policies using SecPAL [12]. The security policies include SecPAL facts, verbs, predicates, and conditions [20].

The trust anchor is the `Organization` principal that runs the data center. `Admin` represents the principal the `Organization` delegates a minimal set of privileges to make trust decisions. The `Admin` trust decisions are limited to defining the set of TPM public keys for each role: root, server, and node. Once those public keys are determined, the `Root` and `Store` roles are responsible for key management in the data center in an hands-off fashion. We will describe delegation in more detail using SecPal claims later on [12].

It is important to observe that the policies presented later in this section don't allow `Organization` and `Admin` principals to have access to the keys and data protected by those keys. We acknowledge that malicious organizations may find ways to defeat such policies. However, as we present later on, TPMs offer unique advantages over traditional HSMs when combined with run-time measurement facilities.

Let us map the abstract principals, `Organization` and `Admin`, to meaningful descriptions in a canonical

hosted services scenario: e-mail. The Admin is the e-mail service provider running e-mail servers and sells hosted e-mail services to businesses. A business corresponds to Organization that delegates key management decisions to the provider as it relates to e-mail services. The group ID_g is the unique identifier of the business, essentially a tenant identifier for the business customer in the e-mail service provider's account database.

A. Security Assertions

Table IV enumerates the set of assertions for the Root role. The Store and Node roles need additional assertions listed on Table V and Table VI.

ROOT ASSERTIONS. The first SecPal claim on Table IV says that the Admin can assign the Root role to any public key. In our proposal, we claim that all public keys must be TPM EK public keys. This is the first level delegation from Admin to the Root role to assign root of trust to TPM public keys.

The second claim says that any machine with a TPM in the Root role can assign Root, Store, and Node roles to any other TPM. This is the second level delegation from Root role to any of the other roles, and relieves Admin from dealing with lower level role assignments.

The remaining claims allow Root to carry out configuration actions: create, delete, read, send, and write. We elaborate on the remaining claims in Section V-C. In the remaining parts of this paper, we assume that the optional parts are not present in the configuration.

Our SecPal implementation doesn't yet support verb groups to express claims 3 through 7 on Table IV in one claim. We would like to create one assertion to address all five cases: "LA says k_1 can v file:///Config :- k_1 possess [roleName:Root], $v \in \mathcal{A}$" which also grants update permission in one assertion.

STORE ASSERTIONS. Root assertions say nothing about cryptography, keys, policies, storege, distributed repository, and data protection. They simply assert which TPMs are allowed to assign roles to accomplish worker tasks Store and Node. Table V gives a list of claims to allow Store TPMs to work with [keyId:ID_k], [policy:Global], and file:///Config resources.

Claims 1 through 5 grant cryptographic key management capability to the Store role.

NODE ASSERTIONS. Nodeassertions are not cryptographic, either. They only grant a TPM key in Node role to read configuration, policy, and keys.

B. Key and Group Separation

We have not yet describe finer-grain access controls that makes our system multi-tenant aware. When we say *multi-tenant aware*, we refer to the ability to non-verlapping use of different cryptographic policies and keys. This is

Table III
CONFIGURATION STRUCTURE.

Item	Description
Root List	TPM EK public key list in Root role.
Policy	Default cryptographic policy
Store List	TPM EK public key list in Store role (optional).
Node List	TPM EK public key list in Node role (optional).

Table IV
ROOT POLICIES. LOCAL AUTHORITY SAYS ...

1.	Admin canSay k_1 possess [roleName:Root]
2.	k_1 canSay k_2 possess a :- k_1 possess [roleName:Root], where a matches roleName:"Root|Store|Node"
3.	k_1 can create file:///Config :- k_1 possess [roleName:Root]
4.	k_1 can delete file:///Config :- k_1 possess [roleName:Root]
5.	k_1 can read file:///Config :- k_1 possess [roleName:Root]
6.	k_1 can send file:///Config :- k_1 possess [roleName:Root]
7.	k_1 can write file:///Config :- k_1 possess [roleName:Root]

Table V
STORE POLICIES. LOCAL AUTHORITY SAYS ...

1.	k_1 can create [keyId:ID_k] :- k_1 possess [roleName:Store]
2.	k_1 can delete [keyId:ID_k] :- k_1 possess [roleName:Store]
3.	k_1 can read [keyId:ID_k] :- k_1 possess [roleName:Store]
4.	k_1 can send [keyId:ID_k] :- k_1 possess [roleName:Store]
5.	k_1 can write [keyId:ID_k] :- k_1 possess [roleName:Store]
6.	k_1 can create [policy:Global] :- k_1 possess [roleName:Store]
7.	k_1 can delete [policy:Global] :- k_1 possess [roleName:Store]
8.	k_1 can read [policy:Global] :- k_1 possess [roleName:Store]
9.	k_1 can send [policy:Global] :- k_1 possess [roleName:Store]
10.	k_1 can write [policy:Global] :- k_1 possess [roleName:Store]
11.	k_1 can send file:///Config :- k_1 possess [roleName:Store]
12.	k_1 can read file:///Config :- k_1 possess [roleName:Store]

Table VI
NODE POLICIES. LOCAL AUTHORITY SAYS ...

1.	k_1 can read file:///Config :- k_1 possess [roleName:Node]
2.	k_1 can read [policy:Global] :- k_1 possess [roleName:Node]
3.	k_1 can read [keyId:ID_k] :- k_1 possess [roleName:Node]

where we introduce the concept of *group* and tie it to the abstract [groupName:ID_g] representation. A group is an independent entity with a cryptographic policy and one or more cryptographic keys. We assume a bijection to map a tenant to a group without losing generality in the following security assertions.

C. Configuration

Claims 3 through 7 on Table III gives Root ability to manage configuration. This statement requires further drilling down on configuration. Table III enumerates the item list in the configuration. As opposed to traditional X.509 certificates and certificate chains that require one or more Certificate Authorities, our approach begins with a list of public keys that represent Root roles. The initial Root public keys can extend this list by delegating the Root role to other TPM public keys, essentially spawning their equals. It is possible to put delegation restrictions, such as controlling the depth, we don't think that such a restriction is realistic in an otherwise flexible (some might call this *elastic*) computing, but recognize that it has an academic benefit. We will not elaborate on suhc restrictions in this paper, but simply note that SecPal is able to express various restrictions.

The third and fourth claims on Table III are optional. We would recommend them for small deployments when the total number of Root, Store, and Node machines is small, updates are infrequent but reliable. We would omit them when the number of machines grows, the network topology becomes more dispersed across multiple geolocations, and rate and duration of failures increase. In general, when configuration updates become problematic, we want to reduce the number and frequency of configuration updates. Removing the configuration parts that change often would reduce our exposure to update failures.

VI. Implementation

We implemented SecPal policies, role functions, network protocol, and cryptographic functionality, and data protection in C# and C++ languages. We give a few SecPal authorization query evaluation results, and describe them in detail.

SecPal query engine accepts three inputs: policies, tokens, and an authorization query. It generates a logical answer and a proof graph if there is one. Tables IV, V, and VI are the SecPal policies serialized in human-readable text form. The requestor attaches SecPal claims to a request token and signs them together. A query is an authorization request expressed as a SecPal claim. We give human-readable serialized SecPal logic statements in the examples below.

We look at the evaluation of a few scenaria and SecPal datalog engine answers to understand how assertions come into play in an authorization decision.

Table VII
Claims in the first query token.

1.	Admin says Root possess [roleName:Root]

Admin says Root possesses [roleName:Root].
  Admin says Root possesses [roleName:Root].
LA says Admin can say Root possesses [roleName:Root].
  LA says Admin can say k1 possesses [roleName:Root].

Figure 3. Query: LA says Root possesses [roleName:Root].

Example 1. We begin with an easy authorization query: "Does K_p possess [roleName:Root]?" We use "LA" to represent the Local Authority; all claims from this local trusted entity are treated as facts. Figure 3 shows the *LA says Root possesses [roleName:Root]* query, the resolved "yes" answer proof graph in an ordered set of logical claims.

The innermost (most indented) statements on Figure 3 are base facts, and are either in a base policy (Tables IV, V, VI) or in the query token (Table VII). The first statement, *Admin says Root possesses [roleName:Root]*, is a candidate proof for the query. It is resolved to the only base fact in the token in the indented second line. But, this is still not a proof: it is not connected to LA. This is where the third and fourth statements in the proof graph come in. The third statement connects the second statement to LA, and the fourth statement connects it to a base fact in the base policy. This concludes the proof by substituting k1 with Root.

Example 2. The second example is more complicated query: "Can K_p read a key?" Token claims are on Table VIII, and the SecPal proof graph is on Figure 4. As simple as the query is, the proof deserves some explanation. Here is a short logical recital of Figure 4 annotated for statement ranges.

1 It is enough to possess the Store role to read a [keyId:ID_k].
2-4 Store possess the Store role, because Root says so. Thus, we need to establish a fact to show that Root can say so.
5-6 LA says that any principal in the [roleName:Root] can assign any of the three roles to another principal. So, can we show that Root is in the [roleName:Root]?
7-9 Admin says that Root is in the [roleName:Root]. Now, we need to prove that Admin can say it.
10-11 LA says that Admin can assign any one of the three roles to another principal. This forms the base fact supporting the chain.
12 This is the regular expression matching constraint and satisfies conditions in claims 6 and 11.

Example 3. The third and final example is a failed authorization query: "Can a [roleName:Root] create a key?" The token claims are given on Table IX. There is no proof graph, because there is no affirmative answer to satisfy the

Table VIII
CLAIMS IN THE SECOND QUERY TOKEN.

1.	`Admin` says `Root` possess `[roleName:Root]`
2.	`Root` says `Root2` possess `[roleName:Root]`
3.	`Root` says `Store` possess `[roleName:Store]`
4.	`Root` says `Store2` possess `[roleName:Store]`
5.	`Root2` says `Store` possess `[roleName:Store]`

```
1 LA says k1 can Read digitalContent:file:///Keys :-
    k1 possesses [roleName:Store].
2 LA says Store possesses [roleName:Store].
3   Root says Store possesses [roleName:Store].
4     Root says Store possesses [roleName:Store].
5   LA says Root can say Store possesses [roleName:Store].
6     LA says k1 can say k2 possesses a :-
        k1 possesses [roleName:Root], where
        a matches roleName:"Root|Store|Node".
7     LA says Root possesses [roleName:Root].
8       Admin says Root possesses [roleName:Root].
9         Admin says Root possesses [roleName:Root].
10      LA says Admin can say Root possesses [roleName:Root].
11        LA says Admin can say k1 possesses [roleName:Root].
12    [roleName:Store] matches roleName:"Root|Store|Node"
```

Figure 4. LA says Store can Read digitalContent:file:///Keys

query *LA says Root can Create digitalContent:file:///Keys*. This is intentional and demonstrates one of the principles of least priviledge in our architecture: `Root` class nodes can't create keys, but can only assign roles to other principles.

It is trivial for a `Root` machine to assign a `Store` role to itself, effectively granting access to keys. We avoid the naive "unless" clause to plug this "hole" to deny access to keys if the principal is in the `Root` role; that would undermine the decentralized nature. As previously discussed, we don't want a yet unknown or witheld claim to negate an otherwise granted authorization request.

It is easy to see that our policies and claims in this section don't permit authorization in group granularity. The most straightforward approach to add group separation is to augment SecPal claims and policies by adding a possess claim constraint. We would add "k_1 possess `[groupName:`ID_g`]`" constraint to the policy claims on Table V and Table VI.

VII. CONCLUSIONS

We described the principles, architecture, and implementation of a policy-based cryptographic key manager. We chose a logic-based policy language, SecPal, created a role-based framework, and defined base policy claims for authorization. We assigned permissions to roles with the same policy language, and took advantage of SecPal's constrained delegation properties. We gave three results from our implementation: a simple role check, a more complicated permission check, and a failed authorization request.

Our three-role architecture is minimalistic in base policy, but yet allows decentralized authorization decisions by local evaluation.

As authorization future work, we want to employ an abduction-based algorithm to specify missing role memberships and verb grants without interacting with the resource guard [22]. We also want to look at using abduction as a policy verification tool.

Table IX
CLAIMS IN THE SECOND QUERY TOKEN.

1.	`Admin` says `Root` possess `[roleName:Root]`
2.	`Root` says `Root2` possess `[roleName:Root]`
3.	`Root` says `Store` possess `[roleName:Store]`
4.	`Root2` says `Store` possess `[roleName:Store]`
5.	`Root` says `Node` possess `[roleName:Node]`
6.	`Root2` says `Node` possess `Node`

ACKNOWLEDGEMENTS

Jason Mackay of Microsoft Research provided invaluable assistance with the development of the SecPal statements and their implementation.

REFERENCES

[1] C. Cachin and M. Schunter, "A cloud you can trust," in *IEEE Spectr.*, Dec. 2011.

[2] B. Parno and A. P. Jonathan M.McCune, "Bootstrapping trust in commodity computers," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[3] B. Parno, "Bootstrapping trust in a trusted platform," in *Proceedings of the 3rd USENIX Workshop on Hot Topics in Security (HotSec), USENIX*, 2008.

[4] Thales, *Hardware Security Module: Thales nShield Connect*, Jul. 2011, http://www.thales-esecurity.com/~/media/Files/Data%20Sheets/nShield%20Connect%20Datasheet.ashx.

[5] ——, *Thales e-Security keyAuthority*, Nov. 2011, http://www.thales-esecurity.com/Resources/~/media/Files/Data%20Sheets/keyAuthorityDataSheet.pdf.

[6] M. Blaze, J. Feigenbaum, and A. D. Keromytis, "The role of trust management in distributed systems security," in *Secure Internet Programming*, 1999, pp. 185–210.

[7] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized trust management," in *In Proceedings of the 1996 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1996, pp. 164–173.

[8] B. L. Ronald L. Rivest, "SDSI - a simple distributed security infrastructure," Tech. Rep., April 1996.

[9] N. Li and J. C. Mitchell, "Understanding SPKI/SDSI using first-order logic," *Computer Security Foundations Workshop, IEEE*, vol. 0, p. 89, 2003.

[10] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen, "SPKI certificate theory," *IETF RFC 2693*. [Online]. Available: http://tools.ietf.org/html/rfc2693

[11] P. Zimmermann, *PGP User's Guide*. Cambridge: MIT Press, 1994.

[12] M. Y. Becker, C. Fournet, and A. D. Gordon, "SecPAL: Design and semantics of a decentralized authorization language," *Journal of Computer Security*, vol. 18, no. 4, pp. 597–643, 2010.

[13] N. Li and J. C. Mitchell, "Design of a role-based trust management framework," in *In Proceedings of the 2002 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2002, pp. 114–130.

[14] Trusted Computing Group, *TPM Main Specification Level 2 Version 1.2, Revision 116*, Mar. 2011, http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[15] C. Tarnovsky, "Hacking the smartcard chip," in *Blackhat 2010*, Jan. 2010, http://www.blackhat.com/presentations/bh-dc-10/Tarnovsky_Chris/BlackHat-DC-2010-Tarnovsky-DASP-slides.pdf.

[16] T. Acar, M. Belenkiy, L. Nguyen, and C. Ellison, "Key management in distributed systems," Microsoft Research, Technical Report MSR-TR-2010-78, Jun. 2010.

[17] M. Y. Becker, A. Russo, and N. Sultana, "Foundations of trust management," in *IEEE Symposium on Security and Privacy*, 2012.

[18] T. Acar, M. Belenkiy, M. Bellare, and D. Cash, "Cryptographic agility and its relation to circular encryption," in *EUROCRYPT10*, May 2010.

[19] M. Bellare and S. Keelveedhi, "Authenticated and misuse-resistant encryption of key-dependent data," in *CRYPTO*, 2011, pp. 610–629.

[20] M. Becker, C. Fournet, and A. Gordon, "Design and semantics of a decentralized authorization language," in *IEEE 20th Computer Security Foundations Symposium (CSF)*, 2007.

[21] D.F.Ferraiolo, D.R.Kuhn, and R.Chandramouli, *Role-Based Access Control*. Artech House, Computer Security Series, 2003.

[22] M. Y. Becker, J. F. Mackay, and B. Dillaway, "Abductive authorization credential gathering," in *Policies for Distributed Systems and Networks*, 2009, pp. 1–8.