A Framework for Self-Protecting Cryptographic Key Management

Anne V.D.M. Kayem, Patrick Martin, Selim G. Akl, and Wendy Powley
School of Computing
Queen's University
Kingston, Ontario, K7L 3N6, Canada
Email: $\{kayem, martin, akl, wendy\}$@cs.queensu.ca

Abstract

Demands to match security with performance in Web applications where access to shared data needs to be controlled dynamically make self-protecting security schemes attractive. Yet, standard schemes focus primarily on correctness as opposed to adaptability and so need to be extended to handle these new scenarios. One of the approaches to enforcing cryptographically controlled access to shared data is to encrypt it with a single secret key that is then distributed to the users requiring access. Data security is ensured by replacing the group key and re-encrypting the affected data whenever group membership changes. Thus, key management (KM) is expensive when changes in group membership occur frequently and involve large amounts of data. This paper presents a framework, based on the autonomic computing paradigm, that allows a KM scheme to continually monitor the rate at which changes in group membership occur and generate keys as well as encrypted replicas to anticipate future changes. Since the keys and encrypted data are generated by anticipation rather than on demand, the long-term cost of KM is minimized. A prototype implementation and experiments showing performance improvements demonstrate the effectiveness of the proposed framework.

1. Introduction

The increasing complexity of managing security for the multiple and varied scenarios that arise on the Web have triggered an interest in applying the autonomic computing paradigm to designing self-protecting security schemes [6, 18, 19]. Standard methods of enforcing access control in Web-based applications include those that are supported by cryptographic key management (CKM) schemes. Unlike authentication schemes that rely on system-specific security policies, cryptographic access control (CAC) schemes do not rely on the physical security of the system on which the data resides [1]. CAC schemes use data encryption to enforce access control, making unauthorized access more difficult because the data remains encrypted irrespective of its location, and only a valid key can be used to decrypt it. Emerging Web-applications like data out-sourcing, collaborative project development, and pay-TV highlight a growing number of applications of cryptographic access control.

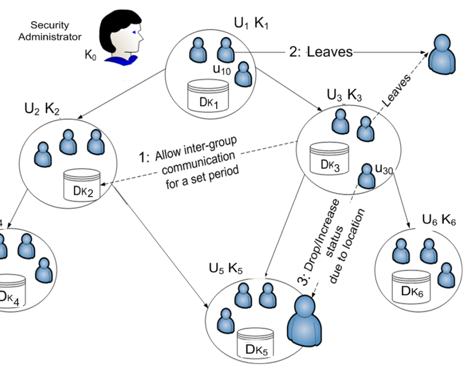

Figure 1. A Key Replacement Scenario

In collaborative applications, controlled access to shared data can be enforced cryptographically by classifying users into exactly one of a number of disjoint security classes U_i, represented by a partially ordered set $(S, \preceq)$, where $S = \{U_0, U_1, ..., U_{n-1}\}$ [1]. In the partially ordered set (poset) $U_i \preceq U_j$ implies that users in group U_j can have access to information accessible to users in U_i, while the reverse is not possible. Security management is facilitated by sub dividing the encrypted data into various categories D_{K_i}, such that $0 \leq i \leq n-1$, where n is the maximum number of user groups in the hierarchy and K_i is the cryptographic key used to encrypt the data d_i. Possession of a "correct" key grants a user access to the data. Key management with a CAC scheme like the one we have just described is expensive because updates require changing the affected key and re-encrypting the data. When large amounts of data are involved and rekeying occurs frequently, the key server takes longer to respond to rekey requests thereby increasing the system's vulnerability [9, 30, 33].

For instance, in Figure 1, when a user u_{30} moves into "unsafe" territory, the application reacts by either reducing his/her privileges and assigning him/her access rights that are equivalent to those assigned to users in U_6 or by locking him/her out of the system. In order to prevent u_{30} from continuing to access D_{K_3}, the key K_3 is updated. Additionally, if possession of the key K_3 allows a user to derive K_5 and K_6, both keys need to be updated as well. Likewise when U_2 and U_3 are merged, a new common key is required both for the new group U_x and the dependent groups U_4, U_5, and U_6. Rekeying implies data re-encryption, and so frequently rekeying large amounts of data increase the key server's response time[1] in handling key update situations thereby widening the vulnerability window[2]. A larger vulnerability window implies a longer wait period for the users remaining in the group and this period can also be exploited by an adversary. Hence, the cost of rekeying is a drawback to integrating CAC schemes into Web-based applications where security and adaptability to changing scenarios are important parameters.

This paper presents a framework based on the autonomic computing paradigm [19] that allows a CKM scheme to adapt to changing scenarios by minimizing the response time and the size of the vulnerability window created by frequent rekeying. The functionalities of the framework are structured into six components: the *sensor*, *monitor*, *analyzer*, *planner*, *executor* and *effector*, that are linked together to form a feedback control loop (FBCL). The FBCL continually monitors the arrival rate of rekey requests at the key server and, at regular intervals, computes an acceptable resource (keys and encrypted replicas) allocation plan to minimize the overall cost of rekeying. Each component of the framework contributes to enhancing a standard CKM scheme's performance without changing its underlying characteristics. A prototype implementation and experiments showing performance improvements demonstrate the effectiveness of the proposed framework.

The rest of the paper is structured as follows. Section 2 reviews related work on CKM schemes and autonomic security. In Section 3, we present our self-protecting CKM framework and give an example to show how it works. We present and discuss our prototype implementation as well as experimental results showing performance improvements, in Section 4. Concluding remarks are offered in Section 5.

2. Related Work

This section discusses background work on cryptographic key management (CKM) schemes in relation to the rekey problem, and the autonomic computing paradigm as a method of designing self-protecting CKM schemes.

2.1. Cryptographic Key Management

CAC schemes are typically designed using either an *independent* or *dependent* KM model. Schemes based on the Independent key management (IKM) model operate by assigning each class a single independent key. A user belonging to a higher level class is only allowed to access data at lower levels if he/she holds the "correct" lower level class key [13]. Rekeying is handled by replacing the affected group's key, re-encrypting the associated data and distributing the key both to the users remaining in the group as well as to the users belonging to higher level classes that are authorized to access data encrypted with the updated key.

While the flexibility of the IKM model makes it easy to implement in practical systems, the drawback is that all updated keys must be distributed to every class in the hierarchy that needs them to access data. Thus, key redistribution is costly and prone to security violations due to mis-managed or intercepted keys [13]. For example, in Figure 2(a.), the data d_0, d_1, d_2, d_3, d_4, and d_5 is encrypted with the keys K_0, K_1, K_2, K_3, K_4, and K_5 to obtain $D_{K_0}, D_{K_1}, D_{K_2}, D_{K_3}, D_{K_4}$, and D_{K_5}. In this case, the IKM model operates by assigning a user all the keys required to authorize him/her access to portions of the encrypted data. However, if a key, say K_4, is updated the new key needs to be re-distributed to all the users in the classes U_0, U_1, U_2 and U_4 that use it.

Variants of IKM schemes [4, 5, 9, 26] in the literature propose minimizing the information distributed either by encrypting the keys that are to be distributed with a public key or by using proxy re-encryption. In the first approach, the encrypted keys are placed in some public location and a secret key is transmitted to each group. Access to a set of keys is only allowed if a user has the correct secret key. This makes it easier to exclude users that are compromised and reduces the number of keys distributed, but the added public key information increases the chances of an adversary correctly guessing at the secret keys being used [9]. The second approach on the other hand, assigns each group or user in the hierarchy two pairs of keys (a master and a secondary key) [4]. The secondary key is used to encrypt files and load them into a block store where they are made accessible to users outside of the group. External users retrieve the encrypted data from the block store and present the retrieved data together with their secondary key to the access control server. The access control server re-encrypts the data in a format that can be decrypted with the user's secret (master) key, only if the presented secondary key authorizes him/her access. However, the problem remains of having to re-encrypt, update, and distribute new keys when group membership changes.

A good way to alleviate these problems is to minimize the number of keys distributed to any group (class) in the hierarchy. The *dependent* key management (DKM) model does this by assigning higher level classes keys that can be used to derive lower level keys. For example, in Figure 2(b.), the data d_0, d_1, d_2, d_3, d_4, and d_5 is encrypted with the keys K_0, K_1, K_2, K_3, K_4, and K_5 to obtain $D_{K_0}, D_{K_1}, D_{K_2}, D_{K_3}, D_{K_4}$, and D_{K_5}. Possession of the key K_1 allows access to D_{K_3}, and D_{K_4} since the key is associated with the class U_1 that is at a higher level than the

[1] Time required to generate a new key and re-encrypt the data associated with the key.

[2] Period between the emission of a key update request and its satisfaction by the key server.

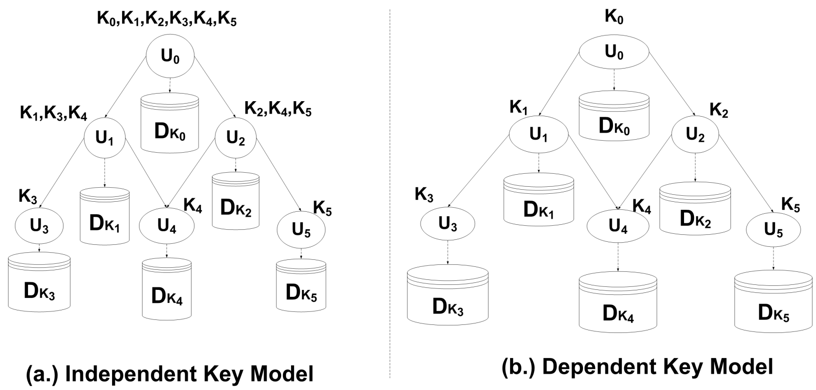

Figure 2. Independent vs. Dependent Key Approaches

classes U_3, and U_4, and the keys K_3 and K_4 are derivable from K_1. The reverse is not possible because keys belonging to lower level classes cannot be used to access information at higher levels.

Instances of the DKM approach in the literature [1, 10, 13, 16, 20, 25, 26] focus on efficient methods of minimizing the storage requirements of the keys and the cost of key derivation but they do not address the issue of key updates. Key updates are handled in these schemes by updating the whole hierarchy and re-encrypting the data. Recently, Atallah et al. [2] and Kayem et al. [17] have proposed methods of updating keys locally, i.e. in the sub-hierarchy associated with the affected class. However, in both schemes [2, 17] when rekeying occurs at the highest point in the hierarchy (e.g. U_0 in Figure 2(b.)) the entire hierarchy needs to be updated to ensure continued data security.

Time-bounded schemes [3, 7, 11, 29, 31, 32], address the key update (rekey) problem by associating a time bound to each key in way that allows a user to access both the encrypted data at his/her class and at lower classes during a specific interval. At the end of the interval, access is denied because the key is no longer valid. This makes handling key updates easier, but is not practical for scenarios where user behavior is difficult to foresee since it is hard to accurately predict time bounds to associate with keys.

Other schemes in the literature are lazy re-encryption, and timestamped schemes [8, 16]. Lazy re-encryption operates by using correlations in data updates to decide when to rekey. Since data re-encryption accounts for the larger part of the cost of key replacement, re-encryption is only performed if the data changes significantly after a user departs or if the data is highly sensitive and requires immediate re-encryption to prevent the user from accessing it. The cost of rekeying is minimized, but the problem remains of having to re-encrypt the data after a user's departure. Moreover, if a sensitive file does not change frequently, lazy re-encryption can allow a malicious user time to copy off information from the file into another file and leave the system without ever being detected.

The timestamped scheme associates each key with a timestamp. Both the timestamp and key are combined to compute a verification signature that is used to authenticate a user before access is granted to the data. Whenever group membership changes, instead of rekeying and re-encrypting the data associated with the keys, only the timestamp is updated and a new verification signature computed. This scheme significantly reduces the cost of rekeying, and so is interesting for dynamic scenarios. However, its reliance on authentication makes it vulnerable, in the sense that, if a malicious user holding a valid key finds a way of generating correct timestamps, there is no straightforward way of detecting or even preventing them from accessing the system.

From the previous paragraphs, we note that rekeying is expensive when it involves data re-encryptions because this widens the vulnerability window and when data re-encryptions are reduced in favor of multiple key distributions, there is an increased chance that the keys could be intercepted by an adversary. Authentication-based KM schemes minimize the cost of re-encryptions and reduce the size of the vulnerability window but the security they provide is system specific. Moreover since re-encryptions are done only once, if an adversary guesses at the "correct" authentication signature, it is difficult to detect or even eliminate them from the system. Therefore, standard KM schemes need to be supported by a framework that allows them make adjustments to security specifications based on the situation with which they are faced.

2.2. Security with Autonomic Computing

The autonomic computing paradigm has inspired the creation of numerous computing models aimed at coping adaptively with varying complex scenarios [18]. Yet, these methods have not gained as much popularity in the domain of access control due to skepticism and reluctance towards autonomic approaches on the part of the users [6]. Security scandals like the one that occurred in January 2007, when hackers broke into Winners'[3] computers and stole customer credit card information, generate public outrage that in turn

[3]Departmental store in the US and Canada specialized in clothing, shoes and accessories.

results in hesitancy in using less conventional security solutions [24]. Hence, business owners tend to opt for security schemes that react in pre-specified and predictable ways, as opposed to those that adapt and evolve dynamically. Increasingly, however, Web applications are faced with scenarios that are difficult to predict a priori, which makes manual security management challenging and prone to error [6, 18]. Breaches created by errors in security policy specifications are currently difficult to trace and prevent, and this will become even harder as systems become more complex [6].

Security via the autonomic computing paradigm was first proposed by Chess et al. in 2003 [6] to address the growing system complexity that makes manual security management time consuming and challenging. They suggested using the autonomic computing paradigm proposed by IBM in 2001 [18, 19] whereby, a system can be designed to use automatic reactions to self-configure and self-manage. The functions of an autonomic system are connected to form a feedback control loop (FBCL) that has two major components: the *autonomic manager* and the *managed resource*. The autonomic manager adjusts the behavior of the managed resource on the basis of recorded observations. The autonomic model

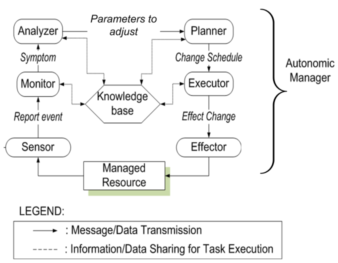

Figure 3. The Autonomic Computing Feedback Control Loop

shown in Figure 3, is comprised of six basic functions: the *sensor*, *monitor*, *analyzer*, *planner*, *executor*, and *effector*. The sensor captures information relating to the behavior of the managed component and transmits this information to the monitor. The monitor determines whether or not an event is abnormal by comparing observed values to preset maximum values in the knowledge base. If the deviations between the observed and maximum values are significant, the monitor transmits a message to the analyzer where a detailed analysis is performed to decide what parameters need to be adjusted and by how much. The analyzer transmits this information to the planner where a decision is made on the action to take. The executor inserts the task into a scheduling queue and calls the effector to enforce the changes on the managed resource in the order indicated by the planner.

Autonomic security aims to provide survivability and fault-tolerance for security schemes [6, 15]. Johnston et al. [15] propose a preliminary approach that uses reflex autonomicity in the development of a multi-agent security system. This is an interesting approach to self-protecting security schemes, but the authors indicate that a real-world implementation of their prototype system would require additional security controls. Moreover, as Moreno et al. [23] point out, the prototype does not support the ability of a security class to operate independently. We note also that current work on autonomic access control focuses mainly on security policy definitions and restrictions on the messages sent and received by users and/or agents in the system. The problem of supporting CAC schemes with a framework that allows them to adapt to different scenarios still needs to be addressed.

3. Key Management Framework

This section describes our proposed approach to self-protecting key management (SPKM) aimed at minimizing the key server's response time and the vulnerability window created in handling rekey requests. As mentioned before, the framework is composed of six functionalities - *sensor*, *monitor*, *analyzer*, *planner*, *executor*, and *effector* - connected in the form of a feedback control loop (FBCL). The FBCL continually monitors (over regular intervals) the arrival rate of rekey requests at the key server and uses a stochastic model to predict an acceptable resource (keys and encrypted replicas) allocation plan aimed at minimizing the overall cost of rekeying.

3.1. Framework Operation

According to this framework, rekeying is handled by monitoring the rate at which the key server (central authority) receives rekey requests and generating keys as well as backup replicas in anticipation of future rekey requests. Instead of waiting to generate replacement keys and re-encrypt the data associated with the affected key on reception of the rekey request, the SPKM framework minimizes the vulnerability window and response time by adjusting the number of available replicas based on observed user behavior patterns.

We assume that there exists a single trusted central authority (key server) U_0 in charge of key generation/assignment and data encryption. Each class in the access control hierarchy represents a group of users authorized to access a portion of the shared data. Rekey requests can be explicitly formulated by a user wishing to depart from a group, in which case the user transmits a message encrypted with his/her key to the key server. Alternatively, the key server can monitor a group's behavior and decide to exclude a user from a group. For instance if a user remains inactive for a long time then the KM system can lock him/her out for safety reasons. Knowledge of the membership of a group is located in a registry in the knowledge base. The registry contains the group and user identification of every user in the system as well as their associated secret keys. Cases of central server crashes are assumed to be handled by some fault-tolerance solution like server replication.

The key server generates a series of keys $K_0, ..., K_{n-1}$, according to the rules of access defined by the security administrator (SA). The key server then transmits these keys secretly to the data server where they are stored in a secret registry of keys. On reception of the key set, the data server proceeds to encrypt the data and transmits a confirmation message to the key server. The key server then proceeds to distribute the keys to the user groups requiring access. Data is only accessible to users if the key in their possession allows them to directly decrypt, or to derive the keys required to decrypt that data. For example, as shown in Figure 4, u_{20} can access data $D_{K_2}, D_{K_{n-2}}$, and $D_{K_{n-1}}$ since his/her key can be used to directly decrypt D_{K_2} and derive the keys K_{n-2} and K_{n-1} required to access $D_{K_{n-2}}$, and $D_{K_{n-1}}$ respectively.

As shown in Figure 4, our framework is comprised of six basic components that are interconnected in the form of a FBCL situated at the key server. The SA defines an initial observation period during which rekeying is handled solely by the conventional approach and data is collected to start off the adaptive KM process. This initial period is divided into two time windows denoted by W_1 and W_2. During W_1 the sensor captures and transmits all rekey requests to the monitor. At the end of W_1, the monitor computes, on a per class basis, the arrival rate of rekey requests and compares this value to a preset maximum arrival rate in the knowledge base. If the current arrival rate is greater than the maximum arrival rate, a message is transmitted to the analyzer indicating that the maximum arrival rate has been exceeded. The maximum arrival rate is reset to the current arrival rate while monitoring continues in the next interval W_2. When the current arrival rate is less than or equal to the maximum arrival rate, the event is discarded. The analyzer computes a probability prediction to determine whether or not the current resource (keys and backup replicas) allocation will satisfy the next series of rekey requests that arrives in the next time interval, W_2, and communicates this value to the planner where an acceptable number of resources (keys and backup replicas) is computed in anticipation of the next series of rekey requests. The planner then calls the executor to define a schedule for creating and assigning the resources, and the executor instructs the effectors to perform each task according to the priority defined. The maximum values (against which observed values are compared) are located in the knowledge base and are set by the SA on the basis of empirical observations. In the meantime, the monitor restarts the adaptive cycle by computing the arrival rate that occurs in W_2. The FBCL cycles continuously over time, analyzing the arrival rate for each subsequent period W_x that occurs after W_2. Copy consistency is maintained on the backup copies by periodically checkpointing the state of the primary on to the backups.

This approach has two main advantages. First, the size of the vulnerability window and response time between key replacements is reduced since rekeying is handled by anticipation as opposed to on demand. Second, the job of the SA is made easier, since the SA no longer needs to take care of

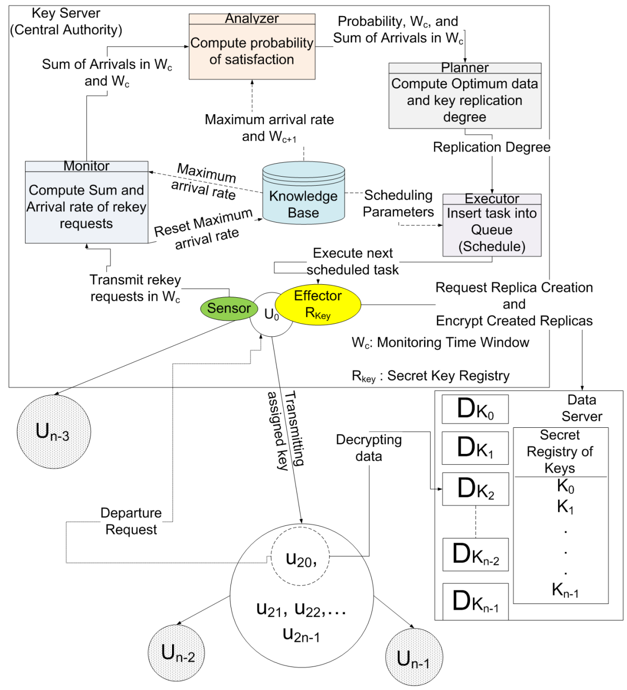

Figure 4. Self-Protecting Key Management Framework

every key replacement scenario but rather, the SA presets specific parameters and allows the scheme to run on its own. Cases directly requiring the SA would henceforth be limited to situations that require the consent/advice of the SA to proceed.

3.2. Self-Protecting Key Management (SPKM) Framework

Our SPKM framework uses a Poisson process to describe the rate of arrival of rekey requests at the key server because it is suited to modeling the occurrence of random events in time [12]. The arrival rate of rekey requests at the key server has this property of randomness as there is no way of accurately predicting the number of rekey events that will occur in a given period. Moreover, the rekey process has the Poisson property in the sense that the observed arrival rate at any point in time is independent of both present and future arrival rates.

For simplicity, we describe the mathematical model underlying our SPKM framework as though it were for a single class U_i, in the hierarchy. In reality, however, resource allocations are made for each of the classes in the hierarchy. We assume that rekey requests arrive independently at a rate λ and denote:

- W_c: The c^{th} predefined monitoring period (time window) for rekey requests arriving from U_i, such that $0 \leq c \leq I - 1$ and I is the maximum number of time windows

- λ_{c_i}: The arrival rate of rekey requests from group U_i during W_c
- λmax_i: The maximum arrival rate of rekey requests for group U_i that the key server has anticipated handling during W_c
- m_i: Total number of rekey requests that originate from group U_i during W_c
- p_{c_i}: The probability prediction that the key server will not be able to satisfy all the rekey requests that will arrive during the next monitoring period W_{c+1} (i.e. to determine whether the current numbers of keys and data copies will satisfy an arrival rate of at least λ_{c_i} during W_{c+1})

The sensor captures rekey requests transmitted to U_0 over a preset period W_c and transmits this information (number of rekey requests and size of the monitoring period W_c) to the monitor. At the end of the period W_c, the monitor computes the sum of the rekey requests received as well as the arrival rate λ_{c_i}. The arrival rate is computed with the formula:

$$\lambda_{c_i} = \frac{m_i}{W_c} \qquad (1)$$

where the arrival rate is measured in terms of number of requests per second. The monitor compares the value of λ_{c_i} to a preset value λmax_i that is located in the knowledge base. The preset maximum values and the size of W_c are set by the SA on the basis of empirical observations. If $\lambda_{c_i} > \lambda max_i$, a message is transmitted to the analyzer indicating that the previous λmax_i has been exceeded, and λmax_i is reset to λ_{c_i}.

The analyzer computes p_{c_i} by computing the probability mass function of the Poisson variable m_i. The reason for using this formula is that it forms a part of the properties of the Poisson model that facilitate making predictions on the basis of very little information, and serves as a simple prediction tool. The probability prediction p_{c_i} is computed using the formula [12]:

$$p_{c_i} = \frac{\mu_i^{m_i}}{m_i!} * e^{-\mu_i} \qquad (2)$$

where

$$\mu_i = \lambda_{max_i} * W_{c+1} \qquad (3)$$

is a prediction of the number of rekey requests that are expected during W_{c+1} and e is the base of the natural logarithm.

The analyzer decides on whether to increase or decrease the resources (number of keys and encrypted backup replicas), by comparing p_{c_i} to a preset probability prediction value, ϵ. If $p_{c_i} = \epsilon$, the value is discarded. Otherwise, if $p_{c_i} < \epsilon$ the analyzer calls the planner with an instruction to decrease the resources, and if $p_{c_i} > \epsilon$ the planner is called with an instruction to increase the resources. This is to ensure that an optimal number of resources is always maintained so that the costs of updating the backup copies do not outweigh the benefits of adaptive KM.

On reception of the value of p_{c_i} and instructions regarding how the resources should be adjusted, the planner proceeds to compute a degree of availability. The degree of availability α_i allows the planner to decide on how to adjust the number of resources to keep the cost of running the system within acceptable limits. Availability, is defined as the fraction of W_{c+1} that the key server is in a position to satisfy any rekey request it receives. In order to determine α_i, we need to know the state of the key server. We consider that the key server can be in one of two states: the ***normal*** state or the ***idle*** state.

- ***Normal State:*** This is the state in which the key server performs two kinds of activities: key distribution or key generation. Let T_R be the total time that the system spends rekeying (generating and distributing keys) during the window W_{c+1}.
- ***Idle State:*** This is the state in which all the required keys have been generated and the key server has relegated the tasks of encryption and checkpointing to the data server [14, 21, 22]. Let T_E be the total encryption time during the window W_{c+1} and T_C be the total checkpointing time.

Our method of computing α_i is inspired by the approach proposed by Jalote [14]. Jalote expresses α_i as a probability function of the overhead producing activities in a system. We extend this concept to our SPKM framework and express the availability of D_{K_i} as follows:

$$\alpha_i = 1 - \frac{O_i}{W_{c+1}} \qquad (4)$$

where O_i is the overhead generated during rekeying, encrypting and maintaining update consistency on the backup copies at the class U_i. Since rekeying, encryption, and checkpointing all contribute to the overhead, we compute O_i with the formula:

$$O_i = E(T_R) + E(T_E) + E(T_C) \qquad (5)$$

where $E(T_R)$ is the expected rekey time (i.e. the expected time required to generate and distribute a key to satisfy a rekey request), $E(T_E)$ is the expected encryption time, and $E(T_C)$ is the expected checkpointing time. Our framework does not handle requests that are not completed during W_{c+1}, so we will assume that all rekey requests that arrive during the time window W_{c+1} are completed before the end of W_{c+1}, otherwise they are processed during the next time window W_{c+2}.

In order to compute the total rekey time during W_{c+1}, we need to determine the fraction of W_{c+1} during which the key server is going to be generating and distributing keys. If the key server is unable to satisfy all the requests it receives during W_{c+1}, the key server will be in a state of key generation and/or distribution during $p_{c_i} * W_{c+1}$ time. Therefore, the theoretical estimate $E(T_R)$ of the expected rekey time can be computed using the formula:

$$E(T_R) = p_{c_i} * W_{c+1} \qquad (6)$$

Rekeying implies re-encrypting data therefore the encryption time is conditioned by the number of rekey requests that will occur during W_{c+1} and the number of backup data copies (replicas) that need to be encrypted/re-encrypted. We denote N_i as the number of replicas and keys that will need to be maintained at class U_i to satisfy μ_i rekey requests in W_{c+1} and compute $E(T_E)$ with the formula:

$$E(T_E) = p_{c_i} * W_{c+1} * N_i \qquad (7)$$

We note that the theoretical estimate of the total time during which the system would not be in a state of rekeying during W_{c+1} (i.e. $E(T_E) + E(T_C)$) can be expressed as $[(1 - p_{c_i}) * W_{c+1} * N_i]$. Since we know $E(T_E)$ from equation (7), we can compute $E(T_C)$ using the following formula:

$$\begin{aligned} E(T_C) &= [(1 - p_{c_i}) * W_{c+1} * N_i] - E(T_E) \\ &= (1 - 2p_{c_i}) * W_{c+1} * N_i \qquad (8) \end{aligned}$$

Standard KM schemes are not supported by replication, so the overhead is given by: $O = E(T_R) + E(T_E)$ in which case $\alpha_i = 1 - (2 * p_{c_i})$. On the other hand in our proposed SPKM framework, overhead is calculated using equation (5). By substituting equations (6),(7) and (8) into equation (4), we can re-express availability as follows:

$$\alpha_i = |1 - (N_i + p_{c_i} - p_{c_i} N_i)| \qquad (9)$$

The growth of the number of replicas/keys is controlled with a heuristic that bounds the availability degree by 1, and positive values of availability are ensured by expressing the results of equation (9) as absolute values.

If a rekey request arrives when there is no existing backup copy available to satisfy it, then the SPKM scheme reverts to generating a new key, reencrypting the primary copy with the new key and distributing the new key to the users that are left in the group. So in the worst case, the cost of rekeying reverts to the cost of KM in a standard scheme.

3.3. An Example

We use an example of a simple read-intensive scenario to explain how our framework operates. In this case, suppose that the observations of the key server during the initial monitoring period W_1 result in a prediction that one rekey request is going to arrive from U_2 during a future monitoring period W_x. In order to handle the rekey request, the analysis from the FBCL at the key server determines that one backup key and replica needs to be generated in anticipation of this request. As shown in Figure 5 the key server creates a new backup key K_2, for the group U_2, and transmits this key to the data server where it is kept in a secret registry. On reception of the new backup key K_B and instructions to replicate D_{K_2}, the data server proceeds to create a new copy of D_{K_2} that it reencrypts with K_B. In order to maintain copy consistency, updates on D_{K_2} are checkpointed onto D_{K_B} by periodically replicating D_{K_2} and reencrypting it with K_B to obtain an updated version of D_{K_B}.

As shown in Figure 6, when the key server receives a departure request during W_x it proceeds to destroy the primary copy and assign K_2 the value of K_B. The primary is replaced

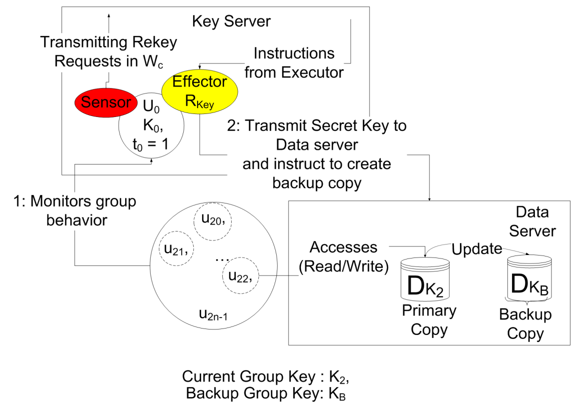

Figure 5. Initial Replacement Scenario with no Update Requests

with the next backup copy in line (in this case $D_{K_2} \leftarrow D_{K_B}$) and the new key $K_2 \leftarrow K_B$ is broadcast to the users remaining in the group. Finally the key server creates a new backup

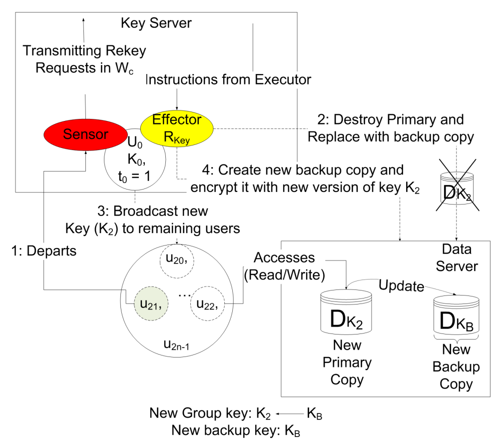

Figure 6. Scenario in which u_{21} departs

key, K_B, that it transmits to the data server and as before the data server creates a new backup copy D_{K_B} in anticipation of future requests.

4. Prototype and Performance Evaluation

This section presents the prototype implementation of our proposed framework and experimental results evaluating its performance with respect to a basic key management (BKM) scheme that is not supported by the paradigm of autonomic computing. We implemented the prototype as though the access control hierarchy were comprised of only one single class. This simplifies the evaluation process since handling several nodes in the hierarchy requires a scheduling algorithm that determines a priority for satisfying requests in a way that minimizes the overall cost of replication and rekeying. A single node still allows us to evaluate the impact of autonomic control on KM.

We evaluate the performance and scalability of the SPKM framework proposed in this paper with a set of experiments conducted on an IBM Pentium IV computer with an Intel 3.00GHz processor and 1GB of RAM. Our evaluation is conducted on a write-intensive file to simulate a scenario in which re-encryption for data security is necessary. Therefore, we do not compare our approach with the lazy re-encryption technique which, as we mentioned before, is better suited to read-intensive scenarios. Our performance evaluation uses the following metrics - the response time, cost of message communications (update costs), percentage of requests satisfied, cost of replication, and the size of the window of vulnerability. The experiments are not exhaustive, but give an intuition about the general performance of the schemes. Results for each case are obtained from averages over 10 runs, with random numbers of rekey requests expressed as proportions of a user group with a maximum of 100 members and files (primary and backup copies of shared data) of size $\approx 32MB$.

4.1. Prototype Description

Our prototype is built on the Microsoft Windows XP platform using the Java 2 Standard Development Kit and Eclipse [27, 28]. The prototype is designed in the form of a chat system using socket programming and a client-server model. In the access control class, the clients play the role of users and the server that of the key and data server, supplying both keys and allowing access to data. In out prototype, the server generates Triple DES (Data Encryption Standard) encryption/decryption keys and keeps an encrypted log file of the group's communications.

The clients (users) communicate via the server and all communications are saved into a log file. Access to the log file is granted only if a user holds the correct group key. Once the server is initialized it waits until it receives a connection request at which point it checks to ensure that it has a free socket. If this is the case, it spawns a thread that allows the client to connect to it. The server then displays the current group key on the client's message board as well as the communications that have occurred since the client joined the group. A client disconnects from the group by transmitting a 'BYE' message to the server. On reception of the 'BYE' message, the server closes its connection to the user from whom the message was emitted and broadcasts a message, indicating the departure event as well as the updated group key, to the other group members.

The prototype starts off the adaptive model by using the initial connection and disconnection requests to collect data empirically. This data is stored in a file that serves as the knowledge base for the server and the server uses this data to start running the FBCL that performs adaptive KM.

4.2. Experiments

In the first experiment we evaluate the comparative response times per request of the BKM and SPKM schemes with respect to the rekey request arrival rate. Response time is the time the server takes to generate a new key, re-encrypt the data and transmit the key to the users left in the group. The size of the monitoring window is set statically to a value of 60 seconds. The sum of rekey requests is computed by adding the number of requests that arrive during the monitoring period and computing the arrival rate using equation (1). The experiment was repeated 10 times for each case with an average interval of 2 to 5 seconds between each request, and the results averaged and plotted in Figure 7. The error bound for each point plotted is ± 4 seconds in the BKM scheme and ± 1 seconds in the SPKM scheme. The response time in the

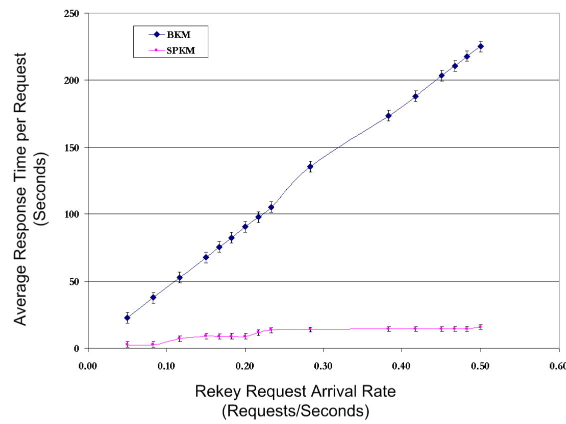

Figure 7. Average Request Satisfaction Time

BKM scheme grows linearly with an increasing arrival rate of rekey requests. By contrast, in the SPKM scheme, little or no additional time is required to handle an increasing arrival rate. This is an indication that the response time in the BKM scheme is affected by an increased arrival rate of rekey requests due to the fact that the server is interrupted every time a rekey request occurs. Each interruption requires restarting the key generation and re-encryption process so, in cases of high rekey request arrival rates, the server takes longer, on average, to respond to a rekey request. By contrast, in the SPKM scheme, since the key server replicates the primary copy of the file and creates supporting keys, on average, response time is equivalent to the time it takes to transmit the new key. Moreover, the SPKM scheme further minimizes its key replacement costs by adaptively adjusting the number of replicas in response to the arrival rate of rekey requests.

Our second experiment evaluates the size of the window of vulnerability created in both the BKM and SPKM schemes during rekeying. The window of vulnerability is a sum of the time it takes a user to commuicate a departure request to the server and the response time. We ran each experiment 10 times, each time over a 60 second time window and the results were averaged and plotted in Figure 8. The error bound for each of the plotted bars is ± 0.5 seconds. We noted that the

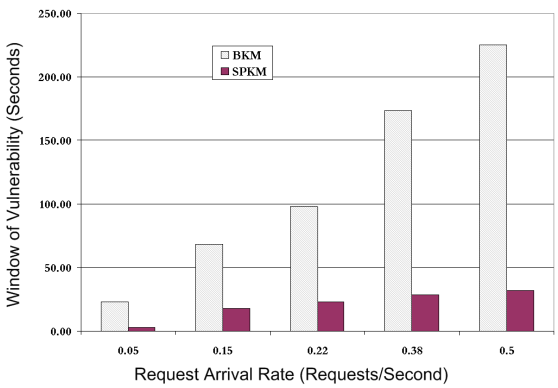

Figure 8. Variation in Vulnerability Window Size with Respect to Request Arrival Rate

size of the vulnerability window in the BKM scheme grows linearly with an increasing arrival rate of rekey requests. By contrast, in the SPKM scheme, little or no additional time is required to handle an increasing number of rekey requests. This supports the result we obtained in the first experiment where an accumulation of rekey requests in the basic scheme results in a longer average response time per request. We note also that the SPKM scheme not only overcomes the drawback of delayed response times in the BKM scheme but also makes for better security by reducing the size of the window of vulnerability between rekey requests.

Finally, we discuss the processing cost incurred by the server. We noted that for an average rekey request arrival rate of 0.256 requests/second the SPKM scheme uses an average of four replicas against one in the BKM scheme. The average time spent updating the replicas is 0.17 seconds in the BKM scheme as opposed 6.6 seconds in the SPKM scheme. However, the SPKM scheme makes up for its shortcomings on the replication and update side by satisfying an average of 52.27% of the requests during the 60 second montoring interval while the basic scheme only satisfies 20.96%. In fact in our experiments when the arrival rate was 0.5 requests/second during the 60 second monitoring interval, the basic scheme satisfied less than 1% of the requests it received while the adaptive scheme initially, (i.e. before adjusting to the new rate) satisfied 12.67% of the requests. Table 1 summarizes the results.

Table 1. KM Schemes: Comparison

	BKM	SPKM
Replication	1	4
Update Costs (seconds)	0.17	6.6
% Requests Satisfied	20.96	52.27

5. Conclusions

In the preceding sections we outlined some of the reasons behind the hesitancy to adopt the autonomic computing paradigm into security frameworks. For reasons pertaining to cost and credibility, business owners prefer to have control over their security mechanisms. Our aim therefore, was to argue that self-protecting approaches can enhance the performance of standard KM schemes without necessarily changing their underlying specifications and make the job of the SA easier.

We considered the problem that arises in shared data scenarios where access is controlled with a CKM scheme. In these scenarios, several users hold a secret key that is used to encrypt commonly accessible data. When group membership changes, data security is maintained by updating the shared group key and transmitting the updated key to the users remaining in the group. Hence, KM is expensive when changes occur frequently and involve large amounts of data.

In order to address this problem we proposed a simple but effective SPKM framework based on the autonomic computing paradigm. The framework enhances the capabilities of a standard CKM scheme with a combination of a stochastic model and replication. The stochastic model determines an acceptable degree of replication to maintain based on an observed arrival rate and the potential impact of checkpointing on the overall performance of the system. Backup replicas and keys are generated to preemptively handle situations of high demand making for better performance than in standard schemes. The SA now only has to preset required parameters and let the system run, without having to be present to manually handle every change. Experimental results show that, in comparison to the BKM approach, the SPKM approach reduces the vulnerability window and response time while increasing data availability.

Regarding the security of the scheme, if we assume that a key generated using the Triple DES (Data Encryption Standard) scheme is secure then it is safe to say that both the BKM and SPKM schemes are secure in the sense that the SPKM scheme only enhances the performance of the BKM scheme by adding in data replication. Checkpointing on backup replicas is secured by ensuring that updates are accepted only from the primary replica. Rekeying results in a destruction of the primary replica associated with the updated key and the selection of a new primary copy by the server. Potential applications of this model, outside the realm of security, include replication for fault tolerance and performance adjustments to meet quality of service demands in Web-based environments.

Future work will involve further implementation and experimentation throughout the entire hierarchy aimed at evaluating the performance of the SPKM approach against the BKM approach. Other challenges include finding other statistical distributions that are more effective than the Poisson model in modeling rekey arrival rates, and finding a good way to define adequate monitoring thresholds. We also need to find a better prediction model for computing arrival rates. An example would be to compute a moving average as opposed to using the maximum arrival rate. Issues of copy consistency can also arise in situations where there is a high volume of communications between users (frequent updates on the primary copy) and rekey requests occur within very short intervals of each other.

References

[1] S. G. Akl and P. D. Taylor. Cryptographic solution to a problem of access control in a hierarchy. *ACM Transactions on Computer Systems*, 1(3):239–248, August 1983.

[2] M. Atallah, K. Frikken, and M. Blanton. Dynamic and efficient key management for access hierarchies. *In Proceedings, ACM Conference on Computer and Communications Security*, pages 190–202, 2005.

[3] G. Ateniese, A. De Santis, A. Ferrara, and B. Masucci. Provably-secure time-bound hierarchical key assignment schemes. *In Proceedings of 13th ACM Conference on Computer and Communications Security (CCS'06)*, pages 288–297, 2006.

[4] G. Ateniese, K. Fu, M. Green, and S. Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. *ACM Transactions on Information and System Security (TISSEC)*, 9(1):1–30, February 2006.

[5] M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic and atomic proxy cryptography. *In Proceedings of Eurocrypt '98*, 1403:127–144, 1998.

[6] D. Chess, C. Palmer, and S. White. Security in an autonomic computing environment. *IBM Systems Journal*, 42(1):107–118, 2003.

[7] H.-Y. Chien. Efficient time-bound hierarchical key assignment scheme. *In Proceedings. IEEE Transactions on Knowledge and Data Engineering*, 16(10):1301–1304, 2004.

[8] J. Crampton. Cryptographically-enforced hierarchical access control with multiple keys. *In Proceedings of 12th Nordic Workshop on Secure IT Systems (NordSec 2007)*, pages 49–60, 2007.

[9] J. Crampton, K. Martin, and P. Wild. On key assignment for hierarchical access control. *In Proceedings, 19th IEEE Workshop on Computer Security Foundations, S. Servolo Island, Italy*, pages 98–111, 2006.

[10] M. Das, A. Saxena, V. Gulati, and D. Phutak. Hierarchical key management scheme using polynomial interpolation. *SIGOPS Oper. Syst. Rev.*, 39(1):40–47, 2005.

[11] A. De Santis, A. Ferrara and B. Masucci. New constructions for provably-secure time-bound hierarchical key assignment schemes. *In Proceedings of 12th ACM Symposium on Access Control Models and Technologies (SACMAT'07)*, pages 133–138, 2007.

[12] M. H. DeGroot and M. J. Schervish. *Probability and Statistics*. Addison Wesley, Third Ed., New York, 2002.

[13] R. Hassen, A. Bouabaallah, H. Bettahar, and Y. Challal. Key management for content access control in a hierarchy. *Computer Networks*, 1(51):3197–3219, 2007.

[14] P. Jalote. *Fault Tolerance in Distributed Systems*. Pearson Education: Prentice Hall, NJ, 1998.

[15] S. Johnston, R. Sterritt, E. Hanna, and P. O'Hagan. Reflex autonomicity in an agent-based security system:: The autonomic access control system. *In Proceedings, 4th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EaSe'07)*, pages 68–78, 2007.

[16] A. Kayem, S. Akl, and P. Martin. On replacing cryptographic keys in hierarchical key management schemes. *Journal of Computer Security*, 16(3):289–309, 2008.

[17] A. Kayem, P. Martin, and S. Akl. Heuristics for improving cryptographic key assignment in a hierarchy. *In Proceedings,* 3^{rd} *IEEE Int'l Symposium on Scecurity in Networks and Distributed Systems*, pages 531–536, 2007.

[18] J. Kephart. Research challenges of autonomic computing. *In Proceedings of 27th International Conference on Software engineering, St. Louis, MO, USA*, pages 15–22, 2005.

[19] J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[20] S. J. Mackinnon, P. D. Taylor, H. Meijer, and S. G. Akl. An optimal algorithm for assigning cryptographic keys to control access in a hierarchy. *IEEE Transactions on Computers*, c-34(9):797–802, September 1985.

[21] M. Mat Deris, J. Abawaly, and A. Mamat. An efficient replicated data access approach for large-scale distributed systems. *Future Generation Computer Systems*, (In Press.) 2007.

[22] M. Mat Deris, J. Abawaly, and A. Mamat. Rwar: A resilient window-consistent asynchronous replication protocol. *In Proc.* 2^{nd} *Int'l Conf. on Availability, Reliability and Security*, pages 499–505, 10-13 April 2007.

[23] A. Moreno, D. Sanchez, and D. Isern. Security measures in a medical multi-agent system. *Frontiers in Artificial Intelligence and Applications*, 100:244–255, 2003.

[24] N. Post. Credit card information stolen from winners. *http://www.canada.com/nationalpost/story.html*, Jan. 2007.

[25] I. Ray and N. Narasimhamurthi. A cryptographic solution to implement access control in a hierarchy and more. *In Proceedings. 7th ACM Symposium on Access Control Models and Technologies, Monterey, CA*, pages 65–73, 2002.

[26] V. Shen, T. Chen, and F. Lai. A novel key management scheme based on discrete logarithms and polynomial interpolations. *Computers and Security*, 21(2):164–171, 2002.

[27] E. Software. Eclipse. *http://www.eclipse.org/*, Jan. 2008.

[28] TechRepublic. Java (sun),http://software.techrepublic.com./. Sept. 2007.

[29] S.-Y. Wang and C.-S. Laih. Merging: An efficient solution for time-bound hierarchical key assignment scheme. *IEEE Transactions on Dependable and Secure Computing*, 3(1):91–100, 2006.

[30] C. Yang and C. Li. Access control in a hierarchy using one-way functions. *Elsevier: Computers and Security*, 23:659–644, 2004.

[31] X. Yi. Security of chien's efficient time-bound hierarchical key assignment scheme. *IEEE Transactions on Knowledge and Data Engineering*, 17(9):1298–1299, September 2005.

[32] X. Yi and Y. Ye. Security of tzeng's time-bound key assignment scheme for access control in a hierarchy. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):1054–1055, January 2003.

[33] S. Zhong. A practical key management scheme for access control in a user hierarchy. *Computers and Security*, pages 750–759, 2002.