A Security Mechanism For Component-Based Systems

Mark Grechanik, Dewayne E. Perry, and Don Batory

The University of Texas at Austin
Austin, Texas 78712

{gmark,batory}@cs.utexas.edu, perry@ece.utexas.edu

Abstract. Security, scalability, and performance are critical for large-scale component-based applications. Weaving security solutions into the fabric of component-based architectures often worsens the scalability and performance of the resulting system. In this paper we analyze the sources of nonscalability and conduct an empirical study that shows that close to 80% of interactions between components and their clients in different commercial systems occur within protected security boundaries. Based on these findings we propose a novel scalable security mechanism for component-based systems called *Component Adaptive Scalable Secure Infrastructure Architecture (CASSIA)*. CASSIA utilizes the topology of the security boundaries and patterns of interactions among components to achieve noticeable improvements in scalability and performance for component-based applications. We conduct a case study that confirms the scalability of CASSIA, and propose a *Secure COmponent Protocol (SCOP)* that incorporates our mechanism into a component infrastructure.

1 Introduction

Component-based applications are ubiquitous. Many software vendors provide various components (many of them for free) whose use reduces the cost and time required for development of commercial products. For example, Microsoft Windows comes with hundreds of generic components ranging from different GUI elements and FTP clients to the sophisticated Internet Explorer browser control. Providing effective security of component-based systems is widely recognized as a crucial problem in *Component-Based Software Engineering (CBSE)*, and it has a major impact on the overall quality of component-based applications [1][2].

No less than security, performance and scalability are also critical for *large-scale component-based (LSCB)* applications. Inadequate architecture, a bad choice of *Commercial-Off-The-Shelf (COTS)* components or a combination of both prevent applications from achieving their performance goals. Performance and scalability are key challenges in building LSCB applications [3] since their predictability is inherently difficult. Weaving security solutions into the fabric of component-based architectures often worsens the performance and scalability of the resulting system. According to Clements, performance is a function of the frequency and nature of inter-component communication, in addition to the performance characteristics of the components themselves [4][5]. It is the nature of inter-component communications that we address in this paper.

We propose a scalable security mechanism for LSCB systems called *Component Adaptive Scalable Secure Infrastructure Architecture (CASSIA)*. CASSIA utilizes information about the topology of the security boundaries and the topology of interactions among components via their infrastructure. By incorporating information about security boundaries into a component infrastructure we eliminate unnecessary encryption and decryption of messages when communicating components are located within the same security boundary. In addition, we eliminate the conflict between the need to partition the environment for some applications and the needs for global secure partitions. We show the results of an empirical study that suggests interfaces of over 80% of the components in real-world component-based systems are used by clients within a protected security perimeter. We use this result to show that the time and space complexities of our solution are $O(n)$ and therefore, it is scalable. We implement our solution for MicoCCM, an open-source component infrastructure and run performance experiments the results of which prove that our solution enables a scalable and secure mechanism for LSCB systems.

As part of our solution we present a *Secure COmponent Protocol (SCOP)*. SCOP is based on the S-CODEP protocol [6] which itself is based on the Kerberos protocol [7][8]. We prove the soundness of SCOP in a separate technical report [9] using BAN authentication logic [10]. *The contribution of this paper is 1) analyzing the sources of nonscalability when using security solutions in LSCB systems, 2) conducting an empirical study that reveals 80% of interactions between components and their clients occur within the same protected security perimeter, 3) proposing and implementing the scalable security mechanism CASSIA and evaluating its performance, and 4) proposing a protocol SCOP that incorporates our scalable security mechanism into a component infrastructure.*

2 Problem Statement

Enterprises partition their computing environments based on different criteria to secure the flow of information. For example, many enterprises use *Virtual Private Networks (VPNs)* as a global solution to securely partition their computing environments [11]. Partitioning of the enterprise environment is often based on geographic locations of departments or the security policy regulating the flow of information between different departments. VPN uses a protocol called IPSec that encrypts application messages on the IP level [40]. In general, security mechanisms like VPN enable unencrypted communications between applications within the same security partition.

Different component-based applications often require additional security partitionings that may different from existing security divisions of the enterprise infrastructures. A situation often arises when a company purchases and installs two or more applications produced by competing companies within a single security partition and on the same computer. As it often happens these applications use the same generic components, and it is possible for these applications to interfere with one another by means of using these generic components. Competing companies often demand that their applications should run in their own security partitions, so that other competing applications cannot eavesdrop on them to understand their interworkings. Since installation and configuration of a VPN is a serious technical exercise, and once installed it is rarely reconfigured [11], reconfiguring VPN every time a different application is installed or run is impractical. When a component-based application should be partitioned within a single computer, VPNs and similar security systems cannot even be used.

As a result of this condition programmers add separate security mechanisms (e.g., encryption) to existing applications, and it leads to worsened performance and scalability of the resulting system. Encrypting and decrypting communications even for small-size computing systems results in a significant performance drawback [12]. A fundamental problem of applying standard security mechanisms based on encrypting algorithms to LSCB systems is that the resulting secure systems are non-scalable. Existing research is limited to offering protocols and algorithms that address different security aspects of software systems without paying much of attention to the integration problems of security mechanisms with large-scale software.

We address a problem of adapting security partitions to business requirements without affecting the scalability and while preserving security properties of the resulting LSCB system. The goal is to provide a security mechanism that enables users to change partitions dictated by the business needs dynamically and securely.

3 Background

3.1 Software Fortresses

The concept of software fortresses is an approach for modeling large enterprise systems as sets of self-contained entities [13]. Each entity is called a *fortress*, and it makes its own decisions about the underlying platform, data storage, and security mechanisms and policies. Fortresses communicate with one another through carefully designed mechanisms. A software fortress architecture is an enterprise architecture consisting of a series of self-contained, mutually suspicious, marginally cooperating software fortresses interacting through carefully crafted and meticulously managed treaty relationships. Once a component is allowed inside a fortress it gains access to all other components within that fortress. The walls of the fortress allow only those communication requests that come into the fortress through approved channels called *drawbridges*. The walls of a fortress are collectively called a *security perimeter* of the system inside a fortress. Only approved communications are accepted inside the fortress. *Treaties* are the formal agreements between fortresses that define how fortresses work together. The trust rule of a software fortress specifies that every entity inside a fortress trusts every other entity inside the same fortress while trusting no entity outside the fortress. *All communications that originated inside a fortress and are designated to entities located inside the same fortress are approved by default*.

The scalability of software fortresses is based on three ideas:

- the cost to process a message at the security perimeter of a fortress is much smaller than then total cost of the processing of this message;
- processing rate is increased by scaling up (i.e. by replacing the hardware on which the fortress is based with more powerful hardware) and by scaling out (i.e. by adding more hardware), and
- the cost of a unit of work remains constant as the throughput increases.

In this paper we assume that component-based systems are built as software fortresses.

3.2 Related Work

While the areas of performance of component-based systems and software security are, each on its own, rife with notable publications their intersection yields few results. It is widely recognized that no single technique can produce completely trusted components [14][16]. Yet it is not clear how to weave security solutions into the fabric of component-based architectures that will not lead to the decreased performance of the resulting system.

The *Trusted Components Initiative (TCI)* is a cooperative effort to provide the software industry with methods, techniques and tools for building high-quality reusable components, thereby elevating the general level of trust that software users, and society at large, can have in these components [17]. TCI's web site contains a page with references to the growing number of publications in this area, however, the lack of publications that discuss the interaction between security and performance and scalability solutions is glaring.

One of main research directions in security solutions for component-based applications is in providing strong authentication control of access to and from components [18][19]. A number of techniques and algorithms are designed to prevent malicious components to gain access to computers inside secured networks, or to prevent components to access domains for which they do not have a proper authorization. However, these solutions fall short to solve the problem that we pose in this paper since they are concerned with solving security problems rather than addressing the issues of performance and scalability as functions of security solutions.

Various component infrastructures use different security standards. CORBA and CCM implementations may use security services defined by the Object Management Group's standards [21] while DCOM/.Net is based on a different standard [22]. *Authorization Token Layer Acquisition Service (ATLAS)* describes the service needed to acquire authorization tokens to access a target system using the CSIv2 protocol. The *Common Secure Interoperability Specification, Version 2 (CSIv2)* defines the Security Attribute Service that enables interoperable authentication, delegation, and privileges. CORBA Security Service provides a security architecture that can support a variety of security policies to meet different needs. The security functionality defined by this specification comprises a variety of mechanisms such as identification and authentication of principals and authorization and infrastructure-based access control [23]. However, the effect of these security standards on the scalability and performance of component-based applications is not yet fully investigated.

When considering LSCB applications, global optimizations may not always be desirable. Evaluating an overall application, potentially consisting of hundreds of components, whenever an individual component or a group of components does not behave as expected, might induce unnecessary overhead and not scale well. Techniques for monitoring the performance of component-based applications and distributing the adaptation mechanism based on the results of monitoring was proposed in [24][25]. If a problem is detected at an individual component level, then this problem is dealt with locally.

A design for adaptive self-optimizing containers for *Enterprise JavaBeans (EJB)* is offered in [26]. An adaptive container performs the discovery of inter-component communication patterns at runtime, analyzes them, and adapts itself to the needs of component-based applications by refactoring its structure. The adaptation process is based on the acquisition of knowledge about call patterns between components of these applications. While there are indications that component-based applications benefit from this approach, it is targeted specifically for EJB platforms and does not address the impact of security on scalability and performance.

A different research direction emphasizes an idea that scalability and performance of distributed and component-based systems can be predicted using traditional performance analysis methods [27][28][29][30][31]. In general, an architectural design in a form of *Unified Modeling Language (UML)* is translated to models for performance analysis based on layered queuing networks or stochastic Petri nets. These models are useful to specify architectural characteristics and capture different performance parameters. However, they often ignore important details of the runtime environment, and as a result the problem of predicting and improving performance and scalability of LSCB applications is still not adequately solved.

In a similar vein, there are attempts to predict performance and scalability of large-scale enterprise component-based applications before they are built [32][33]. This research is based on the assumption that all components in an application are developed from scratch by the same organization that has a clear vision about its evolution. This assumption is quite far from reality. Issues of security, scalability, and performance conflict with intentions of software engineers at design time to view their systems as a set of reusable and composable components that can be moved to different platforms easily without affecting the quality of the design. It means that engineers try to design a complete system first, and then retrofit security and scalability into the design, that is notoriously difficult to do.

Finally, few papers present results on measuring performance of component-based systems [34][35]. An interesting result shown by [34] states that better performance can be obtained by sacrificing modularity and reusability. While we utilize some of the techniques presented in these papers for our case study, our approach is based on preserving good characteristics of component-based systems.

4 Empirical Study

In order to gain insight into possible means by which a scalable security mechanism can be enabled, we analyze interactions among components in large-scale commercial component-based software and identify two important characteristics: clustering and linearity.

Clustering. Our hypothesis is that in component-based systems a large percentage of components service requests of clients within the same security perimeter. This hypothesis is based on multiple observations: GUI components are almost always used within the same security perimeter, and for performance reasons many software engineers prefer to put components and their clients on the same computers within the same security perimeter unless they absolutely have to separate them.

The results of our study of seven companies are shown in Table 1. We analyzed the source code for medium to large-scale projects that were developed in seven different companies by different groups of software engineers. The projects were not dependent on one another in any way. The nature of systems ranged from a web-based online bidding system developed by Ambac-Connect Inc. to a sophisticated wireless workforce management system developed by Arrowsmith Technologies, Inc. In each project we identified distinct components either developed as part of the project or COTS purchased from a third party or available as part of other software packages used in the project. For each distinct component we identified its client components. From the available source code and project documentation we partitioned the project by security perimeters identifying each software fortress. The interactions between components are represented by channels or edges in Figure 1. Then we calculated the percentage of the channels that do not cross any security perimeter. A location in the source code where a distinct invocation of a component's method occurred was counted as one client-component interaction (channel). As shown in Table 1 this percentage is high and close to 80% on average and this experimental result effectively confirms our hypothesis.

Linearity. We noticed that some components interact with few clients while other components service a larger number of clients. This uneven distribution of load among components prompted us to ask whether there is a correlation between the components with a bigger number of channels and the percentage of these channels that do not cross any security perimeters. Figure 1 shows the percentage of channels that do not cross any security perimeters. From this graph we make an important observation that the greater the number of channels, the more likely that a smaller percentage of them will cross security boundaries. Let variables X and Y stand for samples of the number of channels and the percentage of these channels that do not cross any security perimeters respectively. To estimate the correlation between samples of the graph we compute the Pearson product moment correlation coefficient [36] as

$$r = \frac{n\Sigma(XY) - (\Sigma X)(\Sigma Y)}{\sqrt{n\Sigma Y^2 - (\Sigma Y)^2}\sqrt{n\Sigma X^2 - (\Sigma X)^2}}$$

Company	Project	Total Components	% of components interacting within the same security perimeter
IBM	HelpNow!	153	87
KLA-Tencor	Archer Analyzer	23	92
Boundless	Viewpoint Administrator	56	98
Ambac-Connect	Online bidding system	72	75
Arrowsmith Technologies	FleetCon	39	70
Globeset	Secure Wallet	167	83
Schlumberger	Smartcard Management System	21	95

TABLE 1. Percentage of components used in commercial projects that interact within the same security perimeter.

The value of `r` for the data shown in Figure 1 is `0.747` that suggests a strong tendency for components to interact with larger percentage of their clients within the same security perimeters as the number of their clients grows.

5 A High Probability Bound

We use the analogy between an event of flipping a coin and a random selection of an LSCB system. Even though each LSCB system is highly structured internally, the multiplicity of factors which affect the outcome of the process of building these systems makes it easier to reason about a pattern that emerges from a large number of LSCB systems built in different environments in probabilistic terms. The pattern is that the larger an LSCB system is in terms of the number of components it contains, the less likely it is that the majority of channels between its components cross the security boundaries established within that system.

There are many ways to partition the clique architecture into fortresses (specifically, 2^n for a clique of `n` components). Some ways of partitioning result in the majority of channels crossing security boundaries, while the other ways lead to the opposite result. When considering real-world commercial LSCB systems we observed a stable pattern that on the average 80% of channels in these systems do not cross any security boundaries independently of the domain of the system or its architectural style. This number was smaller for smaller systems and the largest system we considered had less than 200 components. Our intention is to extrapolate our empirical findings to predict how likely it is for channels in LSCB systems to cross security boundaries as the number of these channels grows. We assume for our analysis that the number of channels grows proportionally to the number of components that communicate through these channels.

Consider a population of randomly selected samples of real-world component-based systems by approximating fortress architectures to be random with respect to the underlying component architectures. Our analysis uses indicator random variable X_{ij} = {there is a channel between component i and component j}. Let X_{ij} be associated with the event in which the channel crosses one or more security partitions:

$$X_{ij} = \begin{cases} 0, SP_i \equiv SP_j \\ 1, SP_i \neq SP_j \end{cases}$$

where SP_k is a security partition (fortress) in which component k resides. The expected value of random variable X_{ij} is $E[X_{ij}] = 0.2 \cdot 1 + 0.8 \cdot 0 = 0.2$, where 0.2 and 0.8 are the probabilities whether a channel crosses security boundaries or not.

Let $X = \sum_{j=1}^{n} \sum_{i=1}^{n} X_{ij}$. Then by the theorem of linearity of expectations is

$$E[X] = \sum_{j=1}^{n} \sum_{i=1}^{n} E[X_{ij}] = 0.2 \cdot (n \cdot (n-1))/2 = 0.1 \cdot n \cdot (n-1)$$

In our empirical study we work with seven commercial LSCB system whose size is up to 200 components. While it is enough to detect the pattern of distribution of channels with respect to fortress boundaries, we do not have sufficient experimental data to prove that this pattern holds for all systems with very large numbers of components. However, we can derive the probability that this pattern will not hold for all LSCB systems that contain any number of components. To compute this probability we use the *Chernoff bound* [37] stating that $Pr(X \geq \beta E[X]) \leq \exp((1-(\ln\beta + 1/\beta))\beta E[X])$, where $\beta > 1$. By taking $\beta = 1.5$ (which adjusts for the expectation from 80/20 to 70/30 probability distributions) we derive the probability for 30% of n components to have channels that cross security boundaries for an arbitrary component-based system to be less than $\exp(-(0.011 \cdot n \cdot (n-1)))$. It means that while for a system consisting of ten components the probability that more than 30% of these components communicating with one another are located within different security partitions is close to 0.4 (i.e. the probability is relatively high), this probability for a system consisting of 100 components is close to $5.1 \cdot 10^{-43}$ (obviously, extremely small).

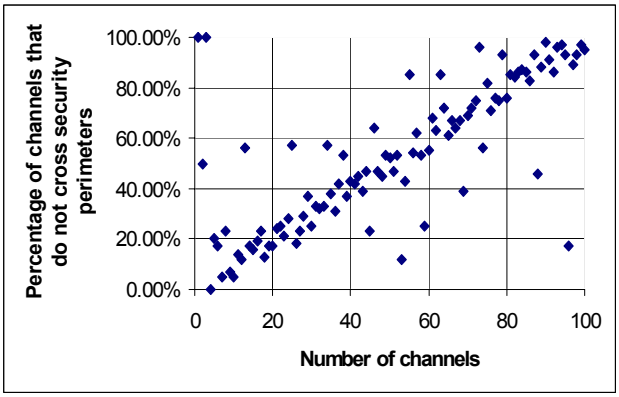

Figure 1: The percentage of channels between components that do not cross security perimeters.

6 Our Solution

CASSIA exploits the nature of interactions among components and their component infrastructure. Since clients invoke methods of interfaces of components via dispatch[1], invocation requests are forwarded to CASSIA that decides whether to use encryption/decryption services. The decision is made adaptively based on the analysis of security descriptors of components and software fortresses. We merge the component infrastructure with the authentication server and offer SCOP, a scalable secure protocol for component-based systems. We have proved its correctness (where the proof is published in a separate technical report [9]) using BAN authentication logic.

6.1 The Nature of Interactions

Even though it is generally thought that a client invokes a method of an interface of some component directly, in fact this request is executed by a component infrastructure that serves as a request broker between components and their clients. Often clients and components are located in different processes and therefore invocation requests are resolved via the late binding or dispatch. The nature of interactions between components is that they are accomplished to a large degree by a component infrastructure. Component infrastructures expose dispatch APIs that clients use to request services from components. Hiding these dispatch APIs in programming languages enables programmers to abstract away the complexity associated with using component technologies, and view component-based architectures as cliques where nodes are components and their clients and edges are the communication channels between them. However, to reflect the nature of interactions among

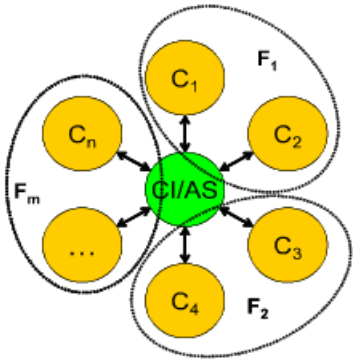

Figure 2: A star topology of interactions between component and their infrastructure.

1. Dispatch is an mechanism used to resolve references to methods that should be invoked in response to client requests

components correctly we use a different representation of component interactions as shown in Figure 2.

In this representation we explicitly show that no two components interact with each other directly. The central node marked as CI/AS is the component infrastructure that we merge with the authentication server. This infrastructure is responsible for taking requests from clients, locating the desired component, loading it in memory, instantiating its interfaces, invoking its methods, and returning the results to the clients. In the worst case, each component may be placed in a separate fortress, and then all interactions between components and their clients should be encrypted. However, by imposing the fortress metaphor we effectively partition the system by imposing security boundaries than encompass many components. If we designate the number of components as n_c and the number of fortresses as n_f, then we have that $n_c >> n_f$. Since encryption is required only when we cross the security perimeter of fortresses, the overall complexity of a security solution based on the number of fortresses may be reduced. By merging an authentication server with a component infrastructure and utilizing the topology of communications among components we reduce the number of shared keys from $O(n^2)$ to $O(n)$, one for each component (or strictly speaking one key for each fortress).

6.2 CASSIA

A central idea of our solution is if two communicating components are located within the same security perimeter, then no encryption and decryption of communications are necessary. It means that component infrastructures should be aware of the configuration of fortresses and components located inside them.

6.2.1 Configuration

An example fragment of a configuration XML-based file for CASSIA is shown in Figure 3. Each fortress entry has a name designator and comprises components located within the fortress security perimeter. A treaty entry specifies fortresses for which it is active and a protocol used for encryption/decryption of communications between the fortresses. This configuration file is created by a component infrastructure that queries the metadata about fortresses and components they comprise. The information is used in the CASSIA adaptive algorithm described in the next section.

6.2.2 Algorithm

Pseudocode for the CASSIA adaptive algorithm is shown in Figure 4. We introduce functions $\texttt{Fortress}: C_m \rightarrow F_n$, $\texttt{Treaty}: F_k \times F_l \rightarrow T_{kl}$, and $\texttt{Protocol}: T_{pq} \rightarrow P_{pq}$.

```
<fortress name="A">
     <component ID="1"/>
     <component ID="2"/>
</fortress>
<fortress name="B">
     <component ID="3"/>
</fortress>
<treaty>
     <fortress name="A"/>
     <fortress name="B"/>
     <protocol>OpenSSL<protocol/>
</treaty>
```

Figure 3: An example fragment of XML-based configuration file for CASSIA.

Function `Fortress` maps elements of a component domain to elements of the set of fortresses. Function `Treaty` maps elements of the domain of fortresses to the elements of the set of treaties between fortresses. Finally, Function `Protocol` maps elements of the domain of treaties to the elements of the set of cryptographic protocols.

The algorithm works as follows. Communicating components C_i and C_j are the first two parameters of the algorithm, and the third parameter is boolean flag variable `encrypt` describing whether communication data between components are encrypted or decrypted. For both components C_i and C_j we obtain the corresponding fortresses F_i and F_j to which they belong by applying function `Fortress`. If fortresses F_i and F_j are the same then we do not apply cryptography for communications between these components. Otherwise, we obtained the descriptor of treaty T_{ij} by applying function `Treaty` to fortress descriptors F_i and F_j. From the descriptor of treaty T_{ij} we obtain the protocol P_{ij} that is used to encrypt/decrypt communication by applying function `Protocol`.

6.2.3 Implementation

We implemented the CASSIA adaptive algorithm for *Mico CORBA Component Model (CCM)* [38][39]. The acronym MICO recursively expands to MICO Is CORBA. Mico is a freely available OpenSource project and fully compliant implementation of the CORBA standard. MICO has been branded as CORBA compliant by the OpenGroup, and it is considered to be an industrial strength software since it is used and supported by different Fortune 1000 and smaller companies around the world. The sources of MICO are placed under the GNU-copyright notice.

OpenSSL is used as a low-level cryptographic protocol to encrypt/decrypt communications between components and their clients. We maintain a configuration table within Mico CCM in an XML format that is a representation of configuration data shown in Figure 3. Rather than implementing security perimeters based on physical entities (e.g. firewalls) we intro-

duce a logical partition by grouping components in the configuration file based on given fortresses. Keys are located in a protected storage and maintained using the SCOP protocol that is the subject of the next section.

6.3 SCOP

The *Secure COmponent Protocol (SCOP)* enables the secure deployment of components in CASSIA. Since we introduce new security mechanism for deployment of components, we need a protocol that guarantees reliable and correct communications among all elements of the system. After formalizing SCOP, we proved its soundness using the BAN authentication logic. SCOP enables us to allocate a single key (fortress) for each component thereby reducing the demand for the size of the storage for keys from $O(n_c^2)$ to $O(n_f)$.

6.3.1 Rationale

With the abundance of various security protocols, why do we need SCOP? Component-based systems have multiple points of attack. Recently we discovered a variation of the impersonation attack and solution based on Kerberos was offered for component-based systems [6]. We need a protocol that is specific for component-based systems, offers comprehensive protection, and works seamlessly with CASSIA. SCOP is used only when CASSIA makes a decision to encrypt messages between components, and it ensures that client components interact with real servers and not their surrogates.

SCOP protocol is built using ideas from Kerberos and IPSec protocols. A cryptographic review of the IPsec protocol showed that the protocol was a disappointment [40]. The study showed that IPSec was too complex, and this led to large numbers of ambiguities, contradictions and weaknesses, IPSec was not 100% secure, and it was either required to decrease the complexity of IPSec and improve its modularization or another alternative should be found. SCOP addresses these issues while targeting specifically component-based systems.

```
CASSIA( Ci, Cj, boolean encrypt )
Begin
     Fi = Fortress( Ci );
     Fj = Fortress( Cj );
     if( Fi != Fj ) then
       Tij = Treaty( Fi, Fj );
       Pij = Protocol( Tij );
       if( encrypt == true ) then
         encrypt( Ci, Cj, Pij );
       else
         decrypt( Ci, Cj, Pij );
       endif
     endif
End
```

Figure 4: Adaptive algorithm for application of cryptography for CASSIA.

6.3.2 Description

Our assumptions do not go beyond the Dolev-Yao threat model [41]. We assume that the overall integrity of the operating system cannot be violated (i.e., no external threats can exploit general security breaches that may compromise the overall integrity of the system). We also assume that when installing any program the operating system creates a virtual shell effectively protecting the installation from being penetrated by an adversary. SCOP assumes that the installation of components is secure so that it can obtain shared keys from component infrastructure CI that is merged with the authentication server AS. During the installation, each principal (i.e. component) C sends a message to server CI/AS (we mean a security service within an operating system) that contains its identity and a private key that C uses to communicate with CI/AS: . This operation includes the installation and registration of components with the system-wide database. When CI/AS receives this initial message it stores it in a secure storage to which only CI/AS has access, and by our first assumption CI/AS cannot be compromised.

A SCOP messaging model that establishes communication channel between components C_i and C_j is shown in Figure 5. As the first step, we establish a secure communication channel between C_i and CI/AS so that CI/AS can request various services from component C_j securely. To do that, C_i sends message M_1 to CI/AS in which it asks CI/AS to instantiate C_j and to provide a reference to it. CI/AS receives message M_1 and sends message M_2 to C_j that contains the identity of component C_i, timestamp *nonce* (a unique number that guarantees freshness of a message) T_s that is used as a session key, and key $K_{j,as}$ for secure communication between CI/AS and C_j. Message M_2 is encrypted with shared key K_{as}.

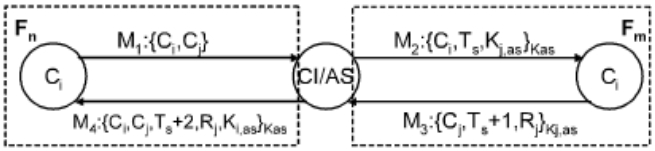

Figure 5: A SCOP messaging model.

C_j receives this message from CI/AS, decrypts it, and responds with message M_3 that consists of the incremented timestamp nonce T_s and reference R_j to the component C_j, en-crypted with the shared key $K_{j,as}$. With reception of the last message from C_j a secure communication channel is established between CI/AS and C_j. CI/AS extracts the reference R_j to the component C_j and forms new message M_4 comprising the incremented timestamp nonce T_s, unique identifiers of components C_i and C_j, and the reference R_j, shared key $K_{i,as}$ and encrypts this message with the shared key K_{as}. Then it sends this message to C_i. After decrypting it C_i holds the reference R_j to the component C_j.

When C_i needs to invoke a method of some interface of C_j, it sends a message to CI/AS that is encrypted with the shared key $K_{i,as}$ that contains the incremented timestamp nonce T_s, the reference R_j to the component C_j, the list of parameter values in the order in which they are specified in the declaration of the method of this interface, and the description of the requested operation (e.g., to invoke an interface of the component). CI/AS decrypts this message and invokes the requested method on behalf of C_i, and receives the return values. Then it forms a message encrypted with the shared key $K_{i,as}$ that contains the incremented timestamp nonce T_s, the reference R_j to the component C_j, the list of return values, and the description of the requested operation. This concludes the description of SCOP.

6.3.3 Soundness

Given SCOP we need to be sure that every time client C_i invokes methods of a component, i.e. $C_i \rightarrow C_j$ it actually communicates with the real component C_j and not some intercepting surrogate C_j'. Proving this property is in fact equal to establishing that the model of the protocol is sound, that is, it does not lead to wrong behavior when some intercepting surrogate C_j' impersonates the real component C_j.

Theorem. When two components communicate, i.e. $C_i \rightarrow C_j$, C_i believes that C_j is true.

The proof of this theorem is based on BAN authentication logic and is given in [9].

7 Case Study

For the case study we implemented the Pet Store application for MicoCCM based on the Java Pet Store demo that is a sample application developed by the Java BluePrints program at Java Software, Sun Microsystems [42]. This sample application is typical in using the capabilities of the underlying component infrastructures that enable robust, scalable, portable, and maintainable e-business commercial applications. It comes with full source code and documentation, so we used it to experiment with CASSIA and demonstrate that we can build scalable security mechanisms into enterprise solutions.

Our first implementation of the Pet Store application is based on the Java Pet Store demo and consisted of twelve components. For each next implementation we used the previous one as a base by taking component C that exports interfaces I_1, ..., I_n and dividing it in two components C_1 and C_2. Component C_1 exports interfaces I_1, ..., I_k and component C_2 exports interfaces I_{k+1}, ..., I_n, where $k = \left\lfloor \frac{n}{2} \right\rfloor$.

We produced five implementations of the Pet Store application. Its first implementation consisted of twelve components. These components exported a total of 132 interfaces that exposed a total of 183 methods. The second implementation had twenty five components, the third had fifty components, the next implementation consisted on seventy five components, and finally, the last contained one hundred components.

We designed the performance test to measure the reliability and sustainability of the transaction processing throughput of our implementation of the PetStore. The test script simulated users logging in, and then proceeding to individually order 100 items similar to the performance tests [35]. For each item ordered, the checkout process was completed, with the last step in this process being the actual placement of the order that activates a transaction, followed by a logout at the end of the script. Each virtual user therefore completes 100 individual transactions during a user session. The test was run at a user load providing peak throughput for duration of 24 hours to show if this throughput was sustainable. We repeated this test for our five implementations of Pet Store. We tested each implementation four times. The first test was run with all components marked in the configuration file as located within the same security perimeter so that no encryption/decryption was necessary. In the second test 20% of interacting components were marked as located in separate security perimeters. The third test was run with 50% of interacting components marked as located in separate security perimeters, and the last test was run with 100% of interacting components marked as located in separate security perimeters. The result of this test is shown in Figure 6 with the legend reflecting the percentage of components communicating through security perimeters. Four bars corresponding to the percentage of interacting components located in separate security perimeters are shown for each of the four implementations of Pet Store application. The leftmost bar in each experiment with a different number of components corresponds to the case when all components located within the same security perimeter, and it indicates the highest throughput since no cryptographic protocols are used when running this implementation. At the other extreme, the right-

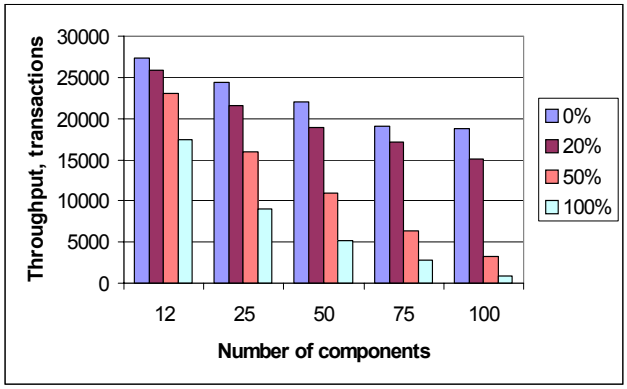

Figure 6: The throughput for 24 hour execution of the Petstore application depending on the number of components and percentage of those components communicating through the security perimeter.

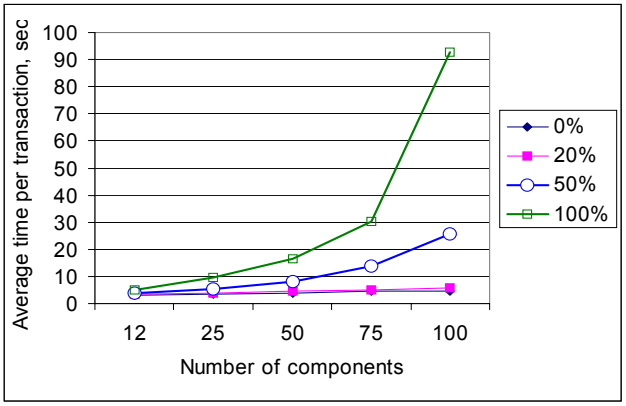

Figure 7: The dependency of the average time per transaction in seconds from the number of components in Petstore application.

most bar in each experiment with a different number of components corresponds to the case when all components are located within security perimeters, and it indicates the lowest throughput since all communications between components are encrypted and subsequently decrypted. Moreover, while the application throughput drops linearly in cases with 0-20% of components, the drop rate of the throughput is polynomial to exponential (based on the approximation of graphs shown in Figure 6 and Figure 7) in cases with 50-100% components located in separate security perimeters. This observation is confirmed by the graph shown in Figure 7 that depicts the dependency of the average time per transaction in seconds from the number of components in the Pet Store application. This graph shows that the increase rate of the throughput is in the range of polynomial to exponential in cases with 50-100% components located in separate security perimeters.

Table 2 shows the speedup provided by our solution with respect to a solution that use cryptographic algorithms for all communications. We measure an average time taken to execute a transaction in our five Pet Store application implementations with different numbers of communicating components for a standard implementation with a mandatory encryption/decryption, and an average time taken to execute a transaction with our CASSIA solution. The last column in Table 2 shows speedups for fifth implementation with different numbers of components. We consider this result to be indicative of at least of order of magnitude speedup due to the deployment of CASSIA.

Finally, we answer the question: how large is the overhead of CASSIA itself. In order to measure it we placed all components in the same fortress and ran the PetStore application with the CASSIA algorithm turned on. Since no encryption is required, the measurements of the average time required to complete a transaction with and without CASSIA will reflect

Number of Components	Average time with encryption, sec	Average time CASSIA, sec	Ratio of speedup
12	4.84	3.33	1.45
25	9.58	4	2.4
50	16.58	4.56	3.64
75	30.1	5.05	5.96
100	92.7	5.71	16.2

TABLE 2. Speedups of our adaptive CASSIA solution when 20% of communicating components interact through security boundaries.

the CASSIA's overhead. The results of this experiment are shown in Table 3. CASSIA's overhead ranges from 1.9% to 3.7% and on the average is 2.9%, which is quite small.

8 Conclusions

We analyzed the sources of nonscalability when using security solutions in LSCB systems and conducted an empirical study where we found that 80% of interactions between components and their clients occur within the same protected security boundaries. We evaluated a high probability bound showing the negligible probability that in LSCB applications over 30% of communicating components may be located in different fortresses. We successfully implemented CASSIA, a novel scalable security mechanism for LSCB systems that is based on our empirical findings and it utilizes the topology of the security perimeters and a pattern of interactions among components via their component infrastructure. We conducted CASSIA's performance evaluation case study that confirms its scalability and showed its very small overhead, proposed and implemented a Secure COmponent Protocol (SCOP) that incorporates our scalable security mechanism into a component infrastructure, and we have proved that our solution was sound in a separate technical report [9].

Number of Components	Average time without encryption, sec	Average time with CASSIA, sec	%, CASSIA overhead
12	3.16	3.22	1.9
25	3.55	3.63	2.25
50	3.9	4.04	3.7
75	4.53	4.66	2.8
100	4.61	4.77	3.6

TABLE 3. Overhead of our adaptive CASSIA solution.

9 References

[1] B. Meyer, "The Grand Challenge of Trusted Components," *The 25th International Conference on Software Engineering*, Portland, OR, 2003

[2] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming*, 2nd Edition, ACM Press, Addison-Wesley, 2002

[3] M. Sitaraman, "Compositional Performance Reasoning," *Proceedings 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction*, Toronto, Canada, May 2001

[4] P. Clements and L. Northrop, "Software Architecture: An Executive Overview," CMU/SEI-96-TR-003

[5] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 2nd edition, April 2003

[6] M. Grechanik and D. Perry, "Secure Deployment of Components," *2nd International Conference on Component Deployment*, Edinburgh, UK, May 2004

[7] B. Tung, *Kerberos: A Network Authentication System*, Addison-Wesley, 1999

[8] M. Bishop, *Computer Security*, Addison-Wesley, 2003

[9] M.Grechanik, D.Perry, and D.Batory, "CASSIA: Scalable Security Infrastructure For Large-Scale Component-Based Systems", Technical report TR04-31, The Department of Computer Sciences, The University of Texas at Austin, 2004

[10] M. Burrows, M. Abadi, and R. Needham, "A Logic of Authentication," *ACM SIGOPS Operating Systems Review*, 23(5), 1989

[11] S. Northcutt, L. Zeltser, S. Winters, K. Frederick, and R. Ritchey, *Inside Network Perimeter Security: The Definitive Guide To Firewalls, VPNs, Routers, and Intrusion Detection Systems*, New Riders Publishing, 2003, p.193.

[12] C. Benson and G. Nagendra, "Java SSL Performance: A Simple Exploration," Joint HP and Intel Study, http://h21007.www2.hp.com/dspp/ne/ne_EventDetail_IDX/1,1394,655,00.html

[13] R. Sessions, *Software fortresses : modeling enterprise architectures*, Addison-Wesley, 2003

[14] DoD Eligible Receiver Exercise. http://www.globalsecurity.org/military/ops/eligible-receiver.htm.

[15] B. Meyer, C. Mingins, and H. Schmidt, "Providing Trusted Components to the Industry," *IEEE Computer*, 1998, pp. 104-115

[16] J. Viega and G. McGraw, *Building Secure Software*, Addison-Wesley, 2002

[17] The Trusted Components Initiative: http://www.trusted-components.org

[18] N. Bagarathan and S. Byrne, "Resource Access Control for an Internet User Agent," *The 3rd USENIX Conference on Object-Oriented Technologies and Systems*, 1997

[19] U. Lindqvist, T. Olovsson, and E. Jonsson, "An Analysis of a Secure System Based on Trusted Components," *Proceedings of 11th Ann. Conf. Computer Assurance*, 1996

[20] B. Hartman, D. Flinn, and K. Beznosov. *Enterprise Security with EJB and CORBA*, Wiley Computer Publishing, 2001

[21] Object Management Groups security standards: http://www.omg.org/technology/documents/formal/omg_security.htm

[22] N. Brown, and C. Kindel, "Distributed Component Object Model Protocol - DCOM/1.0. Internet Draft," January 1996. http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-02.txt

[23] U. Lang and R. Schreiner, *Developing Secure Distributed Systems with CORBA*, Artech House, February 2002

[24] A. Diaconescu and J. Murphy, "A Framework for Using Component Redundancy for Self-Optimising and Self-Healing Component-Based Systems," *Workshop on Software Architectures for Dependable Systems (WADS)- International Conference on Software Engineering*, Portland, Oregon, May 2003

[25] A. Diaconescu and J. Murphy, "A Framework for Automatic Performance Monitoring, Analysis and Optimisation of Component Based Software Systems," *Proc. of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS '04), 26th International Conference on Software Engineering*, Edinburgh, UK, May 2004.

[26] M. Trofin and J. Murphy, "A Self-Optimizing Container Design for Enterprise Java Beans Applications," *Proc. of 8th International Workshop on Component Oriented Programming,* Germany, July 2003

[27] C. Smith and L. Williams, "Performance and Scalability of Distributed Software Architectures: An SPE Approach", *Parallel and Distributed Systems*, Vol. 13, No.2, February 2002

[28] 1] S. Bernardi, S. Donatelli, and J. Merseguer, "From UML sequence diagrams and statecharts to analysable Petri Nets models," *Proceedings of the 3rd International Workshop on Software and Performance*, 2002

[29] D. Petriu, C. Shousha, and A. Jalnapurkar, "Architecture-based performance analysis applied to a telecommunication system," *IEEE Transactions on Software Engineering*, 26(11), 2000.

[30] R. Pooley, "Using UML to derive stocastic process algebra models," *Proceedings of the 15th UK Performance Engineering Workshop*, 1999

[31] A. Mos and J. Murphy, "Performance Management in Component-Oriented Systems using a Model Driven Architecture Approach," *The 6th IEEE International Enterprise Distributed Object Computing Conference*, Lausanne, Switzerland, September 2002

[32] A. Liu, "An Approach for Constructing High Performance, Scalable Distributed Object Systems," *International Conference on Software Engineering*, Limerick, Ireland, June, 2000

[33] S. Chen, I. Gorton, A. Liu, and Y. Liu, "Performance Prediction of COTS Component-based Enterprise Applications," *5th ICSE Workshop on Component-Based Software Engineering*, Orlando, Florida, 2002

[34] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and Scalability of EJB Applications," *OOPSLA*, Seattle, Washington, 2002

[35] The Middleware Company, "J2EE and .Net Reloaded: Yet Another Performance Case Study," *The Middleware Company Case Study Team*, 2003

[36] G. Box, W. Hunter, and J. Hunter, *Statistics For Expirementers*, John Wiley and Sons, 1978

[37] N. Alon and J. Spencer, *The Probabilistic Method*, Wiley, New York, 1991

[38] A. Puder, *MICO: An Open Source CORBA Implementation*, Morgan Kaufmann, March 2000

[39] MICO Is CORBA: http://www.mico.org

[40] N. Ferguson and B. Schneier, "A Cryptographic Evaluation of IPsec," Counterpane Internet Security, Inc., 1999.

[41] D. Dolev and A. Yao, "On the security of public key protocols," *Proceedings of the IEEE 22nd Annual Symposium on Foundations of Computer Science*, pp. 350–357, 1981.

[42] http://java.sun.com/blueprints/code/index.html#java_pet_store_demo