Component isolation in the THINK architecture

Christophe Rippert
SARDES project, LSR-IMAG laboratory, CNRS-INPG-INRIA-UJF
INRIA Rhône-Alpes, 655 av. de l'Europe Montbonnot 38334 St Ismier Cedex France
Christophe.Rippert@inria.fr

Abstract

We present in this paper the security features of THINK, an object-oriented architecture dedicated to build customized operating system kernels. The THINK architecture is composed of an object-oriented software framework including a trader, and a library of system abstractions programmed as components. We show how to use this architecture to build secure and efficient kernels. Policy-neutral security is achieved by providing elementary tools that can be used by the system programmer to build a system resistant to security hazards, and a security manager that uses these tools to enforce a given security policy. An example of such a secure system is given by detailing how to ensure component isolation with a elementary software-based memory isolation tool.

1 Introduction

All-purpose operating system are well suited for everyday work on workstations and servers, but specific platforms like embedded systems usually require a greater flexibility than these systems have to offer. On the other hand, writing a system from scratch is a tedious and difficult task that increases production costs and requires expert programmers. The THINK architecture is a solution to this problem as it provides an object-oriented framework and a complete library of operating system abstractions for the programmer to use when he builds his system. One of the main advantages of the THINK architecture is the flexibility it provides to the system programmer, since all components are optional and can be loaded and unloaded dynamically. However, flexibility must not be ensured at the expense of security and quality of service, especially in systems where resources are scare.

We present in this paper the work we have conducted in the THINK architecture to provide a secure framework. We first present the THINK architecture and its framework. Then we describe the security manager, a component dedicated to the management of security policies, and the software-based memory isolation mechanism we have implemented, and present how they can be used to provide component isolation. Finally we list some related works before concluding.

2 The Think architecture

The distributed systems architecture THINK is a platform for the development of distributed operating systems kernels. The goal of the THINK architecture is to ease the development of efficient, flexible, and secure operating systems. THINK provides the system programmer with interfaces that reify the underlying hardware, and optional system abstractions proposed as libraries. The development of a kernel with THINK is made easier by its object-oriented framework, since in THINK, all resources (both hardware and software) are considered as objects. These objects export interfaces which define their behaviour and make them accessible to other objects. Each interface has a name in a given naming context, and is linked to other interfaces by bindings. A binding is essentially a communication channel between two objects. These bindings can take many forms, as simple as the association between a variable name and its value, or more complex like a binding over a network between two objects on different machines. Bindings are created by dedicated objects called binding factories,

whose main function (i.e. creating bindings) can be freely extended to enforce a chosen behaviour. Finally, objects can be grouped in domains according to a common property (e.g. security domains, fault-domains, etc).

All those concepts are presented in the minimal software framework detailed in Figure 1. Java is used in THINK as the interface description language[1]. The `Top` interface is the greatest element in the THINK type lattice, the common type from which all interfaces derive. The `Name` interface is the common type for all names in THINK. The method `getDefaultNC` returns the current naming context and the method `toString` provides a serialized form of the name. The `NamingContext` interface is the common type for all naming contexts in THINK. Its method `toName` deserializes a name known as a String. The method `export` provides a name for a given interface. As a side effect, it also creates a binding between the returned name and the given interface. The `BindingFactory` interface is the common type for all binding factories in THINK. The method `bind` creates a binding between the calling object and the object which name is given to the method.

The THINK framework also includes a simple trader component. This trader exports two methods, `register` and `lookup`, whose prototypes are given in Figure 2. The `register` method permits to publish the reference (i.e. an instance of `Name` in Think) to a given service under a symbolic name which eases its localisation and abstracts its implementation. For example, a memory manager can be registered under the symbolic name "`mem_manager`" which other components will use to locate it. Moreover, the THINK architecture supports the registration of multiple services under the same symbolic name. For example, the symbolic name "`mem_manager`" can represent two different memory manager, one for flat memory and one for paged memory. The `lookup` method is used to find the reference to a service known by its symbolic name. The second parameter, `hints`,

[1]The choice of the Java language as an IDL was guided both by the will to use a language known by a large number of programmers, and also considering the fact that we are currently working on a new version of the THINK architecture in which application level components will be fully written in Java and compiled to binary code by way of a dedicated Java to C compiler.

```
interface Top {
}

interface Name {
  NamingContext getDefaultNC();
  String toString();
}

interface NamingContext {
  Name toName(String name);
  Name export(Top itf);
}

interface BindingFactory {
  Top bind(Name name);
}
```

Figure 1: The core software framework in THINK.

is used to specify which service is wanted in case of multiple registration under the same symbolic name. For example, in the previous example of two memory managers, the programmer could call `lookup(``mem_manager'', ``flat'')` to obtain the `Name` of the flat memory manager.

```
interface Trader {
  void register(Name name,
                String symbName);
  Name lookup(String symbName,
              String hints);
}
```

Figure 2: The Trader component interface.

A more detailed presentation of the THINK framework can be found in [1].

3 Protection in the Think architecture

The THINK architecture permits to build flexible and adaptable systems. However, security is a critical issue in a modern operating system and should not be compromised by flexibility. We are working to provide in the THINK architecture the tools necessary for the system programmer to build a secure system which offers protections of data and

resistance to denial of service (DoS) attacks. These tools must be policy-neutral so as not to compromise the flexibility of the system. On the other hand, the system programmer must be able to define the security policy that suits his system best and have it enforced. We present here the security manager, a component dedicated to manage and enforce security policies, and an elementary security tool, a software-based memory isolation mechanism, and show how these two entities can be combined to provide component isolation in the THINK framework.

3.1 The security manager

The security manager is the key component in THINK security framework. It is responsible for managing the security policy defined by the system programmer. This policy specifies how the system should react when confronted to security hazards. The security manager uses the elementary tools implemented in the kernel to enforce the given policy. Thus, the tools remain completely policy-neutral and only the security management is concerned with the details of the security policy.

The security manager is represented here as a centralised entity as a simplification, but it can also be distributed. A distributed security manager can be seen as a federation of elementary security managers, each one managing a security domain in the system. For example, if the policy specified by the programmer includes a global resource allocation scheme, it is possible to associate a security manager with each resource in the system. Each elementary security manager then manages its resource, and global decisions are made by a central security manager responsible for managing the global policy, but not concerned with the details of each resource management.

The language used to express the security policy is obviously a key element in this architecture. To manage the QoS in a system for example, the security manager must find a compromize between the requests of the various applications which need to use resources, and the current resource allocation state in the system, all the while taking into account the general guidelines specified in the security policy. This requires a constraint solving algorithm based on a constraint declaration language which enables the system programmer to easily specify the security policy he wants to enforce. One again, a distributed security manager is probably more appropriate than a centralised one, since each elementary security manager needs then to resolve only the constraints linked to the resource it manages and so can be specialised and more efficient. Some work remains to be done on that topic, but we believe that Domain Specific Languages [2] are the right tool for that purpose.

3.2 Software memory isolation

The software memory isolation mecanism implemented in THINK consists of parsing the process code at creation time, and replace each memory access (i.e. load, store and branch instructions) by a call instruction to a well-known label in the security manager. This label is the entry point of a method whose task is to check that the destination address of the checked instruction is in a memory area which the process can access. The calling process can easily be identified by the return address which is stored on the stack or in a dedicated register by the call instruction, since this address must point to a memory area which belongs to the calling process. The checking method of the security manager then access a table where the replaced instruction has been stored, and according to its type (i.e. a store, a load or a branch), check that it has the right to do what it is supposed to do (i.e. a process can access a read-only area with a load but not with a store or a branch). If the access is allowed, it is executed and then execution goes back to the calling process. Otherwise, an exception is thrown.

This permission checking is fully customizable by the system programmer which defines the chosen security policy managed by the security manager. Thus, he can define memory areas with different permissions on a per-process basis. For example, an area which a component can branch but not read or write to can be seen as an execution-only area, just as some hardware isolation mecanisms permit to define execute-only segments for application code. However, in the case of hardware isolation, permissions are usually globally fixed (i.e. if a segment is tagged as read-only, then no process in the system can write to that segment, be

it a system or an application process[2]). Our software isolation mecanism on the other hand permits to define permissions on a per-process basis: since permission checking is done by changing code in the process itself, the same memory area can be tagged as read-only for one process and execution-only for another one for example, thus achieving a complete flexibility in the isolation of components.

We present how to use the THINK framework to isolate components by detailing a simple example. Considering two components, a client component and a server component, we want to ensure that the client component which wants to call the `alloc` method exported by the server component is allowed to do it by the security policy defined by the system programmer. The figure 3 illustrates this example. The sequence of actions necessary to call the `alloc` method is detailled below:

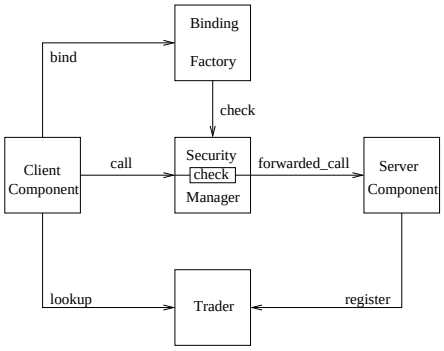


Figure 3: Software framework for component isolation

1. The server component registers itself in the trader, thus exporting the `alloc` method through its interface.

2. The client component uses the `lookup` method of the trader to find the service provided by the server component.

3. The client component requests the creation of a binding between itself and the server component to the local binding factory.

4. The binding factory checks that the client component has the right to create a binding with the server component.

5. The client component calls the security manager which checks that the client has the right to call the `alloc` method. If the securiy policy allows it, then it forwards the call to the `alloc` method of the server component.

Compared to inter-process communication over hardware isolation, software-based memory isolation has proven its efficiency (see [3] section 5). In our architecture, we monitored the cost of an absolute branch instruction with software-based memory isolation and found that the increase of the execution time was only of +16.67%. We compared this inter-process call with an optimised LRPC [4] implemented over hardware isolation in THINK and found that the LRPC is more than 25 times slower than our mechanism.

4 Related work

Compared to the OSKit [5] framework and set of libraries, THINK provides a better flexibility since all components are independent. The KaffeOS [6] project proposes some techniques to preserve quality of service in a Java environment, using the type safety properties of the language. Similarly, the SPIN [7] extensible operating system uses the properties of the Modula-3 language to permit the safe binding of modules in the operating system. In THINK, we aim to remain independent of the component development language. In the DTOS [8] project, policy-neutral security is enforced by way of security servers, which check that inter-component calls are allowed. This requires a modification of the component source code, whereas in THINK, binding factories can make security checks, and the binary code of the component is modified by the software-based memory isolation mechanism without needing any modification of the component source code. The Scout/Escort [9] project focuses on protection against denial of service attacks by defining the I/O path abstraction. However, this does not take into account the resources allocated in the operating system kernel, whereas in THINK, we aim toward a global view of the resources which

[2]Some architectures (e.g. the Intel ia32 architecture) permit to define privilege levels for processes of different classes (e.g. kernel, system services, applications, ...), but this usually remains very restricted (the Intel ia32 architecture defines only 4 different privilege levels for example)

enable the system to associate each allocated resource with the benefiting user. This point of view is close to the resource containers abstraction [10], although this work has been conducted in monolithic kernels whereas in THINK we advocate a more modular architecture. The exo-kernel [11] advocates the same philosophy of a minimalist kernel, although it does not propose an object framework for the components as in THINK. Reflective operating systems such as Apertos [12] also propose a philosophy close to ours since they give to applications a "grey-box" view of the system and some means to customize it as needed, but they seldom go as far as the THINK architecture which provides full access to the low levels of the system (if the security policy allows it of course) and advocates a "white-box" view of the kernel closer to the exo-kernel architecture.

5 Conclusion and future work

As we have seen in this paper, the THINK architecture can be used to build flexible and secure operating systems. The software-based memory isolation mechanism presented here is one of the policy-neutral elementary tools provided in the THINK architecture. Coupled with THINK component-based framework and high-level abstractions like a policy-aware security manager, these tools make the THINK architecture a valuable base for the system programmer to build a secure and DoS attack resistant system. With that secure and flexible framework, we believe that the THINK architecture is a fitting tool for the building of customised and efficient operating system kernels.

References

[1] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia Lawall and Gilles Muller. THINK: A Software Framework for Component-based Operating System Kernels. In Proceedings of the USENIX Annual Technical Conference, 2002.

[2] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages – An Annotated Bibliography. ACM SIGPLAN Notices, 2000.

[3] Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham. Efficient Software-Based Fault Isolation. In Proceedings of the ACM SIGOPS'1993.

[4] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, Henry M. Levy. Lightweight Remote Procedure Call. In ACM Transactions on Computer Systems, Vol. 8, No. 1, February 1990, pages 37-55.

[5] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, 1997.

[6] Godmar Back, Wilson C. Hsieh, Jay Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation, 2000.

[7] Przemyslaw Pardyak, Brian N. Bershad. Dynamic Bindings for an Extensible System. In Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation, 1996.

[8] Duane Olawsky, Todd Fine, Edward Schneider, Ray Spencer. Developing and Using a "Policy Neutral" Access Control Policy. In Proceedings of the New Security Paradigms Workshop, 1996.

[9] Olivier Spatscheck, Larry L. Peterson. Defending Against Denial of Service Attacks in Scout. In Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation, 1999.

[10] Gaurav Banga, Peter Druschel, Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation, 1999.

[11] Dawson R. Engler, M. Frans Kasshoek, James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In Proceedings of the 15th ACM Symposium on Operating Systems Principles, 1995.

[12] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In Proceedings of the 7th ACM conference on Object-Oriented Programming, Systems, Languages, and Applications, 1992.