Protection in Flexible Operating System Architectures

Christophe Rippert

Protection in Flexible Operating System Architectures

Christophe Rippert
Christophe.Rippert@inria.fr

SARDES project, LSR-IMAG laboratory
INRIA Rhône-Alpes, 655 avenue de l'Europe
Montbonnot 38334 St Ismier Cedex, France

Abstract

This paper presents our work concerning flexibility and protection in operating system kernels. In most existing operating systems, security is enforced at the price of flexibility by imposing protection models on the system programmer when building his system. We prove that flexibility can be preserved by separating the management of the protection policy from the tools used to enforce it. We present the secure software framework we have implemented in the THINK architecture to manage protection policies and guarantee they are carried out as specified. We then detail the elementary protection tools provided to the programmer so he can protect his system against unauthorized accesses and denial of service attacks. These tools are implemented in a policy-neutral way so as to guarantee their flexibility. Finally we validate our results by evaluating the flexibility of the protection provided on selected examples of dynamic modification of the protection policy.

1 Introduction

Flexibility and security are often seen as two conflicting goals in traditional operating systems. Securing a system often results in imposing constraints on the system architecture and on the services provided. On the other hand, improving the flexibility of the system is generally seen as introducing vulnerabilities. For example, recent Linux kernels supports the use of *modules*, that is system services which can be loaded and unloaded at will in the kernel. This permits the system administrator to add functionnalities to the kernel without having to recompile it, thus enhancing flexibility. However, a malicious user might trick the system administrator into loading a harmful module in the kernel to gain extra privileges for example. On the other hand, forbidding the use of modules strengthens the security of the system but reduces its flexibility since the system administrator is forced to recompile it each time he wants to add a service.

We believe that it is possible to guarantee both flexibility and security in an operating system. Our work focuses especially on protection as defined in [5], that is access control and protection against denial of service attacks. We prove in this paper that flexibility can be preserved in a secure system by separating the management of the protection policy from the tools used to enforce it. The protection policy is the set of rules specifying the access rights of principals and the management of ressources in the system. We thus propose a secure software framework that regulates the interactions between the components in the system and includes the security manager, a component dedicated to the management of policies. Coupled with a set of policy-neutral protection tools, we guarantee that protection policies can be changed dynamically to suit the needs of the system administrator, without needing to recompile or reboot the system, thus preserving its flexibility. Our work is validated using THINK, a flexible system architecture which permits the programmer to build customized operating systems.

We first present the THINK architecture and the secure software framework we have implemented with it. We then describe the elementary protection tools we have specified and implemented to be policy-neutral. Finally, we present the experiments we have conducted to test the flexibility of our protections tools by dynamically changing the protection policy. We conclude by presenting the future work we plan to conduct concerning the specification of protection policies.

2 The Think architecture

The THINK[1] architecture aims at providing a programming model and a set a tools to ease the task of system builders. This architecture is specified in [4] where its first implementation is also described. This implementation does not address protection issues and let the system vulnerable to abuses. We present below our secure implementation of the THINK software framework, before detailling in Section 3 the elementary protection tools we have devised.

2.1 Architecture

THINK is composed of three main entities: the nano-kernel, the service library, and the software framework.

The nano-kernel is the part of the kernel in charge of booting the system and providing access to the underlying hardware. It does not add any system abstraction nor provides any service. It is meant to be as small as possible since it is a mandatory part of the system and we want to minimize the code imposed to the system programmer, as advocated by the Exokernel philosophy [3].

The service library is composed of system services which can be used by the system programmer when he builds his system. Each service is implemented as a stand-alone component. It means that a service can depend on another service's interface (as for example a file system depends on the interface of the hard drive driver), but not on the implementation of this service. Thus, any service can be replaced by another one exporting the same interface. This provides a complete flexibility since no component is mandatory in the system and the programmer can replace any library component by his own if it better suits his needs.

The software framework materializes the programming model advocated in THINK. It is based on the ODP model [6], and includes the well-known notions of interface, component, binding, binding factory, name and naming context. This model specifies the way components must interact, as presented here with the example of a component `Src` calling the method `meth` of the component `Dst`:

1. Component `Dst` exports its interface with the `export` method of the naming context interface.
2. Component `Src` uses the `bind` method of the binding factory interface to create a binding with the interface exported by component `Dst`.
3. Component `Src` calls the method `meth` of the `Dst` component using the `call` method provided by the software framework. In the first implementation of the THINK framework, this `call` method is in fact a wrapper so as to optimize its performances. It is important to note that methods exported by components cannot be called directly since they are protected by the `static` C statement.

2.2 Implementation

We present here the secure version of the Think software framework. This framework guarantees that the interactions between the components respect the protection policy specified by the system programmer. We present the implementation of the main notions of the programming model proposed in the THINK architecture.

[1] THINK stands for *THink Is Not a Kernel*

2.2.1 Interfaces

Interfaces are implemented as a structure composed of the static name of the exporting component, its methods and its public data. The static name is an unique identifier which can be used to statically define the access rights of the component.

2.2.2 Name

Names represent interfaces in a given naming context. Our implementation of a name is a structure composed of an identifier used to find the interface corresponding to the name, and the naming context in which the name is defined.

2.2.3 The security manager

The security manager is the component dedicated to the management of security policies on the system, which includes protection policies. Its interface is described in Figure 1.

```
Secret getSecret(Itf itf);                boolean checkBind(char *staticName,
                                                            Secret secret,
                                                            Name dstName);

boolean checkExport(char *staticName,     boolean checkCall(char *staticName,
                    Secret secret,                          Secret secret,
                    Itf itf,                                Itf dstItf,
                    NamingContext nc);                      char *methStaticName);
```

Figure 1: Interface of the security manager

The `getSecret` method is called when a component is initialized and returns a secret token. This token permits to dynamically identify a component in a way similar to a PGP public key [9] which identifies a user. In our implementation this token is a 128-bit integer, which makes it statistically very difficult to forge. The `checkExport` method is used to check that the calling component has the right to export the interface `itf`. The `checkBind` method verifies that the calling component can create a binding with the interface `dstName`. The method `checkCall` checks that the calling component has the right to call the method `methStaticName` exported in the interface `itf`.

2.3 Naming contexts

Naming contexts are components managing the naming tables containing the names of the interfaces in the system. The interface of a naming context is presented in Figure 2.

```
Name byteToName(NamingContext nc,     Name export(NamingContext nc,
                char *str);                       char *staticName,
                                                  Secret secret,
                                                  Itf itf);
```

Figure 2: The `Naming Context` interface

The `byteToName` method takes as a parameter the serialized form of a name and returns that name if it exists in the given naming context. In our implementation, names are serialized as strings. The `export` method is used by a component identified by its static name and its secret to export an interface in the given

naming context. It returns a name which identifies the interface in the naming context. This method first calls the `checkExport` method of the security manager to verify that the component has the right to export the given interface in this naming context.

2.4 Bindings and binding factories

Bindings are not represented as components in THINK but they can be seen as communication channels between components. Binding factories are components exporting the interface presented in Figure 3.

```
Itf bind(char *staticName,
         Secret secret,
         Name dstName);
```

Figure 3: The `Binding Factory` interface

The `bind` method creates a binding between the calling component, identified by its static name and secret, and the interface which name is given as a parameter. It returns the interface corresponding to the given name. Before creating the binding, the `bind` method calls the `checkBind` method of the security manager to verify that the component has the right to create it.

2.5 Inter-component calls

Components call interfaces using the method exported by the inter-component calls component which interface is presented in Figure 4.

```
void *call(char *staticName,
           Secret secret,
           Itf dstItf,
           char *methStaticName,
           void *params);
```

Figure 4: The `inter-component calls` interface

The `call` method is used by a component identified by its static name and secret to call a method exported by a given interface. The called method is identified by its static name so as to permit the static specification of call rights. This method first calls the `checkCall` method of the security manager to verify that the component has the right to call the method. If this verification succeeds, the method is called with the proper parameters.

3 Elementary protection tools

We have implemented some elementary protection tools that we have included in THINK library. We call these tools elementary since each one can protect the system against a particular threat. All these tools are implemented as components and thus remain independant from each other. We show that these tools are policy neutral which ensures their flexibility.

3.1 An adaptable disk manager

3.1.1 Principle

Classical disk managers typically use a geographical scheduling policy which treat request according to the distance between the desired sector and the current position of the head [8]. These algorithms are well suited for all-purpose operating systems but not for specific uses as in real time or multimedia systems for example. Moreover, they are usually vulnerable to denial of service attacks as a malicious process issuing lots of requests so as to slow down the treatment of other requests. We have implemented an adaptable disk manager which permits to dynamically change the scheduling policy. The scheduling algorithm can thus be changed according to the needs of the applications or to counter a denial of service attack.

3.1.2 Implementation

The disk manager we have implemented is detailled in Figure 5.

```
typedef struct {
  int readOrWrite;
  unsigned sector;
  char *buffer;
  unsigned pid;
} DiskReq;

void initQueue();
char *readSector(unsigned sect,
                 char *buffer,
                 unsigned pid);
void writeSector(unsigned sect,
                 char *buffer,
                 unsigned pid);
void flushQueue();

DiskReq *getQueue();
void setSched(void (*f)());
void delReq(DiskReq *req);
```

Figure 5: Interface of the adaptable disk manager

The `DiskReq` structure describes a request emitted by a process identified by its process identifier (`pid`) to a given sector. The `initQueue` method initializes the request queue and the `readSector` and `writeSector` methods permit components such as a file system to send read and write requests to the disk manager. The `flushQueue` method treats the requests stored in the queue when called by the process managing the burst mode.

The `getQueue` method is our first elementary tool. It offers a view of the current state of the request queue to the security manager to help him take decisions according to the protection policy.

The `setSched` method is our second elementary tool. It permits the security manager to change the scheduling algorithm by passing the corresponding function as a parameter to the `setSched` method. This function is called by the `flushQueue` method before beginning to treat the requests, so as to sort them in an order matching the scheduling policy.

The `delReq` method is our third elementary tool. It can be used by the security manager to remove a request from the queue. This is useful during a denial of service attack to clean the queue of the requests emitted by the malicious process.

All these tools are policy neutral since all decisions are made by the security manager. These tools provide the necessary support for the security manager to enforce the protection policy. For example, the security manager can detect a denial of service attack using the `getQueue` method and change the scheduling algorithm with the `setSched` method or erase the offending requests with the `delReq` method. Criteria defining a denial of service attack and the countermeasures to take in case of attack are specified by the protection policy.

3.2 Software-based memory isolation

3.2.1 Principle

Memory isolation is a mandatory mechanism in a multi-user operating system. It is indeed necessary to prevent a malicious or malfunctionning process to read or modify other processes' data. However, most operating systems base memory isolation on hardware mechanisms provided by the processor, like segmentation for example. This approach has several drawbacks. First it requires that the memory management unit of the processor provides an isolation mechanism which is not always true especially on embedded systems. Moreover, hardware isolation usually induces a high overhead for interprocess communication since processes must use services as sockets to circumvent the hardware isolation. Finally, hardware isolation lakes flexibility since the test carried out to validate the access is hard coded in the memory management unit and cannot be changed by the system programmer. Thus, we chose to implement a software-based memory isolation mechanism that allow the programmer to fully customize the access control he wants to enforce in his system.

3.2.2 Implementation

The tool we have implemented is inspired from the software-based fault isolation mechanisme presented in [7]. When a process is created, the tool parses the binary code of this process and replaces memory accesses with a branch to a well-known entry point in the security manager. This entry point corresponds to a function which task is to check that the memory access is allowed by the protection policy. Thus all memory accesses are checked dynamically when the process code is executed. The access is executed if it is allowed by the protection policy and the execution of the process continues. If the verification fails, a security exception is thrown so as to notify the operating system.

This tool is very flexible since the access control is enforced by a function programmed by the system builder. Thus he can program it as needed to fit the protection policy he wants to enforce in the system. Moreover, our tool permits to choose which memory access must be checked. The programmer can for example decide to verify only branches and not load and store instructions if it suits his needs.

3.2.3 Evaluation for segment matching

The tool we have implemented permits to implement any kind of access control. To evaluate it, we have programmed a segment matching policy in the security manager. This policy simply consists in verifying that the destination address of the memory access belongs to a segment defined by its lower and upper bounds. It is very similar to the technique presented in [7]. We detail its operation on a simple example.

The original code of the process is presented in Figure 6. It is composed of a single PowerPC load instruction which task is to copy in register R1 the content of the memory word at the address computed by adding 8 to the content of register R2.

```
Initial_code:
  lwz 1,8(2)
```

Figure 6: Initial code: 32-bit integer loading

This code is modified by the memory isolation tool by replacing the load instruction by a branch to the checking function in the security manager, as detailled in Figure 7.

The verifying code can be generated dynamically when the process is created since this code is very simple. This permits to generate a code optimized for the replaced instruction, as shown in Figure 8. For a more complicated verification it may not be possible to dynamically generate the checking function. In our example, the code first calculate the destination address in a temporary register (`add 14,2,8`), then check this address against the lower and upper bounds, and raise an exception if the address is not in the allowed

```
Modified_code:
  ba Entry_Point_LWZ
```

Figure 7: Modified code: branching to the security manager

segment (`tw 8,14,15` and `tw 16,14,16`). If the test is successful, the original instruction is executed (`lwz 1,8(2)`) and the instruction pointer returns to the next instruction in the process (`ba Modified_Code + 4`).

```
Verification_code:
  add 14,2,8
  tw 8,14,15
  tw 16,14,16
  lwz 1,8(2)
  ba Modified_Code + 4
```

Figure 8: Generated code: checking the destination address and loading the integer if allowed

A simple optimization of this tool is not to generate code for memory access which can be checked before the process execution. For example, the PowerPC platform provides a relative branch instruction for which the destination address is computed by adding an immediate value to the address of the branch instruction. Since this immediate value is coded into the branch instruction itself, the destination address can be computed when the process is created and verified without having to execute the code. This optimization is very effective since relative branches are mostly used in loops for which a dynamic verification for each iteration would be very costly.

3.2.4 Benchmarks

Code generation delay The verification code is generated when the process his created in our example for segment matching. We monitored the delay induced by this code generation and found that the process creation delay increases by 3.20 ms for a test program composed of 29000 instructions (including 9000 memory accesses). Since the time necessary to read the file on the hard drive and allocate the memory for the process is approximately 16.5 ms, the overhead of our tool is only 20%, which seems moderate for most programs.

Execution overhead We monitored the overhead induced by the execution of the verification code for various program types.

Load/store instruction The execution time of a single load or store instruction is multiplied by 3.5 when verified by our tool. This might seem prohibitive but obviously a program is never composed only of load and store instructions.

Relative branch There is no overhead for the exectution of a relative branch since the destination address of this type of instructions can be computed when the process is created, using the optimization presented above.

Absolute branch This type of instructions is mostly used for interprocess communication. We thus compared the cost of an inter-segment call using our software isolation mechanism with the cost of an IPC call over a hardware isolation provided by the processor. The IPC mechanism is based on LRPC [1] and takes 0.490 μs. With our software isolation tool, the delay is 0.019 μs which is 26.4 times faster. Compared to an

absolute call without any isolation, the overhead is only of 16.67%, which seems very moderate. Software-based memory isolation is therefore very interesting for interprocess communication.

Square root computation This algorithm is an example of a program which includes very few memory accesses. It is an iterative algorithm based on Heron of Alexandria's well-known method for computing square roots. The overhead for the computation of the square root of 100000 is only 1.55% which is obviously negligible.

GZip This program includes lots of memory accesses since it is based on the comparison of strings stored in a dictionnary. Using maximum compression (with the `-9` option), the overhead for a 128MB file is 117%, which can be considered prohibitive.

Conclusion These benchmarks show that performances of a software-based memory isolation mechanism strongly depend of the type of program isolated. However, the system programmer can choose which programs he wants to isolate with it, which is not possible with a hardware isolation. Moreover, the system programmer can customize the access control he wants to enforce by changing the protection policy as needed, thus permitting him to implement lightweight tests for performance critical applications.

4 Evaluation

To evaluate the flexibility of our framework and our tools, we present below several situations where the protection policy is dynamically modified during execution of the operating system.

4.1 Dynamic modification of the memory isolation politic

4.1.1 Using the software-based memory isolation tool

We show on a simple example how our software-based memory isolation tool can be dynamically reconfigurated. In our system, component C2 is exporting a set of public variables which might be accessed by component C1, but only after they have been initialized. So the segment containing component C2 is initially isolated from the segment containing component C1, as shown in the access matrix in Figure 9.

	SegC1	SegC2
SegC1	RWX	$\emptyset$
SegC2	$\emptyset$	RWX

Figure 9: Access matrix for the memory isolation tool

In the security manager, the access rights are stored as capacity lists [5]. These capacities are protected from tampering since the security manager itself is located in a isolated segment. The read, write and execution access rights are stored as boolean which can be modified using the interface presented in Figure 10.

The method `getRights` permits to get the current access rights of the segment containing the component identified by its static name and secret. This method returns the read, write and execution rights for the segment `segSrc`. The method `setRights` permits to set the access rights in a similar way.

This scheme guarantees the flexibility of the isolation. The memory isolation tool generates code to check that the replaced instruction has the right to access the destination segment, which basically means checking that the corresponding boolean is set to true. Thus, all the programmer of component C2 needs to do to grant to C1 the access to C2 public variables after they have been initialized is to set the boolean using the `setRights` method. The modification of the policy is immediate and the generated code does not have to be changed. Dynamic modification of the isolation policy is therefore very simple thanks to the separation

```
void getRights(char *staticName,     void setRights(char *staticName,
               Secret secret,                       Secret secret,
               unsigned srcSeg,                     unsigned srcSeg,
               boolean *read,                       boolean read,
               boolean *write,                      boolean write,
               boolean *exec);                      boolean exec);
```

Figure 10: The access rights modification interface

between the management of the protection policy in the security manager and the tool used to enforce it, that is the software-based memory isolation mecanism.

4.1.2 Combining the tool and the framework

The memory isolation tool provides a way to isolate areas of memory from each other. A finer grain access control can be provided by combining the isolation tool and the secure framework implemented in THINK. Figure 11 illustrates an example of a component C1 calling method `meth` of component C2.

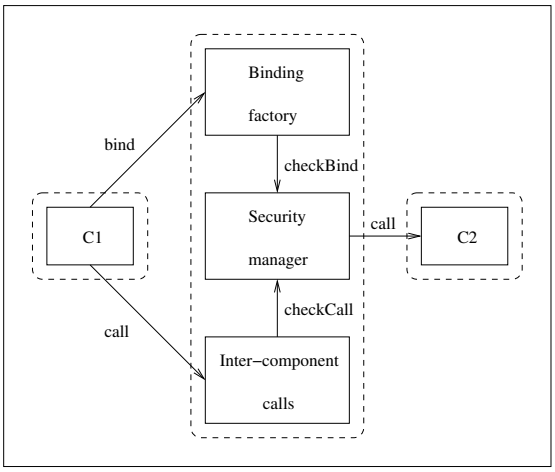

Figure 11: Fine-grain access control between two components

The dashed lines represent segments enforced by the memory isolation tool. The access rights for this isolation are summed up in Figure 12.

	SegC1	SegC2	SegFramework
SegC1	RWX	$\emptyset$	X
SegC2	$\emptyset$	RWX	X
SegFramework	RWX	RWX	RWX

Figure 12: Access matrix for the three segments

As can be seen in Figure 12, component C1 and C2 can call methods of the framework since it is located in a segment for which they have the execution right, but cannot call each other's methods directly. So component C1 is forced to use the binding factory and inter-component calls interface to call component

C2's method `meth`, thus allowing the framework to check that the protection policy allows that call, using the `checkBind` and `checkCall` methods of the security manager.

4.2 Dynamic modification of the disk scheduling algorithm

To test the flexibility of our adaptable disk manager, we devised a simple scenario of a denial of service attack. A malicious process which wants to slow down accesses to the disk simply floods the disk manager with requests chosen according to the current scheduling algorithm. For example, if the current algorithm implements a Shortest Seek Time First scheduling [8], the malicious process sends lots of requests for sectors close to the current position of the head so as to cause a starvation for the other processes.

The security manager can detect this kind of attacks using the `getQueue` method of our adaptable disk manager. A denial of service attack is usually caracterized by lots of requests coming from the same process. The security manager decides according to the protection policy if the current distribution of requests in the queue corresponds to an attack and what countermeasures should be taken. The security must describe the characteristics of an attack since a great number of requests coming from the same process in the queue can be legitimate if this process needs to access a big file on the disk for example.

The protection policy can specify that if a denial of service attack is detected, the current disk scheduling algorithm must be replaced by a fair scheduling algorithm. This type of algorithms reorganizes the requests in the queue according to the process which emitted them. Figure 13 describes an example of this reorganization for two processes P1 and P2 and compares it with a Shortest Seek Time First scheduling. A `PxRy` request means "request emitted by process Px to read sector y".

```
Emission order :  [P1R1, P1R8, P2R4, P2R1]
Fair scheduling:  [P1R1, P2R4, P1R8, P2R1]
SSTF scheduling:  [P1R1, P2R1, P2R4, P1R8]
```

Figure 13: Reorganization of requests by a fair algorithm

It is important to note that this fair algorithm is not meant to be used as a default scheduling algorithm since its performances are much worst than most geographical algorithms. Its goal is only to prevent starvation by interleaving requests from the processes accessing the disk. When using this algorithm, performances will therefore be slowed down equally for all processes, instead of being very high for the attacking process and very low for all the others.

Once the risk of starvation is cancelled by the fair scheduling algorithm, the security manager can use the `delReq` method to delete from the queue requests coming from the attacking process. It can also ask the process scheduler to kill the offending process if the protection policy requires it. The security manager may need to analyze the request queue for some time before deciding to kill the process, which emphasize the need for a fair disk scheduling algorithm to protect from starvation the processes accessing the disk legitimately during the time needed to take the decision.

This example illustrates the flexibility of the adaptable disk manager. By separating the management of the scheduling policy (implemented by the function passed as a parameter of the `setSched` method) from the tool used to enforce it (which is the adaptable disk manager itself, with its methods `readSector`, `writeSector` and `flushQueue`), we guarantee that the scheduling policy can be changed dynamically without modifying the tool itself. We chose here an example of a denial of service attack to illustrate a situation where the scheduling algorithm needs to be changed, but the adaptable disk manager can also be useful to adapt the scheduling policy for different types of applications. For example, the system administrator of a general-purpose operating system may want to change the scheduling policy to an algorithm optimized for multimedia applications when such an application is executed on the system, then switch back to a classical geographic algorithm when it is over.

5 Conclusion and future work

As we have seen in this paper, it is possible to ensure both flexibility and protection in an operating system. By separating the management of the protection policy from the tools used to enforce it, we guarantee that the protection policy can be changed dynamically without modifying the tool. The secure framework we have implemented in the THINK architecture includes the security manager, a component dedicated to the management of protection policies. Its interface includes methods ensuring that component interactions are secured and that no component can bypass the framework to escape the protection policy. The elementary protection tools included in THINK library provides the system programmer with support to build a system resistant to unauthorized accesses and denial of service attacks. These tools are programmed to be policy-neutral which makes them completely flexible and reconfigurable at will. We believe our results can be applied to any system architecture based on a modular programming model implemented as a flexible software framework.

Some work remains to be conducted concerning the specification of protection policies. In our examples, policies are hard-coded in the security manager, which complicate their dynamic modification. It would be much easier to add and modify policies by using a constraint-based language like Prolog [2]. However, integrating a Prolog interpreter in an operating system is not feasible due to the performance costs it would induce. It would therefore be interesting to specify a constraint-based language which would ease the specification of protection policies without hindering the execution of the system.

References

[1] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight Remote Procedure Call. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 1989.

[2] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer, 1996.

[3] D. R. Engler, M. F. Kasshoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[4] J-Ph. Fassino, J-B. Stefani, J. Lawall, and G. Muller. THINK: A Software Framework for Component-based Operating System Kernels. In *Proceedings of the Usenix Annual Technical Conference*, 2002.

[5] B. W. Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, 1971.

[6] ODP Reference Model, Foundations, ITU-T ISO/IEC Recommendation X.902 International Standard 10746-2, 1995.

[7] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, 1993.

[8] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling Algorithms for Modern Disk Drives. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and Modeling of Computer Systems*, 1994.

[9] P. R. Zimmermann. *PGP: Source Code and Internals*. MIT Press, 1995.