Portable Resource Control in Java

The J-SEAL2 Approach[*]

Walter Binder
CoCo Software Engineering
Margaretenstr. 22/9
A-1040 Vienna
Austria
w.binder@coco.co.at

Jarle G. Hulaas
University of Geneva
rue Général Dufour 24
CH-1211 Geneva 4
Switzerland
hulaas@cui.unige.ch

Alex Villazón
University of Geneva
rue Général Dufour 24
CH-1211 Geneva 4
Switzerland
villazon@cui.unige.ch

ABSTRACT

Preventing abusive resource consumption is indispensable for all kinds of systems that execute untrusted mobile code, such as mobile object systems, extensible web servers, and web browsers. To implement the required defense mechanisms, some support for resource control must be available: accounting and limiting the usage of physical resources like CPU and memory, and of logical resources like threads. Java is the predominant implementation language for the kind of systems envisaged here, even though resource control is a missing feature on standard Java platforms. This paper describes the model and implementation mechanisms underlying the new resource-aware version of the J-SEAL2 mobile object kernel. Our fundamental objective is to achieve complete portability, and our approach is therefore based on Java bytecode transformations. Whereas resource control may be targeted towards the provision of quality of service or of usage-based billing, the focus of this paper is on security, and more specifically on prevention of denial-of-service attacks originating from hostile or poorly implemented mobile code.

Keywords

Bytecode rewriting, Java, micro-kernels, mobile object systems, resource control, security

1. INTRODUCTION

Java [17] was designed as a general-purpose programming language, with special emphasis on portability in order to enhance the support of distributed applications. Therefore, it is natural that access to low-level, highly machine-dependent mechanisms were not incorporated from the beginning. New classes of applications are however being conceived, which rely on the facilities offered by Java, and which at the same time push and uncover the limits of the language. These novel applications, based on the possibilities introduced by code mobility, open up traditional environments, move arbitrarily from machine to machine, execute concurrently, and compete for resources on devices where a very wide range of configurations can be found. We are therefore witnessing increased requirements regarding fairness and security, and it becomes indispensable to acquire a better understanding and grasp of low-level issues such as resource management.

Operating system kernels provide mechanisms to enforce resource limits for processes. The scheduler assigns processes to CPUs reflecting process priorities. Furthermore, only the kernel has access to all memory resources. Processes have to allocate memory regions from the kernel, which verifies that memory limits for the processes are not exceeded. Likewise, a mobile object kernel must prevent denial-of-service attacks, such as mobile objects allocating all available memory. For this purpose, accounting of physical resources (i.e., memory, CPU, network bandwidth, etc.) and of logical resources (i.e., number of threads, number of protection domains, etc.) is crucial.

Whereas J-SEAL2 [5, 6] is primarily designed for mobile objects, the approach described here is in many ways applicable to other distributed programming paradigms practiced in Java, since the mobile object paradigm is very comprehensive in terms of involved issues and technologies. The techniques employed in J-SEAL2 could thus greatly improve stability and security in the execution of Java Applets, or traditional distributed applications, where strong protection domains and resource control mechanisms are often needed. Further potential use cases include technologies such as World-Wide-Web server extensions (Java Servlets [27]) and Java application servers (e.g., Enterprise JavaBeans containers [25]).

The great value of resource control is that it is not restricted to serve as a base for implementing security mechanisms. Application service providers may e.g. need to guarantee a certain quality of service, or to create the support for usage-

[*]This work was funded by CoCo Software Engineering GmbH and the Swiss National Science Foundation grants 20-54014.98 and 5003-057753.

based billing, in order to amortize investments in hardware and software set at customers' disposal. The basic kernel extensions described here will be necessary to schedule the quality of service or to support the higher-level accounting system, which will bill the clients for consumed computing resources. This paper is however restricted to the kernel extensions that were necessary to add resource control to J-SEAL2; faithful to the micro-kernel approach, J-SEAL2 relegates to the higher levels the mechanisms which do not absolutely have to be part of the kernel.

This paper is organized as follows: The next section gives a brief overview of the J-SEAL2 mobile object kernel. Section 3 presents the design goals and the resulting resource control model, and section 4 the corresponding APIs. Section 5 explains our implementation techniques, for which section 6 presents some performance measurements. Section 7 compares our approach with related work, whereas section 8 concludes the paper.

2. THE J-SEAL2 MOBILE OBJECT KERNEL

This section gives some basic background on the J-SEAL2 mobile object kernel [5, 6], which we selected as the target platform to integrate our resource control model. For details regarding J-SEAL2, see the web pages at `http://www.jseal2.com/`.

J-SEAL2 is a micro-kernel implemented in pure Java, which supports the hierarchical process model of the Seal Calculus [33] that was first implemented by the JavaSeal mobile object system [9]. The J-SEAL2 kernel manages a tree hierarchy of nested protection domains[1], which may be either mobile objects or service components. Each mobile object and service executes in a protection domain of its own, called a sealed object or *seal* for short.

In J-SEAL2 mobile objects and service components are completely separated from each other. Untrusted mobile objects are not allowed to directly use certain functionality of the JDK, such as file or network IO, but they have to access dedicated J-SEAL2 services that are executing in separate protection domains. The inter-domain communication facilities of the kernel prevent direct sharing of object references between distinct domains. Details on the communication model of J-SEAL2 can be found in [5].

Figure 1 depicts a typical hierarchy of protection domains, including several service components, stationary sandbox domains that enforce appropriate security policies on their subdomains, as well as mobile objects. The root domain, *RootSeal*, is responsible for creating the service components as well as the stationary domains. The network service allows mobile objects to migrate to another site. In figure 1 one sandbox executes authenticated, fully trusted mobile objects, while the other one is responsible for anonymous, potentially malicious mobile objects. The sandbox of trusted mobile objects is granted access to all installed services, whereas the sandbox of anonymous mobile objects may only use the network service in a restricted way.

In J-SEAL2 each protection domain has associated its own set of threads, which cannot cross domain boundaries arbitrarily. Mobile objects are not allowed to directly create Java threads, but they have to use a safe wrapper class instead. The J-SEAL2 kernel enforces additional constraints on mobile objects, in order to ensure that a parent domain may terminate its children at any time, forcing the children to release all allocated resources immediately. For instance, mobile objects are not allowed to catch `ThreadDeath` exceptions, which are used by the kernel to stop running threads when a domain is terminated. Such restrictions are ensured by extended bytecode verification; details are discussed in [6].

So far, the J-SEAL2 mobile object kernel has provided essential functionality of an operating system kernel, such as protection, mediated communication, and safe domain termination. The kernel has however not been able to control resource allocation. In the next sections we present the design and implementation of a new resource control model for Java and its integration in J-SEAL2, which will complement J-SEAL2 to a complete Java operating system kernel. Therefore, a parent domain will act not only as a communication controller for its children, but also as a resource manager.

3. OBJECTIVES AND RESULTING MODEL

The ultimate objective of this work is to enable the creation of execution platforms where anonymous mobile objects, or more general programs, may securely coexist without harming each other, and without harming their environment. Examples of such platforms are user-extensible databases [16] or decentralized e-commerce and trading systems as e.g. in [18]. Java Applet execution platforms – World-Wide-Web browsers – as well as embedded Java devices also need such guarantees. The desire to deploy this kind of platforms translates into the following requirements:

- Accounting of low-level, physical resources as well as higher-level, logical resources, such as threads.
- Prevention against denial-of-service attacks that are based on CPU, memory, or communication misuse.
- Fair distribution of resources among concurrent domains, even outside the context of malicious activities.
- Sufficiently abstract concepts[2], in order to make mapping of policies into implementations more straightforward, and with a view to making resource control and eventual billing more manageable.

[1] In this paper the term 'protection domain' refers to the concept of a *process* or *task* in an operating system, and not to the JDK class `java.security.ProtectionDomain`.

[2] The abstractions presented in this paper are limited to the resources provided by the kernel, such as memory and CPU. An extensible high-level API addressing arbitrary resources will be provided on top of the resource control model of the kernel.

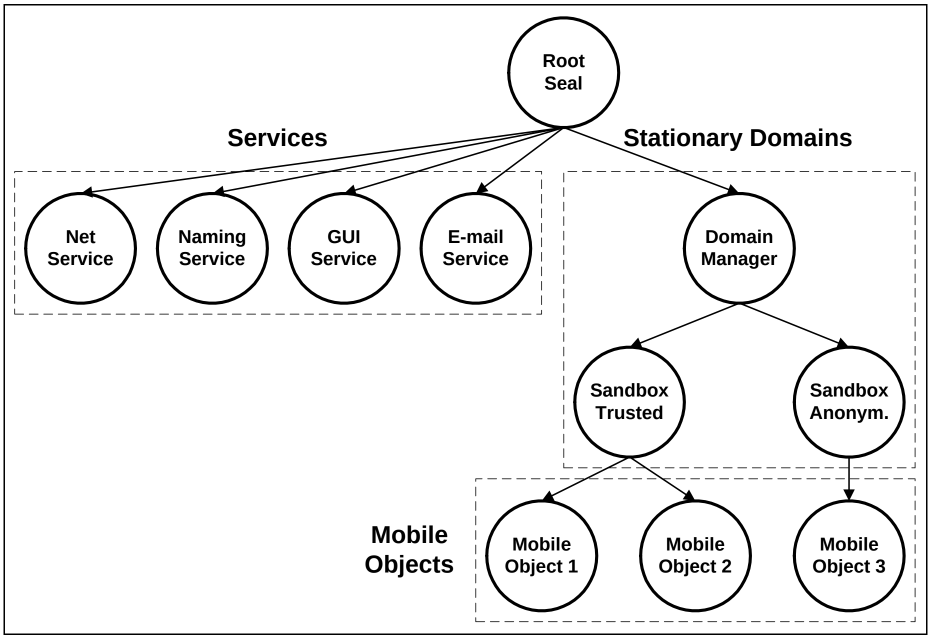

Figure 1: Nested protection domains in J-SEAL2.

- Fine-grained load-balancing of mobile object applications on a cluster of machines.

Since some aspects of resource control are to be manageable by the application developer, it is important that the general model integrates well with the existing J-SEAL2 programming model [5]. The resource control facilities shall reflect the hierarchical system structure. Hierarchical process models have been used successfully by operating system kernels, such as the Fluke micro-kernel [14]. The Fluke kernel employs a hierarchical scheduling protocol, CPU Inheritance Scheduling [15], in order to enforce scheduling policies. In this model, a parent domain donates a certain percentage of its own CPU resources to a child process. Initially, the root of the hierarchy possesses all CPU resources.

A general model for hierarchical resource control, such as e.g. Quantum [22], fits very well to the J-SEAL2 hierarchical domain model. At system startup the root domain, RootSeal, owns by default all resources the Java runtime system allocates from the underlying operating system, for example 100% CPU, the entire virtual memory, unlimited network usage, the maximum number of threads the underlying Java Virtual Machine (JVM) [21] is able to cope with, an unlimited number of subdomains, etc. Moreover, the root domain, along with the other domains loaded at platform startup, are considered as completely safe, and, consequently, no resource accounting will be enforced on them. This default behavior may however easily be overridden if specific configurations should require accounting even for trusted domains.

When a nested protection domain is created, the creator donates some part of its own resources to the new domain. Figure 2 illustrates the way resources are either shared or distributed inside a seal hierarchy. In the formal model of J-SEAL2, the Seal Calculus [33], the parent seal supervises all its subdomains, and inter-domain communication management was the main concern so far. Likewise, in the resource control model proposed here, the parent seal is responsible for the resource allocation with its subseals. This produces a nested structure, where the parent seal is initially the sole owner of its resources, and it may either share them or dispatch fractions of them to its subseals. However, the sum of all resources within a protection domain, e.g., in the *Untrusted application* of figure 2, remains constant.

Our resource control model stems from further design goals, such as portability and transparency: the next subsections are dedicated to describing these.

3.1 Portability and Transparency

Portability is crucial for the success of any mobile object platform. There are already some Java-based systems offering resource control facilities, such as Alta [32], GVM [4], KaffeOS [1, 2], etc. However, they rely on modified Java runtime systems, which are not portable. As a result, these systems are not suited for large-scale applications that have to support a wide variety of different hardware platforms and operating systems. Our goal is to provide a general-purpose model which is not dependent on specific implementation techniques, and to explore primarily completely portable solutions. This entails that we have to cope with certain restrictions and with performance levels sometimes inferior to those of existing realizations. Our portable approach will nevertheless show its advantages in the longer term: our solution will always perform somewhat slower than the fastest JVMs without resource control mechanisms, but, on the other hand, we will be able to exploit the latest techniques in Java implementation optimizations, which will

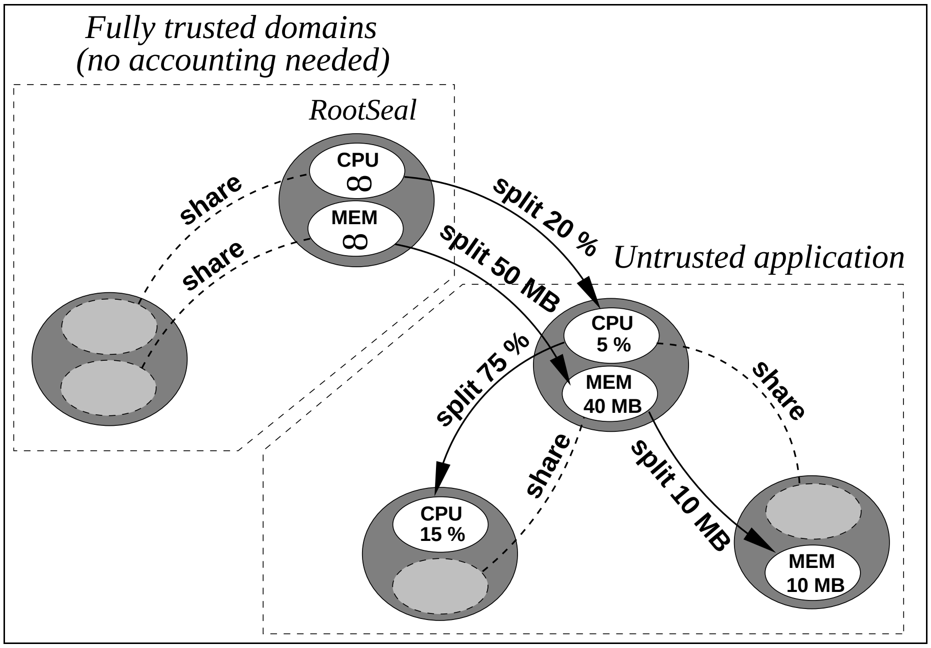

Figure 2: Illustration of the general resource control model.

often not be possible with non-portable implementations.

A related important requirement of our resource control model is that unmodified off-the-shelf applications should be able to execute on our platform. In other words, resource control must be transparent to applications which do not explicitly manage their pool of resources. In section 6.2 we discuss to which extent this wish has been satisfied.

For portability reasons, it should also be stressed that the goal of this work is not to implement any kind of real-time guarantee. The resources that are managed and distributed internally to the JVM are thus entirely dependent upon what the JVM process itself is given by the underlying operating system.

3.2 Minimal Overhead for Trusted Domains

Since J-SEAL2 is designed for large-scale applications, where a large number of services and mobile objects are executing concurrently, design and implementation must minimize the overhead of resource accounting. Some domains, such as core services, are fully trusted. Their resource consumption need not be controlled by the kernel.

3.3 Support for Resource Sharing

In certain situations protection domains that are neighbors in the hierarchy may choose to share some resources. In this case, resource limits are enforced together for a set of protection domains. As a result, resource fragmentation is minimized. For example, consider a mobile object creating a subdomain for a certain task. Frequently, the creating domain does not want to donate some resources to the subdomain, but it rather prefers to share its own resources with the subdomain. A property of our approach is that if a domain has unlimited access to a resource, this means that it is sharing it with RootSeal.

3.4 Managed Resources

Within each untrusted protection domain, the J-SEAL2 kernel shall account for the following resources:

- CPU_RELATIVE defines the relative share of CPU. It is expressed as a fraction of the parent domain's own relative share, but takes a slightly different meaning when the parent itself is a trusted domain; the precise semantics is exposed in section 4.2.
- MEM_ACTIVE is the highest amount of volatile memory that a protection domain is allowed to use at any given moment.
- THREADS_ACTIVE specifies the maximal number of active threads by protection domain at any moment. Uncontrolled creation of threads has to be avoided, as it results in increased load for the scheduler; it may even crash the JVM, as there is currently no standard Java construct allowing one to inquire about the maximum number of threads a JVM implementation is able to cope with.
- THREADS_TOTAL limits the number of threads that may be created throughout the lifetime of a protection domain, as thread creation is an expensive (kernel-level) operation.
- DOMAINS_ACTIVE specifies the maximal number of active subdomains a protection domain is allowed to have at any given moment. This limit is to minimize management overhead inside the kernel by controlling the complexity of the seal hierarchy at any time.

- DOMAINS_TOTAL bounds the number of subdomains that a protection domain may generate throughout its lifetime, as domain creation and termination are expensive kernel operations.

Note that the kernel of J-SEAL2 is not responsible for network control. This is because the micro-kernel does not provide access to the network. Instead, network access can be provided by multiple services. These network services or some mediation layers in the hierarchy are responsible for network accounting according to application-specific security policies. Let us stress that the network is not a special case, since J-SEAL2, thanks to its homogeneous model, may limit communication with any services, like e.g. file IO.

Another resource kind that could be expected in the above list of kernel-managed resources is the total amount of CPU allocated to a given protection domain throughout its lifetime. It is however not clear what the unit of measurement should be for this resource, while still preserving a completely hardware-independent model. The main objective of this kind of resource accounting would be to prevent applications from indefinitely cluttering up platforms; in a heterogeneous set of servers it gives however more sense to express total lifetime abstractly as the wall clock time elapsed since the application was started, than as the number of consumed CPU cycles. Using as unit of measurement the amount of executed Java bytecodes, although portable, was also regarded as too low-level. Measuring wall clock time can be achieved at the application level, by establishment of a controlling domain with sufficient rights to kill all misbehaving applications; this is a viable approach, since in J-SEAL2, when a parent disposes of a child seal, all resources are guaranteed to be freed properly. Accounting of total CPU time was therefore discarded from the kernel.

Finally, there is also no such resource as MEM_TOTAL, a limit to the accumulated amount of memory used throughout the lifetime of a protection domain. It could be needed to prevent the kind of denial-of-service attacks where a malicious domain creates a lot of dynamic objects in order to keep the CPU busy with garbage collection. Its implementation would however require maintenance of an additional counter, which we preferred to avoid. Instead, J-SEAL2 will take preventive action by charging an abstract amount of CPU as a compensation for the garbage collection induced by each object created.

The six basic resource types retained for management by the J-SEAL2 kernel are discussed in more detail in the API section below.

4. API

In this section we give an overview of the resource control API provided by the J-SEAL2 kernel. A detailed specification of the API can be found in [7].

There are 2 kernel abstractions dedicated to resource control: A resource object of type `Res` represents a resource of a certain type available for a protection domain. Resource sets of type `ResSet` ease the management of multiple resources. Furthermore, the kernel class `Seal`, which supports domain creation and termination, has been extended to allow a parent domain to restrict the resources of its children.

4.1 Definitions

In this section we provide some definitions, which simplify the description of the resource control API. In the following definitions let S denote an arbitrary domain in the hierarchy.

Root `Res` object: A root `Res` object of the domain S is a `Res` object responsible for resource control in S. A root `Res` object is returned by an invocation of the method `getCurrentRes` in class `Res` (for details see the following section).

Descendant `Res` object: A descendant `Res` object D of the domain S is the result of splitting a root `Res` object R of S. R is also called the parent `Res` object of D. When a descendant `Res` object is used in a `ResSet` object to create a nested domain, it will be used for resource control in the created child domain.

Note that these definitions are relative to the domain S. A descendant `Res` object D of the domain S is a root `Res` object in a child C of S, if D was in the `ResSet` object used for creating C. When we use the terms root and descendant `Res` objects in the description of a method, we implicitly assume `Res` objects of the domain invoking the method.

4.2 Class `Res`

For each type of resource, a protection domain has an associated root `Res` object reflecting how much of the resource the domain has been granted. A `Res` object defines a resource limit and provides information on the current resource usage in order to support resource aware computations. It offers an operation allowing a domain to split up some part of the resource. This operation yields a new descendant `Res` object that may be donated to children domains. The root domain, RootSeal, creates an initial `Res` object for each type of resource during startup. RootSeal distributes resources to service components and to application domains according to a configuration provided by the system administrator. Table 1 summarizes the interface of a `Res` object.

The static method `getCurrentRes` returns the root `Res` object for a given type of resource of the invoking domain. The constants `CPU_RELATIVE`, `MEM_ACTIVE`, `THREADS_ACTIVE`, `THREADS_TOTAL`, `DOMAINS_ACTIVE`, and `DOMAINS_TOTAL` (i.e., relative CPU share, active memory in bytes, as well as active and cumulative threads and subdomains) are used to indicate the requested resource type. The information, for which type of resource a `Res` object is responsible, is permanently associated with the `Res` object in order to prevent the programmer from mixing up different types of resources by mistake. The `getType` method returns the type of resource a `Res` object is representing.

`getLimit` returns the resource limit of a `Res` object. A negative value means that there is no resource limit. Concerning the semantics of the resource limit, the relative CPU share

Table 1: The `Res` API.

```
public final class Res {
  public static final int
    CPU_RELATIVE = 0,
    MEM_ACTIVE = 1,
    THREADS_ACTIVE = 2, THREADS_TOTAL = 3,
    DOMAINS_ACTIVE = 4, DOMAINS_TOTAL = 5;

  public static Res getCurrentRes(int type);
  public int getType();
  public long getLimit();
  public long getUsage();
  public Res split(long limit);
  public void setLimit(long limit);
  public void combine();
}
```

(`CPU_RELATIVE`) is treated differently from all other resource types. A relative CPU share of n means that domains created with the corresponding `Res` object may use at most a fraction of $\frac{n}{\text{sum of all CPU limits in the system}}$ of the CPU time available to domains with a CPU limit ≥ 0[3]. `getUsage` returns the resource consumption of all domains sharing the same root `Res` object. A negative value means that the J-SEAL2 kernel does not account for the resource.

As the `Res` API does not expose any public constructor, the `split` operation has to be used in order to create descendant `Res` objects that may be donated to subdomains. `split` may be invoked only on root `Res` objects. It returns a new descendant `Res` object responsible for the same type of resource as the root `Res` object, which becomes the parent of the descendant. The descendant `Res` object has the resource limit, which was passed to `split` as argument, and an initial resource usage of zero. The resource usage of the parent `Res` object is incremented by the limit given to the descendant.

The `setLimit` method provides a mechanism to modify the resource limit of a `Res` object. The new resource limit is passed as argument. The resource usage of the parent `Res` object is adjusted accordingly. A parent domain may use descendant `Res` objects in order to monitor the resource usage of children domains. With the aid of `setLimit`, the parent is able to adjust the resource limits for the children domains.

The `combine` operation allows to merge `Res` objects that have been split before. If it is invoked on a root `Res` object, `combine` has no effect. If it is called on a descendant `Res` object, the descendant is combined with its parent `Res` object, i.e., the resource usage of the parent object (if it is accounted for) is reduced by the limit of the descendant. The descendant `Res` object is marked as invalid and cannot be used anymore. Combination is only possible, if the descendant `Res` object is not used by any subdomain (i.e., all subdomain created with the descendant `Res` object must be terminated before).

Table 2: The `ResSet` API.

```
public final class ResSet {
  public static ResSet getCurrentResSet();
  public ResSet copy();
  public Res getRes(int type);
  public void setRes(Res r);
  public void combine();
}
```

4.3 Class `ResSet`

A `ResSet` object offers a convenient way to manage all resources given to a domain. It holds exactly one `Res` object for each type of resource. Table 2 summarizes the public interface of a `ResSet` object:

The static method `getCurrentResSet` returns a `ResSet` object with the root `Res` objects of the domain the calling thread is executing in. This `ResSet` object may be used to access the individual `Res` objects of the domain. The `copy` method creates a shallow copy of a `ResSet` object. The copy contains the same references to `Res` objects as the original `ResSet` object. The `getCurrentResSet` and `copy` methods are the only mechanisms allowing to allocate new `ResSet` objects. There is no public constructor, because the API enforces the constraint that a `ResSet` always holds exactly one `Res` object for each type of resource.

The `getRes` method return the `Res` object for a given type of resource. The argument is a resource constant defined in the class `Res`. The `setRes` method replaces the `Res` object in the set, which has the same resource type as the `Res` object given as argument. The `combine` method offers a convenient way to invoke `combine` on all `Res` objects in the set.

4.4 Class `Seal`

The `Seal` abstraction provides methods for domain creation (unwrapping) and removal (wrapping or disposing). Table 3 summarizes the `unwrap` methods of the `Seal` class. Other methods are not shown, because they are not affected by the resource control extension.

The `unwrap` method with 3 arguments requires a wrapped representation of the subdomain to create (corresponding to the serialized state of a mobile object), its name, as well as a `ResSet` object with the resources for the new subdomain. The `unwrap` operation with 2 arguments implicitly shares the resources of the unwrapping domain with the created child domain.

When a domain is created, the parent's `DOMAINS_ACTIVE` and

[3] In our current implementation, this resource is controlled by periodic sampling of the amount of executed bytecode instructions. The precision of the measurement is implementation dependent; there is indeed a bias induced by the fact that the CPU resource is not allocated by absolute values, but by relative shares, while in the implementation, the reference value is the aggregated consumption measured among untrusted domains and is not, as could be expected, the resource taken as a whole.

Table 3: The `unwrap` methods of class `Seal`.

```
public class Seal {
  public static void unwrap(WrappedSeal wrapped,
                            String sealname,
                            ResSet resources);
  public static void unwrap(WrappedSeal wrapped,
                            String sealname) {
    unwrap(wrapped, sealname,
           ResSet.getCurrentResSet());
  }
  ...
}
```

Table 4: Resource control example.

```
long MB = 1024*1024;

ResSet rP = ResSet.getCurrentResSet();
Res cpu = rP.getRes(Res.CPU_RELATIVE);
Res mem = rP.getRes(Res.MEM_ACTIVE);

ResSet rA = rP.copy();
long cpuA = (long)(cpu.getLimit()*0.75);
rA.setRes(cpu.split(cpuA));

ResSet rB = rP.copy();
rB.setRes(mem.split(10*MB));

Seal.unwrap(childA, nameOfChildA, rA);
Seal.unwrap(childB, nameOfChildB, rB);
```

`DOMAINS_TOTAL` `Res` objects are charged for the created subdomain(s), while the child's resource objects are charged for the CPU time consumed for unwrapping (involving classloading and linking), for memory allocation, as well as for the child's initializer thread.

4.5 Example

The code fragment in table 4 demonstrates how the resource control API is used to control the resources of children domains. This example corresponds to the *Untrusted application* depicted in figure 2.

A parent domain, which has limited CPU and memory resources, creates 2 subdomains: One child domain (*childA*) gets 75% of the parent's CPU resources and shares the memory resources with the parent, while the other child domain (*childB*) receives 10 MB of active memory and shares the CPU resources with the parent.

5. IMPLEMENTATION

In this section we present the techniques we are using for the implementation of the resource control model discussed in the previous sections. Since accounting for high-level resources, such as active and cumulative threads and subdomains, requires only minor modifications to a few J-SEAL2 kernel primitives, we focus on accounting for physical resources, such as memory and CPU consumption.

5.1 No Direct Sharing

Since its initial release the J-SEAL2 kernel is designed to ease the integration of resource control facilities. It guarantees accountability, i.e., user-visible objects belong to exactly one protection domain. References to an object exist only within a single domain[4], i.e., in J-SEAL2 there is no direct sharing of object references between distinct domains. Therefore, it is possible to account each allocated object to exactly one protection domain. This feature not only simplifies resource accounting, but it is also crucial for immediate resource reclamation during domain termination.

5.2 Bytecode Rewriting

In our approach we employ bytecode rewriting techniques both for memory and CPU accounting. This is because it is to our understanding the only entirely portable way to implement the needed accounting mechanisms. It is unrealistic to expect the source code of every application to be available for modification. Moreover, if we want guarantees against denial-of-service attacks, we cannot rely on foreign code to perform any voluntary self-limiting operations, whereas if we modify its bytecode before it starts executing, we can 'oblige' it to provide any information needed by the kernel and to obey any restriction imposed on it by the environment. Instead of rewriting bytecode for CPU control, the J-SEAL2 kernel might e.g. ask the underlying operating system for information about the CPU consumption of each thread, but this is possible only when Java threads are directly mapped into operating system threads. Another approach would be to run a modified JVM; the arguments against this are however exposed elsewhere in this paper. A further discussion of existing (and non-portable) approaches is to be found in section 7.1.

In the present paper, the bytecode of a Java class is modified before it is loaded by the JVM [21]. Code for memory accounting is inserted before each memory allocation instruction (for details, see section 5.7). CPU accounting uses an abstract measure, the number of executed bytecode instructions. Therefore, code for CPU accounting is inserted in every basic block of code (details are presented in section 5.8).

Rewriting for memory accounting has to be done before rewriting for CPU accounting, because memory accounting inserts additional bytecode instructions to enforce memory limits, while accounting for CPU consumption does not involve any object allocation.

[4] The only exception to this rule are `Res` objects (see section 4.2) used for resource sharing. The parent domain is charged for the `Res` objects of its children. `Res` objects cannot be communicated between domains apart from domain creation. Because `Res` objects are small and the number of `Res` objects donated to a child is limited, this minor inexactness is irrelevant.

5.3 Domain Types

The resource control model supports trusted domains that have unlimited access to certain types of resources. For performance reasons, the J-SEAL2 kernel does not account for the consumption of these resources. Regarding CPU and memory accounting, we distinguish 4 types of domains:

NO-ACC: Domains without memory limit and without CPU control may execute unmodified Java code, as they do not need to execute any accounting instructions.

CPU-ACC: Domains without a memory limit, but with CPU control have to execute CPU accounting instructions. However, code for memory accounting is not required in such domains.

MEM-ACC: Domains with a memory limit, but without CPU control have to execute memory accounting instructions. However, code for CPU accounting is not required in such domains.

CPU-MEM-ACC: Domains with a memory limit and with CPU control have to execute accounting code for memory allocation as well as for CPU consumption.

5.4 Accounting Objects

In MEM-ACC and in CPU-MEM-ACC domains objects of the class `MemAccount` represent memory limit and current usage. In CPU-ACC and in CPU-MEM-ACC domains objects of the class `CPUAccount` maintain CPU consumption. These objects are used only by the J-SEAL2 kernel, they are not accessible by user code. Each thread has associated the `MemAccount` object and a `CPUAccount` object of the domain it is executing in; `null` values indicate that a domain does not need a `MemAccount` or `CPUAccount` object. Java thread-local variables (instances of the class `java.lang.ThreadLocal`) are used to implement this association. The `MemAccount` and `CPUAccount` implementations provide the static method `getCurrentAccount`, which returns the corresponding accounting object of the domain the calling thread is executing in.

Because access to `MemAccount` and above all to `CPUAccount` objects may be extremely frequent, accessing these objects from thread-local variables in every method would cause a significant performance penalty[5]. Therefore, non-native methods are rewritten in order to pass the necessary accounting objects as additional arguments. Native methods are excluded from rewriting, because we cannot account for memory allocated and CPU time consumed by native code. We are relying on modern inter-modular register allocation algorithms implemented by state-of-the-art JVMs to minimize the overhead of passing the accounting objects through the whole method call-graph.

As an example for the rewriting process, consider method `a` given in table 5. The rewritten[6] version of method `a` for a CPU-MEM-ACC domain is given in table 6. Here we are only presenting the additional arguments, while the inserted accounting code is discussed in sections 5.7 and 5.8. In this example, method `a` receives two additional arguments for the `CPUAccount` and `MemAccount` objects[7]. The additional arguments are passed to all invoked methods (resp. constructors).

Table 5: Method a before rewriting.

```
void a(int x) {
  b(null, x);
}
```

Table 6: Method a rewritten for a CPU-MEM-ACC domain.

```
void a(int x, MemAccount mem, CPUAccount cpu) {
  b(null, x, mem, cpu);
}
```

5.5 Callbacks from Native Code

Native code invoking Java methods complicates the resource control implementation, because the native code is not aware of the accounting objects to be passed to Java methods as extra arguments. The following three scenarios of Java method invocation by native code are particularly important:

- Thread creation: The Java runtime system (native code) invokes the `run` method of a thread object when a thread is started with the aid of the `start` method.
- Static initializers: Static initializers are invoked directly during class-loading, i.e., they are invoked by native code.
- Reflection: The methods `invoke` resp. `newInstance` of Java's reflection classes `Method` resp. `Constructor` are native.

When the thread invoking a Java method from native code has already set its thread-local accounting objects, it is sufficient to provide for each method an additional one with the same signature, which takes the required accounting objects from thread-local variables and passes them to the rewritten method. In the rewriting example given in tables 5 and 6 we have to supplement the rewritten method with method `a` in table 7. Note that when a constructor is rewritten according to this scheme, the invocation of another constructor of the same class or of a constructor of the superclass has to antecede the lookup of the accounting objects.

[5] In Sun's JDK 1.3 implementation thread-local variables are managed as hash-maps, i.e., each access to a thread-local variable requires a hash-map lookup.

[6] For the sake of easy readability, we present rewriting transformations at the Java level, even though the implementation works at the JVM bytecode level.

[7] Note that in a CPU-ACC or MEM-ACC domain only one additional argument would be necessary to hold the accounting object.

Table 7: Solving callbacks from native code.

```
void a(int x) {
  MemAccount mem = MemAccount.getCurrentAccount();
  CPUAccount cpu = CPUAccount.getCurrentAccount();
  a(x, mem, cpu);
}
```

When a new thread starts executing its `run` method, the thread-local accounting objects have not been initialized yet. As protection domains in J-SEAL2 do not have direct access to the class `java.lang.Thread` but have to employ a safe wrapper class instead [6], the wrapper initializes the thread-local accounting variables with the accounting objects of the protection domain the new thread belongs to. These objects are passed to the constructor of the wrapper by the J-SEAL2 kernel.

When a new protection domain is created, the J-SEAL2 kernel allocates a new initializer thread with the accounting objects for the new domain. While starting this thread, the thread wrapper initializes the thread-local accounting variables and starts to load the classes of the new protection domain. The class-loading already happens in the accounting context of the new domain.

5.6 Class-loading

The J-SEAL2 kernel distinguishes between shared and replicated classes [6]. Shared classes are loaded by the system class-loader (they exist only once in the JVM), while replicated classes, such as the classes of a mobile object, are loaded by the class-loader of a protection domain (they are reloaded in each domain). All JDK classes[8] as well as most classes from the J-SEAL2 kernel are shared. Certain J-SEAL2 library classes that are frequently used may be shared as well, in order to avoid the overhead of reloading them multiple times.

Since a shared class may be referenced by fully trusted domains (no accounting necessary) as well as by untrusted domains (resource control required), it has to provide multiple different versions of each method. The version without resource control corresponds to the unmodified code, while the versions used by untrusted domains take the accounting objects as extra arguments and include the necessary accounting instructions. Optimizations to reduce the code size of rewritten shared classes are presented in [7].

Shared classes are rewritten off-line (e.g., during the installation of the J-SEAL2 platform), because we cannot modify the system-classloader, which is part of the Java runtime system, in a portable way. Replicated classes are rewritten on-line, immediately before they are linked into the JVM. Therefore, a mobile object may execute unmodified code on a J-SEAL2 platform where it is trusted, while on another J-SEAL2 installation the code of the same mobile object may be rewritten for resource control.

Table 8: Rewriting methods in shared classes.

```
void a(int x) {
  MemAccount mem = MemAccount.getCurrentAccount();
  CPUAccount cpu = CPUAccount.getCurrentAccount();
  if (cpu == null)
    if (mem == null) a(x, (NoAccount)null);
    else a(x, mem);
  else
    if (mem == null) a(x, cpu);
    else a(x, mem, cpu);
}
void a(int x, NoAccount _no) { b(null, x, _no); }
void a(int x, CPUAccount cpu) { b(null, x, cpu); }
void a(int x, MemAccount mem) { b(null, x, mem); }
void a(int x, MemAccount mem, CPUAccount cpu) {
  b(null, x, mem, cpu);
}
```

The example in table 8 shows how method `a` given in table 5 would be rewritten, if it was defined in a shared class. A method with the same signature as the original method dispatches to the appropriate implementation, when it is invoked from native code. For each type of domain, there is a different method implementation. In this example we distinguished the signature of the NO-ACC implementation from the dispatcher method by adding a dummy argument of type `NoAccount`. The compilers of state-of-the-art JVMs may be able to remove this useless argument.

Alternatively, it is possible to rename the NO-ACC implementation. This approach complicates rewriting, since a table of renamed methods of shared classes has to be maintained, but it has the advantage that replicated classes of trusted domains (e.g., classes of an authenticated, fully trusted mobile object) can be rewritten very efficiently, because only method signatures in the constant-pool [21] are affected, whereas the method code remains unchanged (in contrast, passing the extra `NoAccount` argument requires additional bytecode instructions).

5.7 Memory Control

Memory control has to limit the allocation of heap memory, as well as the size of the execution stacks of running threads.

5.7.1 Heap

Enforcing memory limits requires exact pre-accounting for memory resources, i.e., an overuse exception is raised before a thread can exceed the memory limit of the domain it is executing in. In contrast to JRes [12], which maintains a separate memory limit for each thread, J-SEAL2 enforces a single memory limit for a multithreaded domain or even for a set of domains in the case of resource sharing.

Because a single `MemAccount` object has to maintain the memory consumption and limit of a set of domains sharing the same memory resources, access to the `MemAccount` must be synchronized. Furthermore, accounting for an object as well as its allocation and initialization has to be an

[8] It is not possible to load a JDK class with a loader different from the system class-loader.

atomic action.

Before the object is allocated, J-SEAL2 ensures that the memory limit is not exceeded and updates the MemAccount. If the memory allocation fails, if the constructor raises an exception, or if the allocating thread is terminated asynchronously, we have to ensure that the modification of the MemAccount is undone. Otherwise, other threads or even other domains (using the same MemAccount) could suffer from memory leakage. Details on the rewriting scheme for memory allocation instructions can be found in [7].

When the garbage collector reclaims an object, we have to update the MemAccount that has been charged for this object. For this reason, the MemAccount maintains a weak reference for each allocated object, which does not prevent the object from being reclaimed. When an object referenced by a weak reference is garbage collected, the weak reference is enqueued in a reference queue, which can be polled by the MemAccount implementation (for details see [7]).

5.7.1.1 Object Size

The size of an object is calculated from the number of fields for each Java basic type, the number of fields holding object references, a constant for the object overhead, as well as a constant for the accounting overhead (i.e., the overhead for maintaining a weak reference to the allocated object). For arrays, the actual size must be computed from the array dimensions available on the execution stack. Depending on the Java runtime system, the overhead for array objects may be larger than for non-array objects, because of the size information stored within arrays.

Constants for the object overhead and for the size of Java basic types and object references are managed in a configuration file by the system administrator. Since in general the administrator does not know the object representation of the underlying Java runtime system, a tool helps to approximate these constants (e.g., by avoiding garbage collection and measuring the difference of allocated memory before and after creating certain types of objects). However, object alignment is not taken into account.

5.7.1.2 Optimizations

While our approach works for objects as well as for arrays, we are also implementing an optimization for non-array objects: Similar to JRes [12], in each allocated object we store a reference to the corresponding MemAccount object. Rewritten finalizers are responsible for updating the MemAccount when an object is reclaimed by the garbage collector. Thus, we can avoid the significant overhead of maintaining weak references, which is particularly important for small objects.

For arrays, such an optimization cannot be implemented in pure Java. However, in practice the overhead for accounting for allocated arrays is not a serious problem, because arrays frequently are large objects (compared to the accounting overhead they cause).

5.7.2 Stack

The computation of recursive methods may rapidly blow up the execution stack of a thread without allocating a single object. Especially if domains are allowed to create large numbers of threads, an attacker could easily create a bunch of threads, and in each thread create a very deep call stack forcing the system to use large amounts of memory (precious memory, which cannot be garbage collected until the methods return).

Most proposals for resource control in Java, like e.g. JRes [12], do not take the memory consumption of the execution stacks into account. Our implementation supports control of stack memory as an optional feature. During the installation, the system administrator has to decide whether stack control shall be enabled. When untrusted domains are allowed to create only a small number of threads and the underlying JVM allocates execution stacks that cannot expand dynamically, it is sufficient to charge the MemAccount for the maximum stack size[9] when a thread is created.

However, if the JVM allows execution stacks to grow up significantly, special effort is necessary in order to limit the size of the stack. For this purpose, we rewrite non-native methods to pass an additional counter, indicating the amount of memory the thread is allowed to use on the stack. On method entry, this counter has to be reduced by the number of local variables and the maximum stack consumption of the invoked method[10]. For each method, this information is available in the Java class-file [21]. If the counter becomes negative, an appropriate exception is raised. The counter can be an integer that is passed by value. Therefore, a good register allocator will help to keep the overhead small. As a further optimization, leaf methods (i.e., methods that do not invoke any other method) may omit the check of the counter.

5.8 CPU Control

For CPU control, we are accounting the number of executed bytecode instructions for each thread running in a CPU-ACC or CPU-MEM-ACC domain. A high-priority scheduler thread, which is part of the J-SEAL2 kernel, executes periodically in order to ensure that assigned CPU limits are respected. The scheduler thread calculates the number of executed bytecode instructions for each set of domains sharing a CPU limit by summing up the CPU consumption of all threads executing in a domain in the set. The scheduler compares the number of executed bytecodes with the desired schedule. If a set of domains has exceeded its CPU limit, the priorities of threads executing in these domains are lowered.

[9] In order to approximately determine the maximum stack size of a JVM implementation, we employ a calibration program executing a recursive method until a StackOverflowError occurs. The maximum stack size corresponds to the product of the maximum recursion depth and the size of a stack frame of the recursive method.

[10] A Just-in-Time compiler will completely remove the Java stack when it creates code for a register machine. Nevertheless, the number of local variables and the maximum stack consumption of a method can be used as an approximation for the size of a stack frame of the method.

5.8.1 Class `CPUAccount`

In contrast to a `MemAccount` object, which is shared by all threads executing in a domain with memory accounting, each thread running in a domain with CPU accounting has associated its own `CPUAccount` object. Since CPU accounting occurs very frequently, it is important that multiple threads do not have to synchronize on a common accounting object. As only the scheduler thread makes any scheduling decisions, it is sufficient to account for each thread separately. The scheduler is responsible for accumulating the accounting data of all threads executing in a set of domains sharing a CPU limit.

A `CPUAccount` object simply maintains an integer counter, which is updated by the thread owning the object. Table 9 shows some parts of the `CPUAccount` implementation[11]. Because the scheduler thread has to read the counter value, we are using a volatile variable in order to force the JVM to immediately propagate every update from the working memory of a thread to the master copy in the main memory [17, 21].

Table 9: The `CPUAccount` implementation.

```
public final class CPUAccount {
   public volatile int usage;
   ...
}
```

In general, updating the counter requires loading the `usage` field of the `CPUAccount` object from memory (it is volatile), incrementing the loaded value accordingly, and storing the new value in the memory. A counter update requires about 6 bytecode instructions.

5.8.2 Scheduler

In this section we describe how the scheduler thread computes the CPU consumption of a set of domains, and how it employs different JVM priority levels in order to prevent CPU overuse. However, we do not present a particular scheduling algorithm, because we are still experimenting with different policies.

For each `CPUAccount` object, the scheduler thread always stores the value of the counter it has read most recently. The scheduler calculates the difference between the current value and the previously stored value in order to determine the amount of bytecode instructions executed during the last time-slice (because of the lack of synchronization, the scheduler must not reset any `CPUAccount` object). If a thread has not existed before, the scheduler assumes the previously stored value to be zero. When a thread terminates, its `CPUAccount` object is not disposed of immediately, but it is maintained until the scheduler has examined it.

The scheduler has to deal with an overflow in the counter of a `CPUAccount` object. The size of the counter must be large enough so that its full range cannot be used in a single time-slice. For current JVMs and a reasonably small time-slice, a Java `int` is sufficient. However, in future high-performance systems, `CPUAccount` objects may have to maintain `long` values[12].

We are using different JVM priority levels to control the CPU consumption of individual domains. As protection domains in J-SEAL2 do not have direct access to the class `java.lang.Thread` (they have to use a safe wrapper class instead [6], which does not offer any mechanism to change the priority of a thread), an user-level thread cannot raise its own priority.

Even though the Java language specification [17] does not define any scheduling policy, current JVM implementations respect assigned thread priorities. Many JVMs employ fixed priority scheduling, where a low-priority thread cannot execute, if there is a high-priority thread ready to run. The J-SEAL2 kernel uses the distinct JVM thread priority levels as follows:

- MAX_PRIORITY: The maximum priority is reserved to JVM internal tasks, such as handling weak references. J-SEAL2 does not run any threads with the maximum priority.
- MAX_PRIORITY-1: J-SEAL2 uses this priority level for kernel-level operations in order to prevent priority inversion, i.e., when a high-priority thread is waiting for an exclusive kernel lock (see [6]) because of a low-priority thread T executing in kernel mode, the priority of T is temporarily boosted until thread T releases the kernel lock.
- MAX_PRIORITY-2: This priority level is used by the J-SEAL2 scheduler thread.
- NORM_PRIORITY–MIN_PRIORITY[13]: The scheduler assigns these priority levels to threads according to the CPU consumption of the corresponding domain and the assigned CPU share. Threads executing in NO-ACC or in MEM-ACC domains are always assigned NORM_PRIORITY. If a domain exceeds its CPU limit, the priorities of its threads are reduced (or at least the priorities of those threads overusing the CPU). If a domain does not consume its assigned CPU resources, the priorities of its threads may be increased again (but never exceeding NORM_PRIORITY). We are experimenting with different scheduling algorithms regarding the history of CPU consumption.

5.8.3 Rewriting Algorithm

In the description of the rewriting algorithm we use the following definition of an accounting block, which is related to the concept of a basic block of code. In order to minimize

[11] For instance, we omitted the static `getCurrentAccount` method mentioned in section 5.4.

[12] For a `long` variable, the volatile declaration is crucial, because some JVMs do not treat non-volatile `long` values atomically [21].

[13] In this description we assume that NORM_PRIORITY < MAX_PRIORITY-2.

the accounting overhead, we are considering blocks of maximal length. An accounting block is a bytecode sequence fulfilling the following constraints:

- If a bytecode instruction, which is neither a method (resp. constructor) invocation nor a JVM subroutine invocation, changes the control-flow non-sequentially (e.g., method return, exception raising, branch, JVM subroutine return, etc.), it must be the last instruction in the accounting block. That is, with the exception of method (resp. constructor) and JVM subroutine invocations, only the last bytecode instruction in the block may change the control-flow non-sequentially. A method invocation does not terminate an accounting block, because otherwise the average block size would be reduced significantly, as method invocations are very frequent in object-oriented programs.
- Only branches to the begin of the block are allowed. There is no bytecode instruction branching to another instruction in the same method, which is not the first one in its block. Furthermore, the first instruction of an exception handler must be always the first instruction in its block.

The bytecode rewriting algorithm involves the following 4 steps (an efficient implementation may perform multiple steps together):

1. Method (resp. constructor) invocations are rewritten in order to pass the `CPUAccount` object as extra argument. Because the `CPUAccount` is always the last argument[14], it can be pushed onto the stack immediately before the method (resp. constructor) invocation instruction.
2. An accounting block analysis (similar to a basic block analysis in traditional compilers) partitions the method code into a set of accounting blocks. Each block has an attribute indicating the accounting size of the block. Initially, this attribute holds the number of bytecode instructions in the block[15]. Furthermore, a control-flow graph with the accounting blocks as nodes has to be constructed, if optimizations are to be performed in order to minimize the accounting overhead. Without any optimizations, accounting instructions have to be inserted into every block.
3. Optimizations analyze the control-flow graph in order to detect situations where accounting for multiple different blocks may be combined. The optimizations may decrement the accounting size attribute of one block and add it to the accounting size of another block. If the accounting size of a block becomes zero, it does not require any accounting instructions. Details concerning optimizations are presented in [7].
4. For every block with a positive accounting size, accounting instructions are inserted at the begin of the block. The only exception to this rule is the first block in a constructor: The invocation of another constructor of the same class or of the superclass has to antecede the accounting code. The included instructions add the accounting size of the block plus the number of inserted accounting instructions to the `CPUAccount` object. For performance reasons, updates of the `CPUAccount` object are not synchronized.

This approach ensures that a thread is charged for at least the number of bytecode instructions it executes. For each accounting block, a thread is charged for the number of instructions in the block, before it executes these instructions (pre-accounting). When an instruction, which is not the last one in its accounting block, raises an exception, the thread has been charged for more instructions than it has consumed. However, since the number of executed bytecode instructions is only an approximation of the exact CPU consumption, and because exception handling is expensive on many JVM implementations, this possible inexactness does not pose any problem.

5.9 Accounting for Garbage Collection

In order to prevent denial-of-service attacks by causing the garbage collector to consume a considerable amount of CPU time (e.g., an attacker may create a lot of garbage without exceeding its memory limit), the J-SEAL2 kernel has to account for the time spent by the garbage collector. Only CPU-MEM-ACC domains can be charged for the garbage they produce, because accounting for garbage collections requires the information, which domain has allocated a certain object (such information is not available in NO-ACC or CPU-ACC domains), and because the time spent by the garbage collector affects the CPU consumption of a domain (CPU consumption is not measured in NO-ACC or MEM-ACC domains).

Since the exact CPU time spent by the garbage collector is not known, we are using an abstract measure. The J-SEAL2 administrator defines a rough approximation of the number of bytecode instructions required to reclaim an object. Before an object is allocated, the J-SEAL2 kernel charges the `CPUAccount` object of the allocating thread. That is, a domain has to 'pay' for the garbage it eventually will produce at the time it 'buys' an object. This simple approach has the advantage that a CPU-MEM-ACC domain is charged for all garbage it produces, even if the domain has already terminated when some objects are reclaimed.

5.10 Compensating for Native Code

With the aid of bytecode rewriting techniques, it is not possible to account for memory allocation and CPU consumption in native code. Untrusted applications are not allowed

[14] Since rewriting for memory accounting is done before rewriting for CPU control, the `MemAccount` argument is passed always before the `CPUAccount` object.

[15] In order to improve the accuracy of measurement, the J-SEAL2 administrator may configure a weighting of bytecode instructions (integer values) according to their complexity. To simplify matters, we assume that all bytecode instructions have a weighting of 1.

to bring native code libraries into the system. Concerning JVM-provided standard operations, the J-SEAL2 kernel tries to compensate for resources used by native code and prevents untrusted domains from using certain functionality leading to a significant resource consumption by native code. In the following we describe some important cases of resource consumption in native code and how J-SEAL2 solves them:

- Class-loading: The Java runtime system manages an internal table of loaded classes. Memory for compiled methods is allocated by the Just-in-Time compiler, which is usually implemented in native code. However, the set of classes untrusted domains (e.g., mobile objects) are allowed to access is limited and known to the J-SEAL2 kernel. Therefore, the kernel accounts for the classes using an approximation, which is proportional to the size of the class-files.
- Deserialization: J-SEAL2 uses Java serialization in order to create messages to be transferred across domain boundaries. When the receiving domain opens a message, it is being deserialized using the class-loader of the receiving domain to resolve class names. Deserialization requires native methods to allocate objects without invoking their constructors. J-SEAL2 solves this hurdle by storing the amount of objects for each type, which is part of the serialized object graph, in the message. The receiver performs resource checks before deserializing the message.
- Object cloning: Java supports object cloning to create shallow copies of objects. The shallow copy is allocated by a native method. A simple solution is to forbid untrusted domains to clone objects.
- Reflection: The Java reflection API provides a mechanism to indirectly create a new instance of a class. The object is allocated by native code. J-SEAL2 simply prevents untrusted domains from using the reflection API.

6. EVALUATION

This section assesses our approach, first by presenting the performance achieved with our current implementation, then by discussing the limitations.

6.1 Measurements

While in J-SEAL2 the overhead for memory control is comparable to the overhead caused by JRes[16] [12], the overhead of CPU control based on bytecode rewriting techniques has to be examined carefully, because such an approach has not been used before. In this section we present performance measurements proofing that the overhead due to our completely portable implementation of CPU accounting is acceptable on modern JVM implementations[17].

We have implemented a bytecode rewriting tool that performs the necessary transformations of Java classes to support resource control. The tool was designed to add resource accounting instructions into arbitrary Java applications, to create an extended version of the JDK, and to modify mobile object applications in J-SEAL2. Our current bytecode rewriting tool supports off-line transformations of arbitrary Java classes. However, we are integrating the resource control mechanism in the J-SEAL2 kernel, which requires also load-time rewriting of mobile objects. Detailed results of our work will be published on our web pages at `http://abone.unige.ch/`.

There are several low-level bytecode engineering frameworks written in Java (e.g., BCA [19], JOIE [11], BIT [20]), as well as higher-level frameworks, such as e.g. Javassist [10]. Our bytecode rewriting tool is based on BCEL (Byte Code Engineering Library, formerly called JavaClass) [13], which allows bytecode manipulations of Java classes and is also entirely written in Java. We chose BCEL since it is one of the most mature bytecode instrumentation frameworks and provides a powerful and intuitive API that is well adapted for our requirements.

We measured the standard SPEC JVM98 benchmarks [30] on a Linux platform (Athlon AMD 1200MHz clock rate, 256MB RAM, Linux kernel 2.4.2) with IBM's JDK 1.3 implementation, which includes one of the best Just-in-Time compilers available today. We measured the overhead due to CPU accounting in three different configurations:

- U_{bench}-U_{jdk}: Unmodified benchmarks on an unmodified JDK.
- R_{bench}-U_{jdk}: Rewritten benchmarks on an unmodified JDK.
- R_{bench}-R_{jdk}: Rewritten benchmarks on a rewritten JDK[18].

For each measurement, table 10 shows the execution time of the benchmark in seconds (rounded to 3 decimal places), as well as the speedup of the original code compared to the rewritten version (rounded to 2 decimal places). In order to minimize the impact of compilation and garbage collection, all results represent the median of 101 different measurements. Furthermore, we also computed the geometric mean for each configuration. We rewrote about 520 Java class-files for the CPU-aware version of SPEC JVM98 benchmarks, and about 5400 class-files for the extended version of the JDK.

The results in table 10 show that the overhead due to CPU accounting is about 25%, if we rewrite applications as well as the whole JDK. With an unmodified JDK, the overhead can be almost halved (this configuration is not useful for resource control, but it helps to extract the overhead due

[16] For an application allocating a new object every 250 bytecode instructions, the overhead for memory control is less than 18%, if no memory limit is exceeded.

[17] We are not measuring the overhead for CPU control incurred by the scheduler, as it can always be kept small by choosing an appropriate time-slice.

[18] Modern JVMs allow to run user-defined library classes with the `-Xbootclasspath` option.

Table 10: Benchmarks measuring the overhead of CPU accounting (time in seconds).

Benchmark	U_{bench}-U_{jdk}		R_{bench}-U_{jdk}		R_{bench}-R_{jdk}	
_227_mtrt	3,783	(1,00)	4,517	(1,19)	4,788	(1,27)
_202_jess	5,299	(1,00)	5,842	(1,10)	6,415	(1,21)
_201_compress	11,605	(1,00)	13,344	(1,15)	13,449	(1,16)
_209_db	19,775	(1,00)	20,443	(1,03)	23,381	(1,18)
_222_mpegaudio	4,484	(1,00)	6,316	(1,41)	6,345	(1,42)
_228_jack	4,120	(1,00)	4,294	(1,04)	5,033	(1,22)
_213_javac	9,400	(1,00)	10,643	(1,13)	12,401	(1,32)
Geometric Mean	6,970	(1,00)	7,989	(1,15)	8,717	(1,25)

to the rewritten JDK). Note that we did not apply any optimizations to reduce the accounting overhead. Simple optimization rules, as discussed in [7], can help to reduce the overhead significantly. The implementation of the optimization algorithm is still in progress.

6.2 Limitations

Here we present the limitations that are inherent to, respectively, the way we implement resource control, and the design of the programming model. We terminate with a short discussion of the security provided by our approach.

6.2.1 *Limitations of the Implementation*

Our objective is to achieve a completely portable implementation of resource control. This means that we should be able to supervise off-the-shelf applications, and also that the execution environment should be independent of any underlying hardware and operating system.

In reality, we had to resort to language features (e.g., weak references) that exist only since the Java 2 platform [26]. These are incorporated in the application during the rewriting process, thus allowing backwards compatibility with code compiled for previous versions of Java. Another limitation comes from the interdiction to use several parts of the reflection package, because it enables the programmer to circumvent our resource control mechanisms. The same remark applies to the use of arbitrary native code libraries.

These limitations still allow our bytecode processing tool to encompass a very wide range of applications. However, once the goal is not only to monitor resource consumption, but also to control it, additional restrictions apply. In our approach, the runtime resource control libraries cannot e.g. allow an application to change the priorities of its threads, since this would confuse our scheduler. More generally, schedulers entirely implemented in Java are always dependent on the reliability of the underlying threads and related priority mechanisms. This is the price to pay for portability.

Finally, if we restrict ourselves to the J-SEAL2 platform, then the application must conform to the corresponding programming model, which implies respecting all of the previously mentioned limitations, plus an additional set, as can be found in [6].

6.2.2 *Limitations of the Programming Model*

Our approach is designed to enable the transparent execution of legacy applications, as long as they respect the above limitations; their resource consumption will then be controlled by the runtime system without their knowing.

From the moment one wants to develop a resource-aware application, it becomes necessary to conform to the basic API described in section 4. This API is however primarily designed for use by the runtime system, and is probably too low-level for the application programmer. Therefore, we still have to define an uniform and extensible programming interface that includes kernel-level resources, as well as service access, bandwidth limitations, elapsed wall-clock time, etc.

Also on our todo-list is the development of high-level programming tools in order to support a friendlier event notification mechanism than the overuse exceptions generated by the J-SEAL2 kernel. User-specified thresholds should enable applications to receive warnings in a timely manner before the actual overuse happens.

6.2.3 *Coverage of Security Issues*

Since the focus of this paper is on preventing excessive resource consumption by hostile or poorly implemented programs, we put forward the following qualitative characterization of our approach, until the resource control model is completely integrated in J-SEAL2:

- When a thread of a parent domain is executing, it is able to terminate its children.
- Because there is no direct sharing between domains, all resources of a terminated domain can be reclaimed properly.
- A parent can always reserve a fraction of its CPU share for a supervisor thread, which will start executing with priority NORM_PRIORITY. This thread wakes up periodically to terminate children that are too long in the system; therefore, this thread will not use a lot of CPU time, and the scheduler will not lower its priority. All threads in the subdomains will have a priority less or equal to NORM_PRIORITY (in particular, a thread which uses the CPU excessively will have a priority less than NORM_PRIORITY). Thus, the supervisor will be eventually scheduled by the JVM, and be able to remove malicious domains.

- Resource limits are assigned by the parent, and only fully trusted domains are able to create new quotas ex nihilo. A domain with resource limits is not able to extend its own limits.

7. RELATED WORK

We distinguish two broad categories of related work on adding resource control to Java: those which have security as main objective, and those which follow other motivations.

7.1 Resource Control for Security Purposes

Compared to existing proposals for realizing resource control in Java, we broadly differentiate our approach in two ways: first, whether the model supports a process-based approach, with well-defined domain boundaries and resource allocation for each application, and, second, to which extent the implementation is portable or not.

JRes [12] is a resource control system which takes CPU, memory, and network resource consumption into account. The resource management model of JRes works at the level of individual Java threads; in other words, there is no notion of application as a group of threads, and the implementation of resource control policies is therefore cumbersome. JRes is a pure resource accounting system and does not enforce any separation of domains; covering this other aspect is the goal of J-Kernel [34], a complementary project of the same research team. For its implementation, JRes does not need any modification to the JVM, but relies on a combination of bytecode rewriting and native code libraries. To perform CPU accounting, the approach of JRes is to make calls to the underlying operating system, which requires native code to be accessed[19]. For memory accounting, it essentially uses bytecode rewriting, but still needs the support of a native method to account for memory occupied by array objects. Finally, to achieve accounting of network bandwidth, the authors of JRes also resort to native code, since they swapped the standard `java.net` package with their own version of it.

KaffeOS [1] is a Java runtime system which supports the operating system abstraction of *process* to isolate applications from each other, as if they were run on their own JVM. Thanks to KaffeOS, a modified version of the freely available Kaffe virtual machine [35], it is possible to achieve resource control with a higher precision than what is possible with bytecode rewriting techniques, where e.g. memory accounting is limited to controlling the respective amounts consumed in the common heap, and where CPU control does not account for time spent by the common garbage collector working for the respective applications. The KaffeOS approach should by design result in better performance, but is however inherently non-portable. This means that optimizations found in compilers and standard JVMs are not benefited from: in a recent publication [2] the authors report that, in absence of denial-of-service attack, IBM's compiler and JVM [24] is 2–5 times faster than theirs.

Developed by the same team as KaffeOS, Alta [32] is a prototype based on the Fluke hierarchical process model, and implemented on the Kaffe virtual machine. The main differences with KaffeOS are that a single garbage collector is responsible for all applications, and that Alta entirely respects the hierarchical process model of Fluke by providing resource control APIs, whereas KaffeOS only retains a more implicit nested CPU and memory management scheme.

NOMADS [29] is a mobile agent system which has the ability to control resources used by agents, including protection against denial-of-service attacks. The NOMADS execution environment is based on a Java compatible VM, the Aroma VM, a copy of which is instantiated for each agent. There is no resource control model or API in NOMADS; resources are managed manually, on a per-agent basis or using a non-hierarchical notion of group. Relying on a specialized VM, it follows that the overhead is smaller than with our approach; currently, CPU control is however not implemented.

Many other systems are proposed in the literature, but none of them are as complete as JRes, Alta, and KaffeOS. An excellent overview is provided in [3]. To summarize, we might say that J-SEAL2 proposes a protection model inspired both from Alta and J-Kernel, and a memory accounting implementation that is more reminiscent of JRes.

7.2 Other Java-centric Approaches to Resource Control

There are several lines of research, where environments and analysis tools have been designed that can be exploited more or less with the same objectives as exposed in this paper.

The Real-Time for Java Experts Group [8] has published a proposal to add real-time extensions to Java. One important focus of this work is to ensure predictable garbage collection characteristics in order to meet real-time guarantees. For instance, the specification provides for several memory management schemes, such as areas with limited lifetime or bounded allocation rates, which could be implemented – or at least simulated – with the J-SEAL2 extensions described in the present paper. Another real-time system, PERC [23], extends Java to support real-time performance guarantees. To this end, the PERC system analyzes Java bytecodes to determine memory requirements and maximal execution times, and feeds that information to a real-time scheduler. The objective of real-time systems is to provide precise guarantees e.g. for worst-time execution; our focus, on the other hand, is on computing approximated resource consumptions in order to prevent denial-of-service attacks. We are more interested in the relative values of applications, and less in absolute figures. This is confirmed by the fact that we are not trying to estimate their real CPU consumption, but rather to compare the respective number of executed bytecodes.

Profilers constitute another class of tools that have many aspects in common with resource control: both intend to gather information about resource usage. Profilers however are designed to help developers optimize the efficiency of their applications, and not to externally control their resource consumption. The Java Virtual Machine Profiling Interface (JVMPI) [28] is an API created by Sun; it is a set

[19] More precisely, CPU accounting in JRes is based on native threads, a feature not supported by every JVM.

of hooks to the JVM which signals interesting events like thread start and object allocations.

Finally, we mention some approaches that rely on economics-based theories, using virtual currencies to achieve natural load-balancing of concurrent applications, as well as recycling of unused resources in open distributed environments, with the anticipated side-effect of preventing denial-of-service attacks [31]. Our focus is however more on how to implement the basic resource accounting mechanisms on a specific platform, Java, than on the design of high-level – and distributed – resource allocation policies. Nevertheless, whereas the spirit of this paper is rather conservative, it does not exclude the application of the presently described techniques to the implementation of open computational markets.

8. CONCLUSION

The contributions of this paper are two-fold. First, we propose a highly portable implementation of resource control in Java, and second, we show how this approach can be cleanly integrated into an existing framework, J-SEAL2, with the creation of a corresponding programming model. The techniques described in this paper have been tested with an off-line rewriting tool, and the overhead due to the accounting code has been measured with the standard SPEC JVM98 benchmarks.

Whereas other approaches focus on high performance, or demonstrate a long-term, deep re-design of the Java runtime system, our proposal might be grossly characterized as a language-based patch. Our resource control system does indeed not provide the same level of accuracy of measurements and execution speed. On the other hand, J-SEAL2 perfectly fulfills its job of isolating applications from each other, and particularly of preventing abusive resource consumption originating from inside the execution platform. Moreover, the extensive compatibility and portability of our approach makes it immediately usable for the benefit of large-scale distributed object systems, especially when mobile code is involved.

Acknowledgments

The authors would like to thank Julien Francioli, Rudolf Freund, Andreas Krall, Patrik Mihailescu, Klaus Rapf, and Jan Vitek for their valuable support.

9. ADDITIONAL AUTHORS

Additional authors: Rory G. Vidal (University of Geneva, rue Général Dufour 24, CH-1211 Geneva 4, Switzerland, email: vidalr5@cuimail.unige.ch)

10. REFERENCES

[1] G. Back and W. Hsieh. Drawing the red line in Java. In *Seventh IEEE Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, USA, Mar. 1999.

[2] G. Back, W. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI'2000)*, San Diego, CA, USA, Oct. 2000.

[3] G. Back, P. Tullmann, L. Stoller, W. Hsieh, and J. Lepreau. Techniques for the design of Java operating systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, USA, June 2000.

[4] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation. Technical Report UUCS-98-015, University of Utah, Department of Computer Science, Aug. 6, 1998.

[5] W. Binder. J-SEAL2 – A secure high-performance mobile agent system. In *IAT'99 Workshop on Agents in Electronic Commerce*, Hong Kong, Dec. 1999.

[6] W. Binder. Design and implementation of the J-SEAL2 mobile agent kernel. In *The 2001 Symposium on Applications and the Internet (SAINT-2001)*, San Diego, CA, USA, Jan. 2001.

[7] W. Binder, J. Hulaas, and A. Villazón. Resource control in J-SEAL2. Technical Report Cahier du CUI No. 124, University of Geneva, Oct. 2000. ftp://cui.unige.ch/pub/tios/papers/TR-124-2000.pdf.

[8] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, Reading, MA, USA, 2000.

[9] C. Bryce and J. Vitek. The JavaSeal mobile agent kernel. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, Oct. 1999.

[10] S. Chiba. Load-time structural reflection in Java. In *ECOOP*, pages 313–336, 2000.

[11] G. Cohen, J. Chase, and D. Kaminsky. Automatic program transformation with JOIE. In *1998 USENIX Annual Technical Symposium*, pages 167–178, 1998.

[12] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33, 10 of *ACM SIGPLAN Notices*, pages 21–35, New York, USA, Oct. 18–22 1998. ACM Press.

[13] M. Dahm. Byte code engineering. In *Java-Information-Tage 1999 (JIT'99)*, Sept. 1999. http://bcel.sourceforge.net/.

[14] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and execution models in the fluke kernel. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI-99)*, pages 101–116, Berkeley, CA, USA, Feb. 22–25 1999. Usenix Association.

[15] B. Ford and S. Susarla. CPU Inheritance Scheduling. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, 1996.

[16] M. Godfrey, T. Mayr, P. Seshadri, and T. von Eicken. Secure and portable database extensibility. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 390–401, New York, USA, June 1–4 1998. ACM Press.

[17] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[18] J. Hulaas, L. Gannoune, J. Francioli, S. Chachkov, F. Schütz, and J. Harms. Electronic commerce of internet domain names using mobile agents. In *Proceedings of the Second International Conference on Telecommunications and Electronic Commerce (ICTEC'99)*, Nashville, TN, USA, Oct. 1999.

[19] R. Keller and U. Hölzle. Binary component adaptation. In E. Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 307–329. Springer, 1998.

[20] H. B. Lee and B. G. Zorn. BIT: A tool for instrumenting Java bytecodes. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (ITS-97)*, pages 73–82, Berkeley, Dec. 8–11 1997. USENIX Association.

[21] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[22] L. Moreau and C. Queinnec. Design and semantics of Quantum: a language to control resource consumption in distributed computing. In *Usenix Conference on Domain-Specific Languages (DSL'97)*, pages 183–197, Santa-Barbara, CA, USA, Oct. 1997.

[23] K. Nilsen. Java for real-time. *Real-Time Systems Journal*, 11(2), 1996.

[24] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[25] Sun Microsystems, Inc. Enterprise JavaBeans Technology. Web pages at `http://java.sun.com/products/ejb/`.

[26] Sun Microsystems, Inc. JAVA 2 Platform, Standard Edition. Web pages at `http://java.sun.com/j2se/1.3/`.

[27] Sun Microsystems, Inc. Java Servlet Technology. Web pages at `http://java.sun.com/products/servlet/`.

[28] Sun Microsystems, Inc. Java Virtual Machine Profiler Interface (JVMPI). Web pages at `http://java.sun.com/j2se/1.3/docs/guide/jvmpi/index.html`.

[29] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, T. S. Mitrovich, B. R. Pouliot, and D. S. Smith. NOMADS: toward a strong and safe mobile agent system. In C. Sierra, G. Maria, and J. S. Rosenschein, editors, *Proceedings of the 4th International Conference on Autonomous Agents (AGENTS-00)*, pages 163–164, NY, June 3–7 2000. ACM Press.

[30] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Web pages at `http://www.spec.org/osg/jvm98/`.

[31] C. F. Tschudin. Open resource allocation for mobile code. In *Proceedings of The First Workshop on Mobile Agents*, Berlin, Germany, Apr. 1997.

[32] P. Tullmann and J. Lepreau. Nested Java processes: OS structure for mobile code. In *Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, Sept. 1998.

[33] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, 1999.

[34] T. Von Eicken, C.-C. Chang, G. Czajkowski, and C. Hawblitzel. J-Kernel: A capability-based operating system for Java. *Lecture Notes in Computer Science*, 1603:369–394, 1999.

[35] T. Wilkinson. Kaffe - a Java virtual machine. Web pages at `http://www.kaffe.org/`.