Submitted to OSDI 2008.

Fast Capability Transfer using Existing Commodity Hardware

Tom Bachmann
e_mc_h2@web.de

Neal H. Walfield
TU Dresden
neal@os.inf.tu-dresden.de

Abstract

Current operating systems provide inadequate mechanisms to protect user data. The main problem is that all of a user's programs run in the same trust domain. A better model is one which is consistent with the principle of least authority (POLA). An object-capability system may be able to better achieve this: capabilities bundle authorization and designation thereby easing delegation and the dynamic creation and management of fine-grained trust domains.

Despite this, object-capability designs are rejected due to a perceived excessive overhead resulting from the degree of decomposition and the corresponding rise in the amount of inter-process communication (IPC). Although the work on L4 has demonstrated that IPC can be made extremely fast, historically, L4 lacks mechanisms to efficiently delegate fine-grained authority.

In this paper, we present a capability transfer mechanism that exploits the memory management unit (MMU) present in all modern commodity hardware by using it to build a content addressable memory (CAM) to expedite capability resolution. For the common case of an IPC carrying a single capability, we observe a 2% increase in message transfer time compared to a less flexible but more commonly used IPC implementation based on capability registers. Relative to the time taken to transfer a similarly sized message containing just data on L4Ka::Pistachio, we observe a 16% increase.

1 Introduction

Users are vulnerable. Computers are being used to store increasingly large amounts of valuable data and perform sensitive transactions, yet, a cracked computer can be bought on the black market for just $0.05 cents [29]. The problem with such commodity systems is that they lack the ability to isolate program instances from one another. Thus, it takes just one malicious or compromised application or plug-in to gain control of a user's system. If programs were run according to POLA, they would have access to only those resources they require to perform the intended task.

To achieve POLA using an access control list (ACL) mechanism is difficult: it requires the dynamic management of principals. To create a new trust domain, a new principal must be allocated and all resources to which it should have access must be updated to include the new principal on their respective ACL. When the principal is destroyed, all references to it need to be removed to allow reusing the principal's identifier and to avoid leaking storage. Run-time delegation requires that the source domain be able to identify the target domain.

In object-capability systems, capabilities bundle authorization and designation. This removes the need for special principal identifiers and a shared name-space for resources: to delegate access to a resource, a capability designating the resource is communicated to the target. As capabilities are self-authenticating, no further access checks are required. As capability are designators, no other identifier is required [15]. Despite their finer-grained access control, run-time delegation need not be an intrusive process [31].

Recently, sandbox environments employing capability concepts have been implemented to realize POLA on existing general-purpose operating systems. This includes Plash [18] for GNU/Linux, and Polaris [27] around the E language for Windows. This paper is concerned with operating system kernel based approaches to capability systems though, and we do not consider these approaches further.

Historically, the most successful capability systems, EROS [25, 23] and KeyKOS [8], have only allowed the direct designation of capabilities via a small, fix-number of so-called *capability registers*, not a large sparse address space. Their limited number allows for fast implementations of capability lookup. However, it requires the scheduling of capability registers by user programs,

resulting in copying. The justification is that the benefits gained from increasing the directly addressable capabilities are less than the impact on performance.

Capability registers represent a further problem when implementing asynchronous IPC: the capability registers referenced by a message remain blocked until the message transfer completes. This increases register pressure and limits the number of outstanding messages. For asynchronous IPC to be viable, the number of registers must be able to support the joint demand of a reasonable number of extant message buffers and an execution context. This can be achieved by the use of a large, sparse capability address space. However, this again raises the question of the impact on performance.

In this paper we explore the use of the hardware MMU, available in all modern commodity hardware, to implement a content addressable memory (CAM) to speed up capability lookup. We believe the observed speed up is sufficient to justify the use of sparse capability address spaces, which also enable the potential use of asynchronous IPC.

We also observe that this technique generalizes to solve a number of translation problems and can be implemented in user-space using operating system supplied virtual memory functionality.

The remainder of this paper is structured as follows: in the next section we give a quick overview of capability related questions. In the third section, we present the proposed technique and its application to capability address translation in both an abstract and a general fashion. Section 4 presents our implementation of an experimental microkernel incorporating the proposed features. We then evaluate the performance and the complexity of this experimental kernel in section 5. Section 6 contains a survey of related work. Section 7 presents suggestions for future work. We then close the paper with concluding remarks.

2 Capabilities

In an object-capability system, object invocation by way of capabilities is the primary means of communication. An object invocation transfers a message, possibly including capabilities, to the invoked object. On such systems, IPC is prevalent as the system is more decomposed than current general-purpose operating systems, which are generally monolithic. This motivates a mechanism for fast capability transfer.

There are different possible strategies to implement capabilities. Password capabilities (also known as crypto caps) [17] are protected via sparsity. That is, they reside in a global name-space and names are chosen that are difficult to guess. This ensures that capabilities must be communicated. One problem with this is that it makes capabilities only as strong as the random number generator used to generate them.

A more secure way is to used protected capabilities. In this case, the bit representation is not exposed to user capabilities and capabilities can only be communicated via *authorized* channels. In this case, local names are used to address capabilities stored in capability tables.

In such a system, the format of an in-memory capability depends on the needs of the implementation.

2.1 Address Spaces

Capabilities must be organized. By some means, userspace processes must be able to refer to them. Traditionally, this has been done using so-called capability registers which basically means that the in-kernel process structure contains space for a fixed number of capabilities that can be referred to by index. We believe that this fixed number represents a severe limitation. If a larger amount of capabilities must be managed than fix in the registers, complex register scheduling become necessary.

The idea of organizing capabilities in address spaces is not new [2]. By analogy to the way memory is addressed by virtual addresses, capabilities are addressed by capability addresses. Address spaces are created by mapping pages, i.e., fixed-sized areas of physical memory, to virtual address ranges in the virtual address space. Whereas memory is mapped using data pages, capabilities are mapped using capability pages, an abstraction implemented by the kernel. The kernel must ensure strict separation of types, as otherwise the complete security primitive is doomed fail.

2.2 Capability Lookup

Capabilities are primarily referenced when invoked and when transferred. When using capability address spaces, a capability is found by indexing the calling task's capability address space to find the location of the capability.

The key problem here is how to find the in-memory representation of that capability. Using capability registers as EROS does just requires indexing a small table. Using capability address spaces, however, requires walking a more complex data structure. This is slow as it requires significantly more memory references. We propose exploiting the MMU for its ability to cache translations and its very fast page table walker. This approach achieves the translation from capability address to kernel virtual address in not only a small, fixed amount of time, but further, no mapping data structures must be traversed explicitly, which is instead done by the memory management hardware, *and gains from all the hardware optimizations*, like translation lookaside buffers (TLBs).

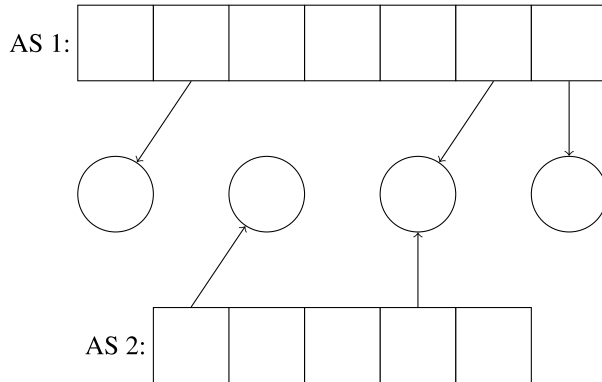

Figure 1: Two address spaces and the objects they address. One object is shared; the others are only accessible via one of the address spaces.

This is very different from the usual approach of manually walking mapping structures and even replicating hardware optimizations like TLBs in software. It is simpler, faster, and will profit from future optimizations as well.

Note, however, that different capability accesses are not necessarily independent. For example, in an interrupt-style microkernel such as EROS, no operation may fail *and* change observable state. This means that irreversible changes must only be made after it has been ensured that the operation will succeed. The fixup function must also reverse changes. In general, this results in a high intermingling of the fixup function and the IPC path, such that, they can only be viewed as a single entity.

3 Fast Address Translation

From a high-level perspective, address translation consists of finding an object associated with an address. That is:

- checking whether an $\{asid, address\}$ pair is valid, and
- translating an $\{asid, address\}$ pair to the location of the object.

Figure 1 illustrates this idea. It shows two separate address spaces and a number of objects. An entry in the address space may designate an object. Often, there is one address space per principal.

The problem is then finding a fast algorithm for this type of address translation and appropriate data structures. We observe that the translation process is essentially the same as translating a virtual address to a physical one, and that the hardware MMU was introduced exactly to support this type of operation.

3.1 The Memory Management Unit

The MMU (including the TLBs) is a piece of hardware present in all common modern commodity computing system. It is responsible for handling memory requests issued by the central processing unit (CPU). In particular, it translates virtual addresses to physical addresses, enforces memory protection and does cache control. To cache translations, the MMU uses a translation look-aside buffer (TLB), a content-addressable memory (CAM, also known as associative memory). On modern x86 and x86-64 processors, this typically consists of 1024 lines.

When handling an address, the result may not be in the cache. In this case and if the hardware supports it, a hardware page table walk is initiated. This is performed by the hardware page-table walker and is significantly faster than a corresponding software implementation. This is used on x86 and x86-64; architectures using a software loaded TLB have become less common. If a valid translation is found, the result is cached in the TLB and returned. Otherwise, a fault is raised (typically by way of an interrupt).

In particular, the MMU translates arbitrary virtual addresses to physical addresses, performing access checks as a side effect of translation, and caches translations. As the kernel fully controls the mapping data structures that the MMU interprets, it can organize the virtual address space layout as seen by the MMU as it sees fit. Using the following algorithm, the MMU can be exploited to do almost any kind of $key \mapsto value$ translation with access permission checks. In short, the MMU can be used to implement a general-purpose CAM.

3.2 Requirements

Before detailing the algorithm, we first enumerate its dependencies:

Address Translation There must be a mechanism that translates addresses according to user controllable rules.

Fault Reflection An address for which there is no valid translation must either produce a user-handleable fault or be otherwise easily detected.

Isolation Other principals on which the user does not rely must not be able to disturb the translation.

In the case of an operating system kernel, the first two requirements are satisfied by the hardware MMU, e.g., in terms of hardware page-tables and protection faults; the last requirement is dependent on the kernel to provide interfaces that do not allow destructive interference.

These requirements are moderate, almost any hardware platform that provides virtual memory will satisfy them. This includes the widely deployed x86, x86-64, powerpc, ia64 and ARM architectures.

Such hardware MMUs also provide the additional feature that in the process of translating addresses, access checks are performed. For instance, page table entries often contain read and write protection bits.

Some kernel virtual memory interfaces also satisfy these requirements. Most notable here are Mach memory objects [2], L4 pagers [7] and EROS keepers [20]. Even the legacy UNIX mmap/SIGSEGV mechanism is sufficient, though it is questionable if the usual implementation is efficient enough. This means that user-space processes can also use the algorithm to speed up translations. We will return to this possibility in the section on future work.

3.3 Sparse Arrays

A sparse array is used for address translation. The size of the array is determined by the range of the key values and the number of bits required to store a value. In the case where the values are object pointers, this will typically be the machine word size.

As the smallest unit of allocation is the page, when setting up a translation, we may also have to introduce translations for other keys. In this case, the MMU will not raise an exception when looking up these other keys. To detect such invalid keys, a special sentinel can be stored as the value. In this case, before using the value, it must be compared with this sentinel. If a value matches, an invalid translation exception must be raised. In the case where the value is a pointer, an appropriate sentinel is NULL.

3.4 Segments

Like most kernels, we logically divide the available virtual addresses (of which there are, e.g., 2^{32} on x86 and 2^{48} on x86-64) into a number of "segments." The term "segment" as used here is not related to any special hardware functionality but simply refers to logical address ranges. The MMU puts restrictions on the segment size and its alignment in the form of an architecture-dependent base-page size.

Most kernels divide a virtual address space into two segments: a kernel segment and a user segment. The kernel segment is mostly inaccessible from code executing in userspace and is the same in all address spaces. The user segment is per-process and valid mappings can be accessed using normal load and store instructions.

We introduce another per-process segment: the capability segment. The capability segment is per-process but, unlike the user segment, is not accessible using normal load and store instructions. Instead, a process must use the kernel interface to manipulate the contained capabilities.

The capability segment, rather than containing pointers to capabilities or capability pages, shadows the capabilities. That is, the kernel aliases the capability pages in the process's capability segment. The major advantage is that when we index the segment to find a capability, there is no level of indirection; the capability representation is immediately available.

Further, the use of a per-process capability segment means that it is mapped at the same offset in each address space (thus, it is at a known, fixed location) and always has the same size. This reduces the number of variables required to perform a lookup.

Finally, by making the segment a power-of-two and imposing the additional restriction that higher-bits are simply ignored, we are able to use bit operations to eliminate branches that would be required to perform bounds checking.

3.5 Algorithm

For each address space, we allocate a range of virtual addresses sufficient to hold the maximum number of keys (addresses) multiplied by the size of a value.

When looking up a value, a number of checks must be performed:

Bounds Check to prevent out of range errors,

Validity Check to ensure the translation is valid, and

Access Check to determine whether the principal is authorized to access the object.

We first perform the bounds check. If it the address is within the array, translation continues.

We then scale the input address according to the size of the value and bias the result relative to the base of the array. For instance, if a value is 8-bytes large, the address may be multiplied by 8 and the result added to the segment's base. This yields a so-called *intermediate virtual address*.

The intermediate virtual address is then used to read the key's value. This is done using normal hardware load instructions. A side-effect of this is that the MMU will perform the remaining address translation and may be used to perform any required access permission checks. Translation failure will be reported just like any other virtual address translation fault.

The generic algorithm described here is shown in listing 1. The kernel specialized algorithm, based on the observations in the previous sections, is shown in listing 2.

```
if key < 0 or key >= max
   throw OutOfRange
value = array[key]
if value == nil
   throw InvalidAddress
return value
```

Listing 1: Generic translation algorithm.

```
addr &= RANGE − 1
cap = BASE | addr ∗ sizeof (cap_t)
if ∗cap == nil
   throw InvalidAddress
return cap
```

Listing 2: Kernel optimized algorithm exploiting the fact that the range of keys is a power-of-two, that unused bits are ignored, BASE is a multiple of the segment size, and that the base and size are known at compile time.

The ability to validate an access is limited by the primitives provided by the underlying translation mechanism. Often the only way to check an address (apart from walking the mapping data structure) is to access it directly. This can be problematic if the kernel wants to ensure special manipulation semantics, e.g., an interrupt style interface is not allowed to do any externally visible modifications before the success of the complete invocation is ensured [6]. Usually the interrupt handler and the object access code can cooperate in such cases, however.

4 Prototype

To evaluate these ideas, we have implemented a prototype EROS-style microkernel called colonel for the x86-64 architecture [1].

4.1 Capability Lookup

To use the design described above, two major changes must be made to the way a microkernel is traditionally implemented. First, the kernel must maintain an additional segment in which it maps capability pages. Second, the IPC implementation and page fault handler must be modified to exploit the address translation mechanism, and, importantly, cooperate to recover from invalid addresses.

First, the kernel must provide an ability to map cappages, and whenever a cappage map request is issued by a userspace process, it must really map the page, but without enabling access to it using normal load and store instructions from userspace. This effectively makes each mapped cappage an address-space-local kernel page. Note that checking if an address is supposedly pointing to user data, kernel data, or a capability can be done without traversing the page table as the address range is statically divided.

Second, the IPC fast path needs to exploit the design. When handling a capability address, we first transform the the address to point to the capability's value (see listing 2). If an address is invalid, dereferencing it will cause a page fault. As any resulting fault will occur at a known addresses, the page fault handler can treat it specially. A fixup function that reports an error to the userspace process can be called. This interaction between the fixup routine and the fast path has to be carefully designed. If the kernel is implemented in an interrupt-style, for example, the fast path must not cause any irreversible changes before it has ensured that it will not fault.

4.2 Virtual Machine Model

The virtual machine model is very similar to that of EROS [20]. There are two types of operations: instruction executions and capability invocations. (The former can in fact be modelled as the latter.) Instruction executions are done by the hardware directly; capability invocations are implemented as system calls to the microkernel.

4.2.1 Capability Invocation

The general capability invocation primitive accepts the following arguments:

- bits indicating the request type (e.g., blocking RPC, send with zero timeout, etc.),
- the address of the capability to invoke,
- N data words to be sent,
- up to M addresses of capabilities to transfer,
- up to M addresses of capability slots in which received capabilities are to be stored,
- an *optional* indirect send string, and
- an *optional* indirect receive string.

In our implementation on the x86-64 architecture, we make four registers available for transferring data words, one for sending a capability, and another for receiving a capability. An additional six capabilities can be specified on the stack, three for sending and three for receiving. As the data words are sent from and received to registers

[1] Full source code (GPLv3) can be accessed at http://sourceforge.net/projects/colonel

only, there is no need to check any pointers into the user data address space.

Request Types In different situations, slightly different IPC semantics are needed. In general, an IPC consists of a send phase and a receive phase, both optional. The send phase is executed before the receive phase, if both are requested, and then the receive phase is executed only if the send phase was successful. Both the send and receive phases can be independently set to be blocking or non-blocking.

If the send phase is non-blocking, then the IPC will fail if the target is not currently in the receive state, if it is blocking, then the process executing the IPC will be blocked until the target becomes available.

If the receive phase is blocking, then the receive part will fail if no sending process is currently blocked on the executing process. Else the executing process will be blocked until another process tries to send to it.

Additionally, an IPC can be requested as a "call" operation, which means sending and receiving are blocking. Furthermore, a special "return" capability that allows return-at-most-once RPC semantics is manufactured and inserted as the first capability argument.

4.2.2 Kernel Implemented Objects

All kernel interfaces are exposed as kernel-implemented objects and accessed by invoking capabilities corresponding to such objects. (Capabilities to kernel-implemented objects are indistinguishable from "normal" capabilities. Thus, in principle, every kernel object invocation can be transparently virtualized by a user-space process.) A consequence is that every call is explicitly authorized. Also as the object is designated explicitly, the confused deputy problem is more easily avoided [9].

Processes Processes are the first-class abstraction of a thread of execution. They consist of an address space, a register set, and certain miscellaneous state like a fault handler. From process capabilities so called 'sender capabilities' can be created. All invocations of such capabilities will be delivered as IPC requests to the associated process. Sender capabilities also contain an opaque 'protected payload' which is set at creation time and delivered alongside the arguments of an invocation. Processes can use the protected payload to use the same entry point for all message but still be able to distinguish among the multiple objects and interfaces (facets) they implement.

Pages Page capabilites are references to mappable regions of physical memory. They contain little more than the physical page address and a few bits to distinguish their type (data page or cappage) and to specify the possible access rights. The only method implemented by pages is access right reduction. So less privileged page capabilities can always be manufactured.

Address Spaces Address spaces are the first class abstraction for the virtual memory layout. In the current implementation, they are quite under-developed. They allow pages, i.e., physical memory regions designated by page capabilities, to be mapped into the virtual address space of a process. Note that capability and memory pages are treated equivalently in this interface, so there is a unified data and capability address space. Note further that multiple processes can share one address space, so the usual multi-threading idiom is implementable without special kernel support.

Return Capabilities In certain circumstances, most commonly during a remote procedure call to a different userspace process, so called 'return capabilities' are created by the kernel. These are invalidated as soon as they are invoked for the first time. This way 'return-at-most-once' semantics can be ensured during remote procedure calls. Return capabilities are also created if a fault message is delivered to a fault handler or if a hardware interrupt is signaled to a process.

Other There are a number of other miscellaneous capabilities: The 'ioperm' capability enables access to the port space (an x86/x86-64 specific method for accessing hardware devices). The 'intctl' capability allows a process to register to receive specific hardware interrupts (or IRQs). The 'sleep' capability allows the suspension of the execution of the invoking process for a certain amount of time. The 'range' capability allows the manufacturing of arbitrary capabilities. The 'wrapper' capability allows for selective revocation of capabilities. There are a few other capabilities to, e.g., create new processes.

4.3 Capability Representation

Colonel represents capabilities as two consecutive 64-bit words: a 16-bit target identifier, which is either a kernel internal process id or a magic value to indicate, e.g., that the capability points to a kernel implemented object, an 48-bit version number, which is used to realize fast invalidation of capabilities, and a 64-bit protected payload.

Destroyable objects (e.g., processes) contain a version number that is increased on object destruction. Capabilities to such objects are only valid if the version number in the capability and the object match. Thus such objects can be re-used without invalidating all capabilities

pointing to them. Of course, the version number space must be sufficiently large to make sure that no overflow occurs. Assuming a maximum of 10^5 possible object destructions and creations per second, the kernel can run for at least $\frac{2^{48}}{10^5}$ seconds or about 89 years without any overflow.

4.4 Virtual Address Space Layout

As mentioned above, colonel implements a shared address space, i.e., both capability and data pages can be mapped into the same address space (at different addresses, of course).

To do so, colonel divides the address space into four segments of equal size, the user data segment, the user capability segment, the kernel data segment, and an unused segment. This is not optimal, as it wastes space: the kernel segment usually can be much smaller than the other segments, and the unused segment is not necessary, however, this simplified the implementation. For the x86-64 platform, which provides a large virtual address space, the impact on address space contention is low and thus the decision does not represent a practical limitation. The segments are used to implement a shared address space for data and capabilities. This is achieved by cross-checking mappings in both segments. Thus for every capability page that is mapped into an address space, colonel creates a "shadow" mapping in the user capability segment that allows the kernel to access the capabilities at an address that is easily computed from the capability address provided by userspace process. This is the key idea to speed up the capability transfer.

4.5 Current Limitations

It has to be stated that our experimental microkernel implementation does not provide all facilities of a real-world kernel. Mostly missing is accountable address space construction, as is support for cache control (e.g., for memory mapped I/O in device drivers). The interface for object allocation is functional in principle, but not as good as it could be. Also some kernel interfaces, like the traditional sleep capability, are not yet implemented. We do believe, though, that it provides everything that is relevant to assess IPC performance.

Address Space Creation Currently, address space creation is not accounted, which is of course a severe problem for microkernels aiming for security by offering capability protected IPC. When a process invokes an address space capability, for instance its own, and requests a page capability to be mapped at a certain address, then the complete operation is secured by capabilities, but the memory needed for the mapping is not accounted for. So the kernel checks, that the process may validly invoke the address space capability, and also that the page provided capability is valid, and so on, but when it comes to creating the mapping structure needed by the hardware, it allocates the memory on its own, as does for example L4Ka::Pistachio [11]. This of course makes the kernel vulnerable to very easy denial of service attacks. This kind of vulnerability is characteristic for (micro-)kernels not implemented with security in mind, e.g., Mach and L4Ka::Pistachio.

This interface shortcoming can be fixed relatively straight-forwardly. Looking at prior work, there are three approaches to this:

Address Spaces as Node Trees In the EROS kernel, address spaces are encoded in a machine-independent fashion as trees of nodes [25]. Nodes also encode the height of the tree they refer to, allowing short-circuit traversal and minimizing the need for complete and therefore relatively tall trees. This architecturally independent encoding is translated on demand into the representation required by the underlying hardware.

Kernel Heaps A different approach is taken by L4.sec [11]. Instead of making explicit all the memory the kernel is supposed to use to fulfill a certain request, that memory is implicitly allocated from so-called "kernel memory objects". A userspace process can convert normal memory pages into pages to be used by the kernel. If not enough memory is available to fulfill a certain request, a fault is delivered to the userspace process, which can then provide more pages. Userspace processes can revoke memory provided to the kernel at any time. This scheme has the advantage of more transparently extending the existing L4 primitives than a EROS-like node tree would do.

VSpaces Still another, novel approach is implemented in seL4 [4]. Generally, address spaces are composed of "CNodes", which are comparable to EROS nodes but of variable size. Several of them are used to describe an architecture independent guarded page table (GPT) [13] address space structure. For software loaded TLBs, there is not much more to say. On more conventional architectures, these GPTs are augmented by so-called VSpaces. V-Spaces superficially resemble the CNode tree structure but are specifically tailored to the underlying hardware. Thus their size is generally fixed, as is the nesting depth, and the object types that can be placed in them are very limited. Basically, they di-

rectly abstract the page table structure of the hardware.

Of course two or more of these techniques could be combined.

Cache Control To implement device drivers and some multiprocessor shared memory protocols, it is necessary to be able to control the caching options of mappings. There is no interface, currently, to do that, but adding options to the `address_space.map` method should be simple. It might be desirable to disallow use of this extended method by all but certain privileged processes. This kind of restriction would be trivial to implement as well, by adding access rights bits to the address space capability. This is already done for, e.g., page capabilities to indicate whether it is allowed to map them writable.

Object Allocation The current interface for object allocation is functional, in principle, but very hard to use correctly and safely. It was derived, in spirit, from the EROS interface [19] but adapted only inadequately for the multiple object sizes colonel offers. There is a so-called range capability which allows new kernel objects to be allocated from an amount of memory reserved by the kernel at boot time. While this seems to open the kernel to the same kind of vulnerabilities as does the current unaccounted address space creation, this capability has to be treated as highly privileged anyway. Typically it will be wrapped by a privileged userspace system process that can realize some allocation policy. This process has to do tremendous book-keeping, though.

Other Kernel Interfaces Certain miscellaneous kernel interfaces have not been implemented and some are not even fully specified. The most important of these are the sleep and scheduler control capability. As kernel interfaces are exposed through kernel implemented capabilities, adding new interfaces is very easy, though.

The sleep capability provides functionality to stop the execution of a process for a certain amount of time. It is generally relatively easy to implement, if the scheduler resides in the kernel. If it does not, the sleep capability is not even implemented in the kernel at all.

Scheduler control is a different matter. Scheduling has long been one of the few policies that remained in the microkernel. For this reason EROS provided a (privileged) scheduler control capability that allowed to manipulate the in-kernel scheduling queues [20]. Recent work on the L4Ka::Pistachio microkernel [28] has demonstrated the viability of entirely removing the scheduler from the kernel. This is harder to implement, of course, but it seems, from a minimalistic perspective, the correct option.

	Pistachio	baseline	capas
sloccount[a]	-	11,067	11,172
cycles	230	261	266
relative	1.0	1.13	1.16
relative	-	1.0	1.02

[a] generated using David A. Wheeler's 'SLOCCount'

Table 1: Comparison of colonel baseline, colonel capas, and L4Ka::Pistachio

Multiprocessor Support Multiprocessor support has been completely excluded from the initial design. It has been shown [30] that only extensive adaptations yield acceptable performance. These kinds of considerations not only distract from the different problem that we want to discuss here, they also require a lot of knowledge about the topic, which we would not claim to have.

5 Evaluation

We evaluate the performance of our capability lookup mechanism by comparing IPC fast path performance. (On colonel, the IPC fast path is taken if the payload consists of only data words and up to one capability.) A version of colonel entirely without capability address space support is used as a baseline. We also report on a partial, more conventional implementation of capability address spaces using a hashmap. Furthermore we compare L4-Ka::Pistachio performance on the same hardware.

All benchmarks were taken on an AMD Sempron64 CPU.

5.1 Comparison to Baseline Colonel and L4Ka::Pistachio.

We measured the performance of the optimized common case, a so-called null RPC, and compare it to the performance of the same operation on the L4Ka::Pistachio microkernel, widely known for its superior performance [14]. This is the case of one process calling a second process and blocking until the reply is received. The called process (e.g., a server) immediately replies. The call carries a minimal payload, that is to say only direct data words; no capabilities and no indirect strings are transferred. Note that colonel still checks all IPC for validity. This means that it does the following things that L4Ka::Pistachio does not:

- dereference and check the capability invoked by the caller,
- generate a 'return capability' for the callee,

- dereference and check the (return) capability invoked by the callee, and
- invalidate the return capability.

L4Ka::Pistachio provides no protected IPC: threads have local global identifiers. Colonel 'baseline' provides capability protected IPC via EROS-style capability registers. Colonel 'capas' provides the same kind of protected IPC, but via capability address spaces.

As can be seen from the table, colonel is only marginally slower than L4Ka::Pistachio, taking respectively 1.13 and 1.16 times as long as L4Ka::Pistachio. However, colonel is, as far as we can tell, significantly less aggressively optimized while, by definition, doing more work.

Furthermore, we have tried to measure the increase of code complexity due to capability address spaces. As can be seen from the table, the code size grew by about 100 SLOC.

We want to emphasize the small cost of switching from a capability registers implementation to a capability address space one. This has tremendous effect on the operating system design space.

The comparison to L4Ka::Pistachio shows that colonel is a realistic microkernel. It has been demonstrated that microkernels offering capability protected IPC can be as fast as those offering no protection [22].

5.2 A More Traditional Implementation

Additionally, we want to demonstrate that exploiting the MMU for capability resolution is really superior to other lookup strategies. For this reason, we added a centralized, fixed-size software TLB for capability lookup. It is implemented as a hashmap, using traditional modular hashing. The number of entries is a fixed prime, typically 1021. Each entry consists of an address space identifier ($asid$) and a virtual cap-page address ($addr$), as well as the allowed access rights. To dereference an $\{asid, addr\}$ pair, an index into the hashmap is computed as

$$asid \oplus addr \pmod N,$$

where $\oplus$ denotes exclusive-or and N is the number of entries. If $asid$ and $addr$ used to compute the index match the stored $asid$ and $addr$, and the necessary permissions are available, the kernel can directly access the capability at $addr$. Otherwise, the page table hierarchy must be walked by hand to check if the access is valid (if this is the case, an entry is inserted into the hashmap). In case of a collision, the entry is simply overwritten.

We want to emphasize that this is a very idealistic design. It is a "best-guess" at the performance that could be achieved. If a hashmap was to be adopted as the primary implementation technique, it would almost certainly be slower than what we report here. For example, we carefully ensured that no collisions would happen in our benchmarks.

	cycles	overhead
MMU	266	-
partial hashmap	306	15%
full hashmap (extrapolated)	-	$> 30\%$

Table 2: Comparison of different lookup strategies

On the fast path, there are four locations where capabilities are accessed, and therefore four hashmap lookups would be necessary. As this code is not completely trivial, we implemented the complete lookup only once (it cannot just be copied to the other locations, because there are not enough registers are available).

As can be seen from table 2, this incomplete code introduces a 15% penalty on lookup time. We believe that the overhead of a complete implementation would be at least 30%, probably more.

A TLB miss implies a lengthy procedure of four consecutive (physical) memory lookups. No reliable numbers seem to be available ([12] reports nine cycles overhead on the used x86 processor) but our measurement seems to imply a TLB miss overhead of about 10–15 cycles.

Furthermore, the hashmap implementation yields a more complicated IPC fast path. For example, the register pressure is increased.

In a sense, exploiting the MMU is an optimal solution in this case: apart from the architectural overhead of switching the address space, fast path performance mainly depends on the number of cold-cache memory references, which directly translate into the number of TLB entries required. There is a certain minimal amount of references that cannot be avoided. Generally, there is one such (cold-cache) reference for the sender, one for the receiver, and one for global kernel data. Adding a capability address space generally adds one more memory reference per capability access. As capability resolution and access are combined if the MMU is exploited the way we describe it, no other strategy can be better according to this metric.

The hashmap, on the other hand, requires two more TLB entries to validate the capability accesses. Furthermore, the hashmap has a larger cache footprint.

Note, however, that our hashmap implementation is suboptimal in that capabilities are not stored alongside the hashmap entries. This optimization would reduce the number of TLB entries required at the cost of slightly complicating kernel logic. It was not feasible to add this feature with a reasonable amount of work, so we cannot report on any hard numbers. We speculate, however,

that this optimized hashmap would not be so dramatically slower than the MMU based implementation. Indeed the number of TLB entries needed would be the same. The cache footprint, on the other hand, would still be increased.

As one additional disadvantage, we note that a hashmap implementation destroys all locality in the capability access patterns. In all but the extremely capability intense processes, the number of accessed capabilities will be rather small and fit inside one cappage. This means that TLB entries can generally be re-used even when invoking different capabilities. We emphasize that this destroyed locality still increases the costs of a hashmap implementation in more realistic scenarios.

5.3 Cold Cache Costs

It is generally accepted that the most performance sensitive situations will be those occurring extremely frequently. Therefore they are expected to generally run with hot caches. This is why pingpong is considered a viable benchmark: the userspace processes involved create minimal cache (and TLB) footprint, so that raw, cache-hot overhead is measured.

It is not clear whether these assumptions still hold for a capability address space model. There are more memory locations involved, so the cold-cache case might indeed be more important than before. For this reason, we attempt to measure cold-cache IPC performance as well. As the relevant costs are secondary effects to the actual cache flushing, which are generally hard to quantify, these measurements are less reliable than the other numbers we present.

TLB Misses Firstly, we measure the impact of flushing the TLB (except for global pages[2]) at the very beginning of the fast path. This simulates TLB-intensive processes doing IPC: in the cache-hot setting, the TLB entries for referenced capability pages are not available on the IPC *to* a process, but on the following IPC *from* the process, the TLB entry is still there and does not need to be reloaded. In the TLB-cold setting, this is not the case.

To make this more concrete, consider the server being called and immediately replying: After the address space switch to the server, a return capability is created at a server-specified location. This will cause a TLB miss. But when the server invokes the return capability in the cache-hot setting, the TLB entry for the return capability is still there and its reference (to validate and thereafter destroy it) does *not* cause any TLB misses. If, however, the server has referenced so many pages in between that the TLB entry for the return capability was replaced, there will be more TLB misses than in the cache-hot case, degrading performance. This is what the TLB-cold setting simulates.

	cycles	relative
cache-hot	266	1.0
TLB-cold	386	1.46

Table 3: Impact of Cold Caches

As can be seen from table 3, there is a considerable, though not a huge, overhead if the TLB entries must be reloaded.[3] It remains to be seen what effects this has on the overall system performance.

Data Cache Misses In addition to the TLB, data caches also have crucial impact on fast path performance (and any performance-sensitive code accessing memory in general). The data caches are indexed by physical addresses, i.e., they need not be flushed during an address space switch. Again memory-intensive userspace processes will cause cache lines to be replaced, thus degrading IPC performance.

Sadly this cannot be easily simulated, as there are significant write-back costs that must not be neglected. This means that generally the secondary effects we want to quantify are not much larger than the costs of actually flushing the caches. Naively inserting a cache-writeback-and-flush instruction (`wbinvd`) at the beginning of the fast path yields code that is about two orders of magnitude slower than the original one. This is because *all* cache lines are written back and evicted. Instead we would like to measure only the secondary costs of evicting only those cache lines accessed by the fast path. This does not seem to be possible in our setup.

To still give an indication of the importance of data cache misses, we note that two 64-bit accesses to the Opteron's[4] L1 cache are possible in one cycle. A L1 cache miss, on the other hand, takes at least 10 cycles to load from the L2 cache [3]. Accessing the main memory has a significantly greater latency still.

Other Hardware Architectures The analysis we present is very much targeted towards the current AMD64 processors. The situation is probably not all that much different on other x86 and x86-64 processors from AMD and Intel. But on other hardware architectures,

[2] Global pages are pages that are not normally flushed during address space switches, e.g., the pages occupied by the kernel code.

[3] The number we present here is slightly problematic in that it includes the time necessary to do the TLB flush itself, not only the secondary effects that we really want to measure. We do believe, though, that these secondary effects are much larger than the cost of flushing in the first place, so this should induce a negligible incorrectness. Attempts to measure the raw flushing time gave contradictory results.

[4] The AMD Opteron is a processor model quite similar to the Sempron.

e.g., those providing software loaded TLBs or very different cache architectures, the situation can be spectacularly changed and other trade-offs may need to be made. This is an area for future work.

6 Related Work

Microkernels have long been a research topic. While Mach fueled this in the late 80s, its interfaces have proven inappropriate for today's hardware architectures. L4 has demonstrated the feasibility of fast IPC [12], and EROS has proved wrong many of the claims of the inherent performance problems of capability systems [25]. Microkernel systems have in the meantime been adopted by the industry as well, which is one more indication of the fact that they have overcome most of the early problems of Mach. On the other hand, they are still not used in the way we imagine, i.e., as the foundation of a decentralized object capability system.

During the last years, a trend of introducing more complexity into the formerly extremely scarce interfaces of second generation microkernels, sometimes even reinstating interfaces from first generation microkernels (in slightly adapted form), can be noted. Liedtke concentrated on IPC performance, which he diagnosed as the most important limiting factor of first generation microkernels [12]. But since then it has turned out that completely reduced interface primitives are neither sufficient to implement complex and secure real-world systems, nor necessary to provide fast interprocess communication. Along these line, we seek to reintroduce the capability address space concept in an optimized form.

6.1 Mach

Mach is one of the most influential first generation microkernels. Capabilities are used for interprocess communication and there is a capability address space. Generally, Mach provides very rich and flexible interfaces. Too often, however, this flexibility has been implemented without considering performance implications. For this reason, Mach performance has almost always been disappointing and often been one reason for ambitious projects to fail [5]. Nonetheless, Mach ideas and interfaces have influenced all following microkernels.

6.2 The L4 Family

Initially, L4 was written as an extremely minimalistic microkernel with one determining goal: IPC performance [12]. To achieve this, all former richness in interfaces has been sacrificed, leaving only the bare minimum. The desired result of fast IPC, however, could be achieved, demonstrating the feasibility of the microkernel approach. Linux has been ported as a userspace server to L4 [10].

L4 is still an active research platform. L4ng[26], seL4[4] and L4.Sec[11] are all developed to improve certain shortcomings of L4, mostly resulting from the extremely scarce interfaces. After achieving its initial goal of super-fast IPC, L4 is continually being extended to accomodate requirements such as security, dependability and usability.

6.3 EROS and Coyotos

EROS [23] was derived from the KeyKOS [8] operating system, initially with security and not performance in mind. As such, it provides capability protected IPC and orthogonal persistence. After the success of Liedtke, EROS IPC was redesigned to achieve comparably high performance. Perhaps surprisingly, it was demonstrated already in 1996 by this redesigned IPC system that extremely minimal interfaces like L4 provides are not necessary to achieve high performance [22].

The EROS project has now been abandoned in favor of two successors, CapROS and Coyotos [24]. While CapROS mostly continues the EROS project, Coyotos departs from EROS, to correct some of its shortcomings, to demonstrate feasibility of an atomic microkernel design in many different situations, and to use software verification methods to prove security properties.

7 Future Work

7.1 Asynchronous IPC

One of the strongest motivations for introducing a capability address space is asynchronous IPC. It turns out that in certain special situations a well-optimized synchronous IPC primitive just is not sufficient. One example of these cases is the traditional of the UNIX `select` system call. Implementing this using synchronous IPC only can be very expensive requiring at least one thread per server and more often one thread per object. Furthermore, experience suggests that not having to worry about certain corner cases of strictly synchronous communication can greatly facilitate system design (of course too eager use of asynchronous communication introduces new corner cases). For all practical purposes, synchronous IPC has to be the common case, as it is almost always much faster than the asynchronous one. But this is not a problem, as asynchronous IPC is introduced to aid a few uncommon but otherwise problematic scenarios.

Implementing asynchronous IPC in a capability registers environment is very impractical. A potentially large number of outstanding requests will always be bounded

by the number of capability registers. This the inherent limitation of course is not unique to the asynchronous IPC case, we use it to motivate the capability address space concept in general. In the context of asynchronous IPC it is much more pressing, though.

7.2 32-Bit Systems

The design in its current form cannot reasonably be used on systems providing only 32-bit virtual address spaces. Creating four equally sized segments would reduce the effective address space to 30 bits, or 1GiB (per segment), which is unacceptable.

This is, however, not necessarily so. The design was worked out for a system with plenty of virtual address space, and with the goal of implementing a fast, shared address space of capabilities and data. (The sharing was supposed to simplify some programming semantics. It has to be evaluated to what extent this is really the case.) Once we drop the requirement of a shared address space and sacrifice some implementation simplicity, the shortcoming can be easily solved by creating three segments of different size, a large one for user data, a smaller one for kernel data, and a very small one for user capabilities.

As an example, assume that an effective user data address space of 3GiB is desired. The capability address space usually can be much smaller. Note, though, that capabilities usually are not byte-sized, so for example if one capability is four bytes in size, 4MiB of capability address space really mean "only" 2^{20} capabilities. Assume that this is still sufficient. The virtual address range can then be divided into three parts:

Address	Region
0	
$\vdots$	User data.
$3 * 2^{30}$	
$\vdots$	Kernel addresses.
$2^{32} - 2^{24}$	
$\vdots$	User capabilities.
2^{32}	

Note that $2^{32} - 2^{24}$ is represented by the address $0xff000000$, or 11111111000... binary. So when the userspace process provides an address into its capability address space, which starts at address 0 and ends at $2^{24} - 1$, the kernel sets the eight most significant bits of the 32-bit address of the capability to 1 to find the 'real' capability address.

7.3 Different Optimizations

We report on one special optimization to improve and speed up IPC. Following the trend we observe, we believe that different optimizations and enhancements in semantics will emerge. We address two of these we can imagine in this subsection.

7.3.1 Larger Direct Payloads

Traditionally, the direct payload that is transferred on the "super fast" IPC path has been very limited. This is because traditionally the direct payload has been transferred in registers only. This very much limits the feasible IPC interfaces. As these (usually) have to be equal on all architectures, the most register-scarce architecture limits the whole design space. Effectively, this traditionally confines all interfaces to the limits imposed by the x86 architecture, or about four 32-bit words of direct fast path payload. Exploring possibilities to exploit specifics of this complicated architecture to overcome this limitation seems to be an interesting future topic.[5] If the x86 situation can be improved, the next most limited architecture would be x86-64, which allows for about a dozen 64-bit direct payload words, so the situation would be greatly improved.

7.3.2 Reference Counting

The need for reference counting arises naturally in certain garbage collection scenarios. The most important one of these could be the implementation of the so-called membrane pattern [16] which is a kind of local reference monitor among otherwise isolated processes. It wraps all capabilities transferred via IPC. A program that implements a very similar pattern is the 'rpctrace' debugging tool. To implement either of them sensibly (which for the membrane implies securely), reference counting is necessary, as otherwise the wrapper cannot guarantee to run in the same space complexity as the wrapped process(es). To see why, consider a wrapped processes continually exchanging capabilities, keeping always only the last one. As the wrapper cannot know the latter without reference counting, it will require space linear in the number of IPCs, whereas the wrapped process only uses constant space.

An implementation of reference counting would most likely have to exploit knowledge (or assumptions) about the common case IPC scenario. It could superficially resemble the "no-senders notification" [2] interface from Mach. For certain theoretic reasons, a secure membrane implementation is very much desired, which makes reference counting a very interesting future topic.

[5] Indeed, there is a very promising proposal [21].

7.4 Application to Other Descriptor Translation Problems

The technique we describe thus far has only been used to speed up capability address resolution. While all kinds of address translation problems that are sufficiently similar to virtual memory address translation are predestined to use the technique, other applications are possible. For instance on systems providing a large amount of virtual addresses, many kernel data structures that have formerly been implemented using hash tables can be replaced by sparse arrays, as has been pioneered by the K42 project [1].

One interesting possibility in this regard is locating pages in the page cache. The page cache is indexed by physical location of the page on backing store. To quickly determine whether a page is in memory and where it is located, we can again use a sparse array. As pages are paged aligned and assuming that there is not more than 2^{32+12} bytes of physical memory (16 terabytes), an address can be saved in 4 bytes of memory yields 1024 addresses per 4k page. Thus, 4k of virtual address space suffice to cover 4MB of backing store and just 2^{40} bytes of virtual address space cover one petabyte of backing store (recall: x86-64 provides 2^{48} bytes of virtual address space).

7.5 Application in Userspace

As we remarked earlier, virtual memory interfaces of recent kernels are powerful enough to even use the technique described in userspace. This opens a completely new realm of possible applications. For example, one could think of database lookups being sped up. There are many possibilities for exciting future work.

8 Conclusion

We present a general algorithm to exploit the MMU of current commodity hardware to speed up many descriptor translation problems. The lookup capabilities of the hardware to realize virtual memory, found in all modern commodity architectures, are utilized directly. Thus, applications can profit from all special optimizations inside the hardware, like TLBs, transparently.

Applying this algorithm to speed up capability resolution dramatically, we measure only marginal overhead both in IPC performance and code complexity. In our benchmarks, IPC performance compared to the same kernel without capability address spaces is degraded by merely 2%.

9 Acknowledgments

Many people have made possible this work. Marcus Brinkmann and Jonathan S. Shapiro have invaluably influenced this work. Without them, their work and their endless fruitful discussions this work would never have been started. We furthermore wish to thank the numerous other active members of the associated coyotos-dev and l4-hurd mailinglists without whom they would be far less valuable and inspiring.

References

[1] APPAVOO, J., AUSLANDER, M., DASILVA, D., EDELSOHN, D., KRIEGER, O., OSTROWSKI, M., ROSENBURG, B., WISNIEWSKI, R. W., AND XENIDIS, J. K42 overview, Oct. 25 2001.

[2] BARON, R., BLACK, D., BOLOSKY, W., CHEW, J., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. *MACH Kernel Interface Manual.* Carnegie-Mellon Univ., 15 Feb. 1988.

[3] DE VRIES, H. Understanding the detailed architecture of amd's 64 bit core. http://chip-architect.com/news/2003_09_21_Detailed_Architecture_of_AMDs_64bit_ Core.html, Sept. 2003.

[4] DERRIN, P., ELKADUWE, D., AND ELPHINSTONE, K. sel4 reference manual. Tech. rep., National ICT Australia, 2006.

[5] FLEISCH, B. The failure of personalities to generalize. In *HOTOS '97: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)* (1997).

[6] FORD, B., HIBLER, M., LEPREAU, J., MCGRATH, R., AND TULLMANN, P. Interface and execution models in the fluke kernel. In *Operating Systems Design and Implementation* (1999), pp. 101–115.

[7] GROUP, S. A. L4 experimental kernel reference manual version x.2 revision 5. Tech. rep., Dept. of Computer Science Universität Karlsruhe, June 2004.

[8] HARDY, N. The KeyKOS architecture. In *Operating Systems Review* (Oct. 1985), vol. 19, pp. 8–25.

[9] HARDY, N. The confused deputy (or why capabilities might have been invented). Tech. rep., Key Logic, 1988.

[10] HÄRTIG, H., HOHMUTH, M., LIEDTKE, J., SCHÖNBERG, S., AND WOLTER, J. The performance of μ-kernel-based systems. In *16th ACM Symposium on Operating Systems Principles* (October 1997).

[11] KAUER, B., AND VÖLP, M. *L4.Sec Preliminary Microkernel Reference Manual*, revision 0.2 ed. Technische Universität Dresden, Oct. 2005.

[12] LIEDTKE, J. Improving IPC by kernel design. In *Proceedings of the 14th Symposium on Operating System Principles (SOSP)* (Asheville, NC, Dec. 1993).

[13] LIEDTKE, J. Page table structures for fine-grain virtual memory, Dec. 20 1994.

[14] LIEDTKE, J., ELPHINSTONE, K., SCHÖNBERG, S., HÄRTIG, H., HEISER, G., ISLAM, N., AND JAEGER, T. Achieved IPC performance. In *Workshop on Hot Topics in Operating Systems* (1997), pp. 28–31.

[15] MILLER, M., YEE, K.-P., AND SHAPIRO, J. Capability myths demolished, 2003.

[16] MILLER, M. S. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, May 2006. See figure 9.3 for the membrane pattern.

[17] POSE, R. Password-capabilities: Their evolution from the password-capability system into walnut and beyond. In *ACSAC* (2001), IEEE Computer Society, pp. 105–113.

[18] SEABORN, M. Plash: tools for practical least privilege. http://plash.beasts.org.

[19] SHAPIRO, J. S. *EROS Object Reference Manual*.

[20] SHAPIRO, J. S. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999.

[21] SHAPIRO, J. S. mailing list message, June 2006. http://sourceforge.net/mailarchive/message.php?msg_id= 1182626772.13131.139.camel

[22] SHAPIRO, J. S., FARBER, D. J., AND SMITH, J. M. The measured performance of a fast local IPC, June 11 1996.

[23] SHAPIRO, J. S., AND HARDY, N. Eros: A principle-driven operating system from the ground up. *IEEE Software 19*, 1 (2002), 26–33.

[24] SHAPIRO, J. S., NORTHUP, E., DOERRIE, M. S., SRIDHAR, S., WALFIELD, N. H., AND BRINKMANN, M. Coyotos microkernel specification. http://coyotos.com/docs/ukernel/spec.pdf, Mar. 2006.

[25] SHAPIRO, J. S., SMITH, J. M., AND FARBER, D. J. EROS: a fast capability system. In *Symposium on Operating Systems Principles* (1999), pp. 170–185.

[26] SKOGLUND, E. Personal communication, 2004.

[27] STIEGLER, M., KARP, A. H., YEE, K.-P., AND MILLER, M. Polaris: Virus safe computing for Windows XP. *Communications of the ACM 49*, 9 (2006), 83–88.

[28] STOESS, J. Towards effective user-controlled scheduling for microkernel-based systems. *Operating Systems Review 41*, 3 (July 2007).

[29] THOMAS, R., AND MARTIN, J. The underground economy: priceless. *;login: 31*, 6 (Dec. 2006).

[30] UHLIG, V. *Scalability of Microkernel-Based Systems*. PhD thesis, University of Karlsruhe, Germany, May 2005.

[31] YEE, K.-P. User interaction design for secure systems. In *International Conference on Information and Communications Security* (2002).