

# Password-Capabilities: Their Evolution from the Password-Capability System into Walnut and Beyond

Ronald Pose

*School of Computer Science & Software Engineering,  
Monash University,  
Clayton, Victoria 3168, Australia.*

Ronald.Pose@infotech.monash.edu.au

## Abstract

*Since we first devised and defined password-capabilities as a new technique for building capability-based operating systems, a number of research systems around the world have used them as the bases for a variety of operating systems. Our original Password-Capability System was implemented on custom built hardware with a novel address translation and protection scheme specifically designed to support password-capabilities. The password-capability concept later formed the basis of Opal developed at the University of Washington, and Mungi from the University of New South Wales, both of which used commercially available hardware. A second generation password-capability based system, Walnut, was developed at Monash University in the 1990s. Walnut was designed to run on commercially available hardware. It addressed some shortcomings of the original Password-Capability System but had to sacrifice some features that depended on hardware support. A third generation system that will extend Walnut to support mandatory security policies and other advanced features is currently being considered. This paper analyses the evolution of the Password-Capability System into Walnut, examines the shortcomings of the systems, and identifies issues to be addressed in the new system.*

## 1. Introduction

In the early 1980s a new form of capability system was devised, password-capabilities, which provided the benefits of capability based naming, addressing and protection but did not require hardware support for tagging or require the segregation of capabilities from other user data. Our first implementation, the Password-Capability System [1], used specially designed hardware to support password-capabilities in a tightly-coupled multiprocessor environment [2,3].

Until our invention of password-capabilities capability-based computer systems tended to rely on special purpose hardware which supported tagging of memory to distinguish capabilities from other data or code, or special purpose hardware or operating system mechanisms to segregate capabilities from other user data [4]. This hardware and operating system support was necessary because capabilities defined the protection and access rights for objects in a shared virtual memory, hence they had to be protected from alteration or forgery by those who wanted to gain unauthorized access.

Password-capabilities rely on a different paradigm for their security. A password-capability provides probabilistic security based on the extremely low probability of guessing a 64-bit random number by which all password-capabilities are protected. In effect, having a 64-bit random component in each capability forms a very large, but sparsely populated virtual address space, in which all objects exist. While access rights are traditionally incorporated within each capability, we chose not to encode the access rights within password-capabilities. To do so would have required a form of cryptographic encoding to prevent unauthorized alteration of the capabilities. Amoeba, another system employing a sparsely populated address space, uses cryptographic protection but required support for this scheme [5]. It would also have increased the size of capabilities, which in the case of password-capabilities, are already 128 bits in size. Instead, a password-capability itself identifies the volume on which the object resides, identifies the object itself on that volume, and has a random password to avoid forgery and unauthorized use. The system inspects the volume to obtain details of the allowed access rights associated with a password-capability.

Because password-capabilities contain no sensitive encoded information about access rights and because they are intrinsically secure, they may be freely intermixed with normal user data and code. No segregation or

tagging is necessary. Thus it is also possible to implement operating systems based around password-capabilities without requiring special purpose hardware. Walnut [6] from Monash University, Opal [7,8] from the University of Washington, and Mungi [9] from the University of New South Wales, were based on password-capabilities but implemented on commercially available hardware platforms.

While password-capabilities were devised with a conventional hardware platform in mind, the original Password-Capability System was developed in conjunction with a tightly-coupled multiprocessor which had novel hardware features to assist with multiprocessor operation and which also aided the efficient implementation of the Password-Capability System [10]. In particular, the novel address translation mechanisms not only provided efficient multiprocessor operation, but also allowed efficient implementation of fine grained protection, down to the level of individual words.

The Monash Multiprocessor and its Password-Capability System were quite radical for their day and performed up to our expectations, however they were by no means perfect. The Password-Capability System had a very simple process model and interprocess communication mechanisms. The address space seen by a process was also very simple. While this was adequate as a building block for more sophisticated systems, it was thought that a new second generation password-capability based operating system could provide a more powerful environment. The new system, Walnut, was also to be written in a hardware platform independent manner, so that it could be ported to a variety of different computer systems with relatively little effort. Walnut has been implemented [6] and runs on IBM compatible PCs in our initial implementation.

The Monash Multiprocessor hardware has also become dated in terms of performance and in terms of scalability to very large systems. A new massively parallel system has been devised [11] but not yet implemented. It is intended that Walnut or its successor will be the operating system employed on the new hardware.

Until now there has been no publication of the motivation for Walnut or an analysis of its design, advantages, disadvantages, merits and shortcomings. This paper in the short space available gives a brief overview of password-capabilities. It goes on to describe briefly their implementation in the Password-Capability System, and then goes on to describe how the model was extended in Walnut and its implementation strategy. Finally mention is made of Walnut's inadequacies and other issues which will be tackled in a third generation password-capability based system.

## 2. Password-Capabilities

The architecture defines a 'virtual memory' that comprises all processes, files, programs, data stacks, input/output channels and other resources to which a process may refer. Each of these is regarded as an 'object'. Hardware units used to support the architecture are not objects. Thus processors and memory modules are not objects and do not appear in the virtual memory. It is of course possible for hardware resources to be memory-mapped and be controlled via the capability system, but this would not be used for volumes containing 'objects' for obvious security reasons.

Every object may be regarded as a set of consecutively-addressed 32-bit words, but in many contexts a user's view of an object is more abstract. For instance, to send a message to a process, it is not necessary (or indeed permitted) that the sender have direct access to the words of that process.

No distinction is made between permanent data such as files and transient data such as a Fortran common block. By including all objects in a uniform virtual memory, a single mechanism can regulate all uses of objects.

A capability scheme is used. In this system, a capability is a 128-bit string, called a password-capability, knowledge of which permits some sort of use of some object.

A password-capability is divided into a 32-bit volume identifier naming the volume (typically a disc pack) containing the object, a 32-bit serial number uniquely identifying the object in the volume, and a 64-bit randomly chosen password. The volume identifier and the serial number form a unique object name that will never be reused in this or any other system using the architecture.

Several different capabilities for the same object may be valid. Each has the same object name but has a different password, and each may confer some subset of the possible rights over the object. For instance, one capability for an object may permit reading of words 105 to 142, another may permit alteration of word 73, and a third may permit the object to be executed as a program.

The capabilities defined for an object are organized in the form of a tree, with the root of the tree being the master capability returned when the object was created. Capabilities derived from existing capabilities must have lesser or equal access rights and are fully dependent on the existence of their parent.

The particular part or aspect of an object which may be reached via a particular capability is termed the 'view' given by the capability. When a view provides read or write access to some consecutive words of an object, the

view is said to define a 'window', within which the accessible words appear to be numbered from zero upwards.

To perform operations on an object, a user or a process is required to present to the system a capability empowering the operation on the object, and other necessary parameters such as an offset within the window.

The 64-bit passwords in the capabilities are randomly chosen. The probability of guessing a valid capability is smaller than that of guessing a user's password in a conventional system. The password is much more random than a user chosen password of the same length.

In a conventional system, unauthorized use of a person's password leaves all that person's files open to the intruder. With the password capability system, there is a very much smaller chance of the security of a single object being compromised.

### 3. The Password-Capability System

Processes refer to words or bytes in the virtual memory via virtual addresses comprising a 128-bit password-capability and an offset within the window defined by the capability. The use of long virtual addresses presents some implementation difficulties. The sheer length of the virtual addresses would require expensive address calculation and translation hardware if conventionally handled. Also, programs could become unwieldy were such addresses to be manipulated.

It is interesting to note that the Password-Capability System does not suffer from exhaustion of the address space that is possible in single address space operating systems (SASOS) that do not re-use the address space. This is due to our virtual addresses being around 160 bits long, thus our address space is much larger than most SASOS implementations. In fact the probabilistic security derived from the randomly allocated sparse virtual address space is not severely affected should one re-use parts of the address space.

In the Monash Multiprocessor, two techniques have been used to solve these problems. To avoid programs having to generate very long virtual addresses for every memory access, the processors are each equipped with 32 window registers. The operation of window registers is similar to the operation of the capability registers found in some other architectures [4].

A program wishing to read or write words of an object must make a kernel call passing a capability defining a window on the object, and a window register number. The kernel software checks the validity of the capability and sets the named window register to become a bounded pointer to some contiguous set of words in the object.

Thereafter, any logical address generated by the processor in a read or write operation whose top five bits name that window register is interpreted by special hardware as addressing some word within the window. The particular word (or byte) is found by treating the low-order bits of the logical address as a byte offset from the start of the window.

Thus a program need present a capability only once to set up a window register, and may then make any number of references to words within the window using only logical addresses. Up to 32 windows on different objects may be simultaneously accessible. Every access via a window register is checked for conformity to the access rights and window size bound set by the capability that was loaded into the window register.

The technique of using window registers has reduced the size of addresses manipulated by programs by mapping windows into a program's logical address space. Logical addresses may be manipulated in the same way as virtual addresses are manipulated in conventional virtual memory systems.

Another technique is required to enable the hardware to determine the physical location of the addressed word in the memory units of the multiprocessor given a logical address. The representation and interpretation of the 'bounded pointer' held in the window register is of concern here.

Given that the memory units are used as a pool of page frames loaded on demand by a fairly conventional demand paging algorithm, based on fixed-size 4096-byte pages, the window register cannot hold simply a physical address of the first word in the window. This is because it is unreasonable to demand that the window, which may be very large, should occupy contiguous physical memory. In fact, the window may exceed the size of the physical memory.

A number of schemes could have been used. For instance, a conventional page table could be maintained for each object, as is done in many segmented memory architectures. The window register would then hold the (physical) address of the page table for the object, an offset defining the first word of the window relative to the start of the object, a window size, access rights, and perhaps a cache of physical page addresses for recently used pages within the window.

Such a scheme was certainly practicable but in a multiprocessor with a persistent, global virtual memory it raised non-trivial problems in ensuring that the physical addresses held in all window registers in all processors of the system were kept up-to-date as pages and objects were swapped in and out of physical memory.

The problems involved in keeping all the window registers in the multiprocessor consistent are made

manageable by use of an additional address space between the virtual address space spanned by capability/offset addresses and the physical memory address space. This additional address space is called the Intermediate Address Space (IAS).

The IAS is designed to be large enough to accommodate all objects of current interest to executing processes, yet small enough to permit any byte in it to be addressed by an address of convenient length.

When any object is first referred to by a process (typically by the process's loading a capability for it into a window register) the entire object is mapped into the IAS, if it is not already present. That is, it is allocated a range of consecutive IAS addresses. If a capability defining a window onto the object is loaded into some window register, the window register is set to contain the lowest IAS address of the window (the base address), the window size, and access rights.

To speed the recovery of information needed to set up window registers, the system holds in its memory an Active Objects Table, which for every object in IAS holds the IAS address and the size of the object, and view-defining information for capabilities to the object. This table acts as a cache for catalogue information about capabilities of current interest, and reduces the need to refer to the capability catalogues on disc volumes.

A processor can easily generate the IAS address of any byte of a window on the object given the IAS base address held in a window register and the byte offset. The translation simply consists of adding the offset to the base address held in the window register. Bounds checking is performed by comparing the offset with the size field of the window register and access rights checking is performed by comparing the processor status with the access rights field of the window register. This translation, and the associated bound and rights checks, are performed by special hardware. The bounds checking is handled by the same adder which calculates the IAS address. In order to do this the complement of the window size is actually stored in the window register.

To avoid the need to keep numerous page translation caches up-to-date, processors make no attempt to translate IAS addresses to physical memory addresses. Instead, they place IAS addresses on the multiprocessor bus. Each memory module is responsible for recognizing IAS addresses in pages currently held in the module.

IAS addresses consist of two parts, an IAS page number and a within-page-address. Each memory module must check every IAS address on the bus to determine whether the IAS page number refers to a page contained within it. Ideally a content addressable memory would be used to perform this check and determine the physical page numbers. However it is

infeasible to construct a content addressable memory which can cycle at the speed of the bus.

It is not acceptable to use a slower content addressable memory since system throughput would be dramatically reduced. Instead a 4096-entry hash table is used. The hash table is addressed by the least significant 12 bits of the IAS page number. The remainder of the IAS page number is compared with that in the hash table. If they match, a physical page number also contained in the hash table is concatenated with the within-page-address to give a physical address. If there is no match, the memory module ignores the address.

A page fault is taken by the processor if every memory module ignores the address. No overflow chaining takes place in the hash tables since there is insufficient time to follow an overflow chain before the next address appears. Without overflow handling, IAS pages with the same least significant page number bits cannot appear together in a memory module. This hash clash problem is rendered unlikely because of the 16:1 sparseness of the hash tables. Special operations allow processors to manipulate the hash tables when handling page faults.

A great advantage of this scheme is that every IAS address referring to a page in physical memory is guaranteed to be translated in 50 ns. The total virtual to physical address translation time is thus only 150 ns.

Objects cannot retain their IAS addresses indefinitely. Eventually some objects have to be removed from IAS to make room for others. Hence IAS addresses held in window registers and used as keys in processor caches can become out of date. However, the IAS is so large that a very simple approach is feasible. Starting from an initially-empty IAS, objects are allocated contiguous blocks of space (page aligned) as necessary. Only after several hours will the space become exhausted. At that point, all objects are removed from IAS, all window register bases are marked invalid, and all caches cleared.

As processes attempt to use invalidated window registers, the original capabilities associated with the registers allow the system to re-map the objects still in use into the now-empty IAS, and to set up new window base addresses. The overheads involved in revalidating the window registers are not as great as were involved in initially loading the window registers.

Note that only the kernel has any knowledge of the existence or size of the IAS or any IAS addresses. The choice in the prototype of a 34-bit IAS address length is merely an implementation detail, not part of the visible architecture.

Use of an Intermediate Address Space also confers other advantages. In a tightly coupled multiprocessor it is possible to have a number of frequently used pages in the

one memory module and hence have contention among the processors for the use of the memory module. An appropriate course of action in such circumstances is to distribute the popular pages among a number of memory modules thus spreading the load. In conventional systems every processor would have to be informed of changes to the physical locations of the pages so it could keep its page tables or translation lookaside buffers up to date.

With processors only knowing about IAS addresses, the physical locations of pages may be changed without regard to the processors. Conversely, if it was desired to change an object's IAS address, it would be possible to retain the pages in the same physical memory locations and just alter the hash tables to reflect the new position in IAS.

## 4. Walnut

Having just espoused the advantages of the IAS and the implementation strategy of the Password-Capability System, we will now revert to more conventional techniques for Walnut.

While the Monash Multiprocessor and its Password-Capability System performed up to our expectations, the pace of microprocessor development was accelerating rapidly, and our multiprocessor was rapidly overtaken in performance by desktop Unix workstations. It became obvious that one must leverage on the latest processor technology for both new parallel hardware designs and as a platform for supporting our operating system. Hence we decided to undertake a second generation password-capability based system which could be supported, not only on currently available commercial hardware, but which would be likely to be easily ported to new, faster hardware as it became available. Walnut [6] is the result of this undertaking.

In the space available it is not possible to give a complete description of Walnut and its implementation strategy, however we will outline the main departures from the original Password-Capability System architecture and try to justify these design decisions. In hindsight, some of the Walnut decisions were unwise. Mention will be made of these in the next section.

### 4.1 Walnut design philosophy

The Walnut kernel was designed to be portable across a wide range of microprocessor architectures. To that end the kernel avoids using processor specific features where there are more widely available mechanisms. A paged memory management scheme was adopted as the basic access control mechanisms because of the many processors which support a demand paging scheme. This

resulted in page sized protection granularity rather than the word granularity of the Password-Capability System.

The kernel architecture was designed to scale from uniprocessors up to massively parallel multiprocessor systems. To accommodate this, device drivers and normal user processes use shared memory to communicate with the kernel. This is analogous to special purpose processors sharing memory with a master controlling processor in a multiprocessor configuration. Interrupts of course exist at the lowest hardware level, but are quickly abstracted away by the Walnut kernel which presents a uniform scheme using subprocesses to handle exceptions and other scheduling matters.

To increase the efficiency of multiprocessor implementations interprocess communication was minimized. Page tables and other system data are expired periodically and new tables constructed, rather than attempting to communicate the required updates. This practice of timing out data ensures that local data is kept up to date, and only a minimum of local state is retained. Furthermore, this practice eliminates the need to inform other processors of changes in local information. Rather than retain large amounts of state information, Walnut invokes a subprocess to handle the exceptional event, and the process simply retries whatever it was doing, hopefully after the exception causing situation has been dealt with.

Walnut differs in semantics from the Password-Capability System in that it introduces operators for the selective removal of rights from a capability and mechanisms that restrict the use of a capability for a specific purpose. Message passing and sub-process mechanisms have been introduced to enhance the handling of asynchronous events. The motivation for this was from requirements of programmers using the Password-Capability System.

Password-Capability System objects were essentially contiguous sets of words addressed from zero upwards. Walnut objects are more like address spaces. They have a maximum extent and a chosen allocation of space, but they may have areas that are not allocated. i.e. objects may have holes in them. Objects also have a defined limit on the number of capabilities that may be defined for them at any time. In other words you specify the maximum address valid for an object and the maximum space it can occupy, which may be less, and pages are allocated on demand as areas are accessed, up to the maximum space allocation. If there is space available on the volume it is possible to increase the space occupied by an object beyond the predefined maximum, but this is not guaranteed.

Another significant change has been made to objects that are processes. Whereas the Password-Capability

System process objects defined a logical address space in which the top 5 bits of the address specified a window register, with the remaining bits being a within-view address, in Walnut the logical address space is partitioned into a kernel-only area and a user area, and the user area has part of it reserved for small objects and part for large objects. The small window area has objects beginning on 4-kbyte boundaries. The large window area has objects beginning on 4-Mbyte boundaries. Objects can begin at any 4-kbyte boundary in the logical address space, so the program must keep track of where things are loaded. There is thus no notion of window registers. This removes the restriction of only allowing 32 concurrently addressable objects but involves extra housekeeping by both the user code and the kernel. The motivation for this was in part to make more efficient use of the logical address space and partly due to not having window registers available in the target hardware platforms. Having the kernel memory in the same logical address space in Walnut simplifies the Walnut kernel since it can see user objects without any addressing contortions. Kernel data structures are not generally readable or writeable by user processes. One of the reasons for having a separate kernel address space, invisible to users, in the Password-Capability system was that the initial target processor did not implement a full 32-bit logical address space, and we felt that address space and window registers were too precious to waste on effectively inaccessible kernel objects.

The process object itself is mapped into its own address space as the first of the large objects. Not all of the process object is readable or writeable by the process.

A more significant change to the semantics of processes in Walnut was the introduction of subprocesses. Subprocesses are independent execution threads which share the address space of the process. Thus there is essentially no protection between the subprocesses of a process; if one wants separate protection domains one should use separate processes. Each subprocess has a priority for scheduling purposes within the process and each has a mailbox for the receipt of messages. Rather than directing messages to a process, one can now direct messages to a particular subprocess. In effect one can treat subprocesses as event handlers and the messages rather like Unix signals. There is a sophisticated mechanism whereby the subprocess can specify what kinds of messages it is willing to accept. The net result is a system powerful enough to implement the equivalent of a sophisticated priority interrupt system for handling asynchronous events.

It should be noted that the subprocesses within a process operate as a set of cooperative coroutines rather than as concurrently executing threads. It is up to the

programmer to ensure that the subprocess stacks do not overlap or interfere in other ways since there is no protection boundary between them.

The interface to the Walnut kernel is also rather different from that of the Password-Capability System. Whereas the original Password-Capability System used a fairly conventional system call trap instruction to invoke kernel operations, Walnut uses subprocess 0 as a handler for kernel operations. A shared parameter block is used to pass parameters between user code and the kernel, and a message to subprocess 0 invokes the kernel operation. In this way it is possible for a process (subprocess) to invoke a kernel operation on itself or even manipulate another process given a suitable capability allowing it to send a message to subprocess 0. This should not be seen as a departure from the essentially in-process structure of the Password-Capability System. Rather it can be viewed as a different syntactic approach since at the lowest level of course some kind of trap instruction must be executed. This approach minimizes the amount of kernel state and facilitates extension to multiprocessor operation and to concurrently executing subprocesses should they be desired.

The parameter block is contained in a parameter page together with a message block. The parameter page is part of the process object which can be read and written by the process, and in fact forms the medium for communication with the kernel.

We have seen how the fundamental semantics of objects and especially processes and their interactions have changed significantly in Walnut. While we still use password-capabilities as the underlying naming, addressing and protection mechanism, the user's view of the persistent global virtual memory is quite different. It is a much more powerful system in some ways although it should be said that much of what has been achieved could also have been implemented using the Password-Capability System, somewhat less efficiently, at the level of user code.

## 4.2 The Walnut implementation

The differences between Walnut and the original Password-Capability System are not only visible at the level of virtual memory semantics. The underlying implementation itself is quite different and in part has driven some of the semantic changes.

The Password-Capability System was implemented on specially designed hardware that provided hardware for window registers and a novel address translation scheme designed to support efficiently shared memory multiprocessor operation over a shared communication medium. This made the implementation rather

straightforward in that there was direct support for the kinds of addressing required and no predefined address translation mechanisms in the hardware around which one had to work.

In contrast Walnut was implemented on an Intel 80486 architecture. i.e. a cheap PC. This meant that one had to make do with the addressing and protection mechanisms provided by the hardware. This presents many problems since the Intel virtual memory paging hardware was certainly not designed to support a persistent global virtual memory.

The Intel 80486 and its successors have a paged segmented addressing architecture. On the surface it would appear attractive to use the segmentation hardware as surrogate window registers. We rejected that on two grounds; such hardware is not available on most modern processors so relying on it would be an impediment to portability across platforms, violating one of our design principles, and the other ground for avoiding the segmentation hardware was that there were simply too few segment registers for a useable system.

The two-level paging hardware and microcode of the Intel machine was used. Two-level page tables are commonly found on modern processors so were not really an impediment to portability, however the Intel approach of having the hardware know the layout and operation of page tables certainly makes life difficult and increased that amount of machine dependent kernel code. Machines such as the MIPS RISC processors that have a hardware translation lookaside buffer (TLB) but which have the kernel code doing the loading of the TLB in software, give one the flexibility to organize page tables in a more optimal manner. In fact what one tends to do is keep kernel page tables and other tables separate from the Intel style page tables which are set to contain a relevant subset of the information.

A well known parameter page was used for communication with the kernel rather than passing parameters via registers or on the stack since it simplified the kernel interface especially when one has subprocesses involved in making kernel calls. There is no well defined process stack as such. Any object can be used as a stack. Our approach however meant that a mechanism was required to control the use of this parameter page since multiple subprocesses could potentially be trying to use it. The mechanism adopted was very simple. A bit in the parameter page indicates that it is in use for a kernel call. While that bit is set, no switching between subprocesses is allowed to take place. i.e. a kernel call is treated rather like an indivisible operation, an instruction surrogate.

Subprocesses are useful not only as message handlers but also as trap and exception handlers. The Walnut kernel detects exceptions and provides facilities to allow

subprocesses to be sent messages informing them of a trap and allowing them to take appropriate action. By default if no subprocess has been set to handle a trap or exception, the process is terminated. A subprocess cannot handle its own traps, so if a subprocess causes a trap which normally is directed to that subprocess, the process is terminated.

It should be noted that page faults and the like are not considered as traps or exceptions in Walnut. They are completely hidden behind our virtual memory interface. We are concerned here with things like floating point exceptions or undefined instruction execution.

The laying out of objects and their capabilities on a volume are quite different in Walnut compared with the Password-Capability System. In the Password-Capability System the serial number field of the capability merely served to identify the object. A catalogue on the volume was consulted by the kernel to determine whether an object existed and the validity of a capability. The catalogue contained the trees of capabilities for the objects stored in the volume. Associated with each object was also a page table that indicated where the pages for the object were on the volume. The catalogue itself was not an object in the system. i.e. no capability exists for the catalogue, so it is inaccessible to user processes.

In contrast, Walnut uses the serial number field in the capability as the disk address of the object's capability tree data structure. This is also where the page table information for the remainder of the object is stored. Linking the capability so tightly to a disk location may seem dangerous, but with modern disk drives which use logical block numbers rather than physical sector addresses, one is fairly well insulated from problems of bad blocks, and their re-mapping.

The Password-Capability System had a centralized catalogue on each volume which was consulted by the kernel when validating capabilities and locating objects, while Walnut uses a more distributed approach that localizes the information associated with an object to the area where the object itself is located.

Many other differences in the implementation approaches of the Password-Capability System and Walnut are also apparent but cannot be covered in a short research paper. Maurice Castro's Ph.D. thesis [6] is the best source for such information. Details of the programmers' view of the Walnut system may be found in [12].

Apart from the changes in system semantics introduced in Walnut, the Walnut implementation itself did not completely implement the model defined in the Password-Capability System. In the case of protection, this was reflected in an official change from word granularity protection to page granularity protection.

This leaves things a bit vague if page size varies as it does between processors, although 4-kbyte pages are currently popular. One could argue that word granularity was also a bit vague. This is now evident with the increasing popularity of 64-bit processors. Rather more controversial was the failure to implement explicit execute permission control. i.e. one can control read and write access, but execute access is assumed to be allowed if you can read the page. The problem occurs when you want to allow execution of code in an object but not allow the object to be read or copied. This was a pragmatic decision based on the lack of support for execute access rights in the Intel 80486 architecture. Workarounds can be done but were not implemented for reasons of efficiency.

## **5. Comparison of the Password-Capability System with Walnut**

The Password-Capability System was designed as a minimal kernel that provided naming, addressing and protection mechanisms but specified very little in terms of policies for their use. The aim was to protect users and processes from one another and provide a flexible platform for the development of applications in a distributed and parallel environment. To this end it was fairly successful. We managed to demonstrate its effectiveness even to the extent of being able to support Unix on top of it.

A particularly interesting aspect was the ability to be able to support sophisticated discretionary security policies at the user level. Even difficult areas such as information confinement and charging for user services in a secure manner [13] were achievable using the system.

Some of these more advanced uses of the system did however rely on the provision of means to be able to execute code without being able to read it. For instance one might want to have the code contain some secret key which it uses during its execution, but which should not be known to the caller. While the Password-Capability System provided the required mechanisms, Walnut has omitted some of these. It would be possible to implement these in Walnut, but at the cost of some efficiency due to lack of hardware support.

The loss of fine grain protection in Walnut is probably not such a major concern. While there are obvious applications where fine grained protection is useful, being forced to set protection boundaries on page boundaries doesn't really make things impossible, although there could be some wastage of memory.

What is much more worrying is the exposure of the underlying page size to the user. We live in a world where data is outlasting the programs that created and

first manipulated it, and certainly outlasting the hardware in which it is manipulated. This was evident with the much publicized Year 2000 'bug'. It is of some concern that data structures and applications may be built which depend on a certain page size for protection granularity, and in the future that may not be supported. The trend is for page size to increase as memory costs and disk costs come down and memory and disk wastage is of less concern.

The introduction of subprocesses into Walnut was something we had contemplated for many years as a way of handling in a clean manner asynchronous events, traps, exceptions and the like. They certainly make life easier if one wants to handle such events, but it remains to be seen whether we have the best model in Walnut. Walnut subprocesses should thus be considered as exception handlers and event handlers, rather than as execution threads. Certainly Walnut provides no mechanism to allow a process's subprocesses to execute in parallel. Another interesting decision was that to have indivisible kernel calls at the subprocess level. This is a tradeoff between simplicity and potential parallelism which needs further evaluation for future large scale multiprocessor systems. This design decision makes sense in the context of subprocesses not executing in parallel, but could indeed be a bottleneck otherwise.

Communication with the Walnut kernel via a parameter block in the process's address space rather than via the stack or registers and a system call instruction, was a deliberate decision which allows for processes to invoke kernel operations on other processes given appropriate capabilities. Of course the use of a fixed area of the address space has implications for polluting the cache. This is an issue we must evaluate for future systems.

While it seems that we have increased the complexity of the system considerably in Walnut compared with the Password-Capability System, it is still a fairly lean system. Some things have been moved into the kernel but in general the aim was to minimize what went into the Walnut kernel.

## **6. Future developments**

Both the Password-Capability System and Walnut handle quite effectively and elegantly the standard problems involved in discretionary security models. What is missing, partly due to the flexible and open philosophy of the system, are means of enforcing mandatory security models such as the Bell LaPadula rule based model [14].

Monads and its successor, SPEEDOS [15], currently being developed at the University of Ulm, were based on an information hiding module concept in which data

structures were encapsulated along with their semantically appropriate access routines rather than being free-standing entities. This has apparent advantages in terms of ensuring correctness, modularity and robustness of large software systems. The Password-Capability System appears to have adequate mechanisms to support similar information hiding modules at the user level. Walnut could easily be adapted to support similar modules.

There are however some deficiencies in the support required to implement rule-based mandatory security policies. These are being tackled in the SPEEDOS system [15] through the use of 'bracket' routines and a more sophisticated confinement mechanism than is possible with the existing Password-Capability System.

It appears that the changes required to the Password-Capability System and Walnut are not enormous and we plan that a third generation system based on password-capabilities will be developed to handle these and the other shortcomings mentioned above.

## 7. Conclusion

A brief overview of the concept of password-capabilities was presented. We then looked at its first implementation, the Password-Capability System. Its successor, Walnut, was designed with criteria which aimed to make the system more efficient and practical so that we could increase our user base and get experience with users running such a system. We looked at how Walnut dealt with off-the-shelf hardware rather than the custom-designed hardware platform used by the Password-Capability System. Both these systems are not perfect, and we identified some of the issues to be tackled in the next generation password-capability based system. This paper is limited to discussion of password-capability based systems. Other issues relating to capability systems and to SASOS systems warrant further investigation.

## Acknowledgements

The Password-Capability System, the Monash Multiprocessor on which it is implemented, and Walnut were all supported by ARGS and ARC research grants.

In 2000 I am visiting the SPEEDOS development group led by Les Keedy at the University of Ulm. They have contributed to the comparison of facilities and mechanisms of SPEEDOS and the Password-Capability System, and provided the forum where it became obvious that many of the details of the Password-Capability System and of Walnut have never been published over the many years in which they have been used.

This paper is the beginning of a long overdue evaluation and exposition of Monash University based password-capability research.

Insightful comments by the referees have been invaluable in improving this paper.

## References

- [1] Anderson, M.S., Pose, R.D. and Wallace, C.S. (1986) A Password-Capability System, *The Computer Journal*, Vol 29, No. 1, pp. 1-8, 1986.
- [2] Pose, R.D., (1989) Capability Based Tightly Coupled Multiprocessor Hardware to Support a Persistent Global Virtual Memory, *Proc. 22nd Hawaii International Conference on System Sciences*, Vol. 2, pp. 1-10, 1989.
- [3] Pose, R.D., Anderson, M.S. and Wallace, C.S. (1987) Implementation of a Tightly-Coupled Multiprocessor, *Australian Computer Science Communications*, Vol. 9, No. 1, pp. 330-340, 1987.
- [4] Abramson, D. (1982) Computer hardware to support capability based addressing in a large virtual memory, Ph.D. thesis, Monash University, 1982.
- [5] Tanenbaum, A.S. and Mullender, S. (1981) An overview of the AMOEBA distributed operating system, *Operating Systems Review* Vol. 15, No. 3, 1981.
- [6] Castro, M.D., (1996) Walnut: A Password-Capability Based Operating System, Ph.D. thesis, Monash University, 1996.
- [7] Chase, J.S., Levy, H.M., Feeley, M.J. and Lazowska, E.D. (1994) Sharing and Protection in a Single-Address-Space Operating System, *ACM Transactions on Computer Systems*, 12(4), November 1994.
- [8] Chase, J.S. (1995) An Operating System Structure for Wide-Address Architectures, Ph.D. Thesis, Department of Computer Science and Engineering, University of Washington, August 1995.
- [9] Heiser, G., Elphinstone, K., Vochtelo, K., Russell, S. and Liedtke, J. (1998) Implementation and Performance of the Mungi Single-Address-Space Operating System, *Software: Practice & Experience*, 28(9), pp. 901-928, July 1998.
- [10] Pose, R.D. (1991) A Capability-Based Tightly-Coupled Multiprocessor, Ph.D. Thesis, Monash University, 1991.
- [11] Pose, R.D., Wells, J. and Fazio, V. (1997) Self-Tuning Paradigm for a Distributed Multiprocessor System with Flexible Interconnection, *Computer Architecture'96*, pp. 183-196, Springer-Verlag, 1997.
- [12] Castro, M. (1995) The Walnut Kernel: User Level Programmer's Guide, Technical Report 95/222, Department of Computer Science, Monash University, May 1995.
- [13] Wallace, C.S. and Pose, R.D. (1990) Charging in a Secure Environment, *Security and Persistence*, WIC Springer-Verlag, 1990. Ed. John Rosenberg and J. Leslie Keedy, ISBN: 3 540 19646 3, pp. 85-96.
- [14] Bell, D.E. and LaPadula, L.J. (1973) *Secure Computer Systems: Mathematical Foundations*, Mitre Corp., Bedford, Massachusetts, 1973.
- [15] Keedy, J.L., Espenlaub, K., Hellmann, R. and Pose, R.D., (2000) SPEEDOS: How to Achieve High Security and Understand It, *Proc. CERT Conf. 2000*, Omaha, Nebraska, USA, .Sept. 2000.