S-RISC: Adding Security to RISC Computers

J.L.Keedy
keedy@jlkeedy.net
formerly Professor/Honorary Professor at the University of Newcastle, NSW and Monash University, Melbourne
the Technical University of Darmstadt, the University of Bremen and the University of Ulm in Germany

Abstract

Over many years computer architecture researchers have sought ways of combining the efficiency of RISC computers with the security potential of capability systems, in some cases in multi-million dollar projects. The paper shows how a model for protecting both small and large segments in a paged system, first presented in 1980, can easily and straightforwardly be adapted for use in a RISC environment (without the use of tagged memory) and thus achieve this aim. The design was originally implemented in the MONADS (CISC) systems in the late 1970s.

The crucial difference from current RISC systems is the way in which addresses are translated by the hardware. Relatively trivial changes to compilers would enable current applications to run unchanged, but the modified RISC hardware would also make possible the development of secure capability systems, as the MONADS project has already demonstrated.

The paper begins with a discussion of protection mechanisms in capability systems, then outlines the essential features of RISC systems and shows how these two aims can be combined in S-RISC (Secure RISC) systems. It then discusses various issues in more detail and concludes with a discussion of related work.

1 Introduction

Early attempts to design highly secure computer architectures were basically of CISC design and could not compete in efficiency with that of RISC processors. Since then a combination of RISC design and ever improving chip miniaturization and technology has enabled computers to achieve amazing performance improvements, but as Hennessy and Patterson, who coined the name RISC, have observed:

> "Security and privacy are two of the most vexing challenges for information technology in 2011. Electronic burglaries, often involving lists of credit card numbers, are announced regularly, and it's widely believed that many more go unreported. Hence, both researchers and practitioners are looking for new ways to make computing systems more secure. Although protecting information is not limited to hardware, in our view real security and privacy will likely involve innovation in computer architecture as well as in system software." [1, p. 105].

Hardware and architectural improvements cannot guarantee that the overall security of computer systems will be improved, because the final result depends not only on the mechanisms, but also on strategies and policies and also on their error free implementation. However, without a core of good basic mechanisms on which to build it is more difficult to devise good strategies which can be implemented efficiently, cheaply and correctly.

This paper describes a mechanism for combining the efficiency of RISC systems with the security of capability systems. (Here the word "security" is used at the architectural level, not in terms of correctness proofs or of encryption/public key security.)

2 Capability Based Operating Systems

In the pre-RISC era the most promising security strategies at the architectural level were undoubtedly found in the early capability systems, e.g. [2, 3, 4, 5]. A capability, first proposed by Dennis and van Horn [6], is a data structure describing access rights which the presenting process can exercise over a defined object (e.g. for memory segments read and/or write and/or execute) and can be used to check the bounds of the object (thus preventing accesses outside

the segment). Most early capability system designers chose variable length segments of memory as the objects to be protected; these allow different processes to have separate capabilities for the same segment, which could, if appropriate, have different access rights[1]. Individual segments might be small or large. Hence a capability system which supports both is ideal, especially since capabilities themselves are normally relatively small objects.

Because they contain sensitive information, capabilities themselves must be protected. The system must in some way guarantee both that existing capabilities cannot be modified in arbitrary ways and that new capabilities cannot be arbitrarily manufactured or forged by unprivileged users. Several capability protection mechanisms were proposed and used in early systems. At the time of their development the Internet did not exist. Hence a newer criterion for judging these must be their adaptability to the Internet environment, in particular to a system in which capabilities for the same object can be distributed to users at different nodes.

2.1 Protection in the Operating System Space

An obvious solution is to protect capabilities in a similar way to that by which segment and page table entries are protected in conventional systems, i.e. by storing them in capability lists (C-Lists) in the operating system's private data space. In this case the creation and modification of capabilities can only be carried out by the operating system, at the request of users (via calls which the operating system can check for validity).

The main disadvantages of this solution are a performance overhead when the user accesses his capabilities and a lack of flexibility in the way they can be organized into lists. Furthermore this solution cannot easily be scaled up for general use over the Internet.

This approach is rather like having a bunch of keys which is compulsorily held by a hotel porter who will open and close your room for you whenever you ask him or her, but you always have to go to him or her to lock/unlock your room.

2.2 Password Protection

A technique known as password capabilities, implemented in the Monash Capability System (not to be confused with the MONADS systems), allows capabilities to be stored in the user's address space while using conventional computer hardware [7]. Rather than holding a set of access rights the capability in this case contains a password, which is a large integer value (see Figure 1).

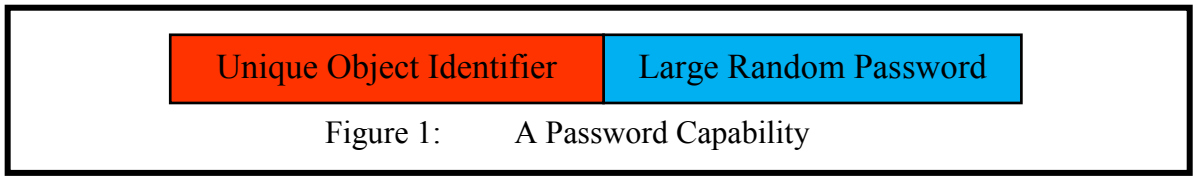

Figure 1: A Password Capability

This password, which in the Monash Capability System was 64 bits long, is a system generated random number. The idea is that the number must be large enough that it cannot easily be guessed or systematically generated. The capability is stored in user space and is not protected from being modified using normal instructions. (That is why it works using conventional hardware.) However, it can only be used as a capability in calls to the operating system, and at that point its validity is checked.

The operating system has an internal table in which the object name, the password and the permitted access rights are stored. When it receives a request to carry out an operation the operating system checks the object name and password fields against entries in its table. If it finds a match, the capability is valid for the access rights stored in the table. If the requested

[1] This contrasts with conventional systems, which place the access rights in page tables or segment tables and thus determine that all users share the same access rights.

operation conforms to these access rights the operation may proceed (see Figure 2). Different sets of access rights have different passwords.

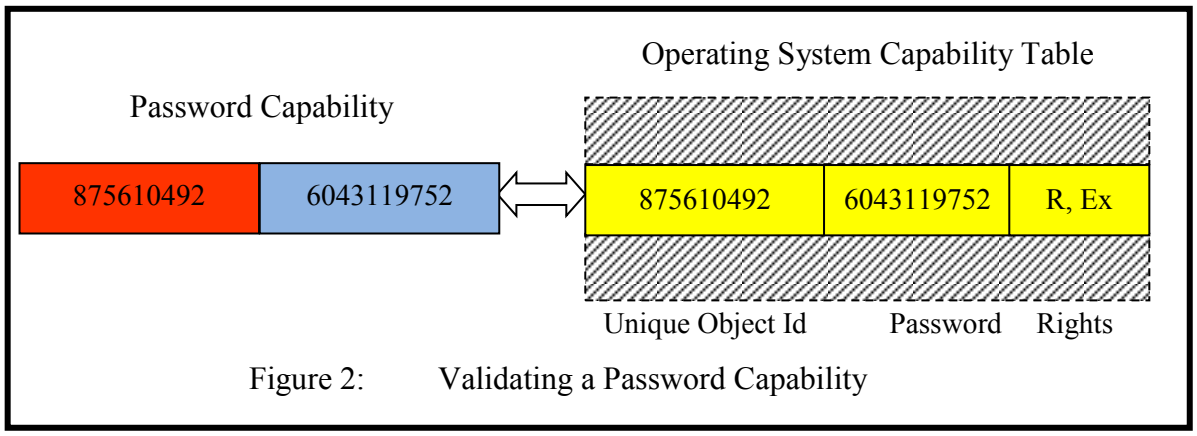

Figure 2: Validating a Password Capability

This solution has the advantage that capabilities may be stored flexibly in a user address space. But it implies that each use of a capability must be made via an operating system call and that there are additional tables in the operating system. This makes it difficult for example to store capabilities on an external storage device and use them later on a different computer.

The Mungi system [8] used password capabilities in a local area network system in which all the nodes shared a common range of addresses, such that the top bits of 64-bit addresses determine which node was involved. However, this technique cannot be scaled up to work in a general way on the Internet.

Password capabilities are like having a piece of paper with a password on it, which a hotel porter checks before taking a key to open your room door.

2.3 Protection by Tags

Another approach which allows capabilities to be stored freely in the application address space and which avoids the problems associated with password protection is to use tag bits to identify which memory locations hold capabilities. This was the solution adopted in the IBM S/38 computer system [9, 10], which was a latecomer on the capability scene. The tagging solution involves having one or more additional bits associated with each word in the main memory. If this hidden tag bit is set to 0 the remainder of the word is a normal data or instruction word, but if it is set to 1 the rest of the word is part of a capability. Because a capability might take up several words of memory, the address at which it actually begins can be recognized by its byte position, i.e. capabilities must start at fixed byte positions. The tag bit can only be set and unset when the system is in privileged mode.

With this solution the CPU can check, as it executes instructions, whether these are being applied to capabilities or to normal data and instruction words. To create a new capability the operating system must be called to set the tag bits. Similarly, attempts to modify a capability using normal instructions will be detected by the hardware.

Thus tagged capability protection is more flexible and more efficient than the earlier solutions, since protection tables in the operating system need not be consulted on every access, but it is achieved at the cost of an extra bit of memory for each word. Unfortunately this extra bit has to be copied to secondary memory whenever a program segment is discarded from the main memory to make room for another. Secondary memory blocks are usually organized into sizes which are powers of 2; this creates difficulties when words in memory have an extra tag bit. Hence it is unlikely that tagged capabilities will be universally accepted [11].

This solution is rather like having to use keys which are too big to fit into your pocket.

2.4 Protection by Partitioning Segments

A fourth solution is the use of *partitioned segments* [5]. Here the idea is to allow normal data and capabilities to coexist in a single segment, but to segregate them into two parts of the segment. Normal data words are addressed by positive offsets from a base register, while capabilities are placed at negative offsets, below the address at which the base register is pointing. The instruction set of the computer is so organized that negative addresses either cause an exception into the operating system, which can then validate the action required in relation to the capability, or that special capability instructions which use only negative offsets can be supported at the architectural level (see Figure 3). The base register must be a protected register which can only be modified by the system kernel.

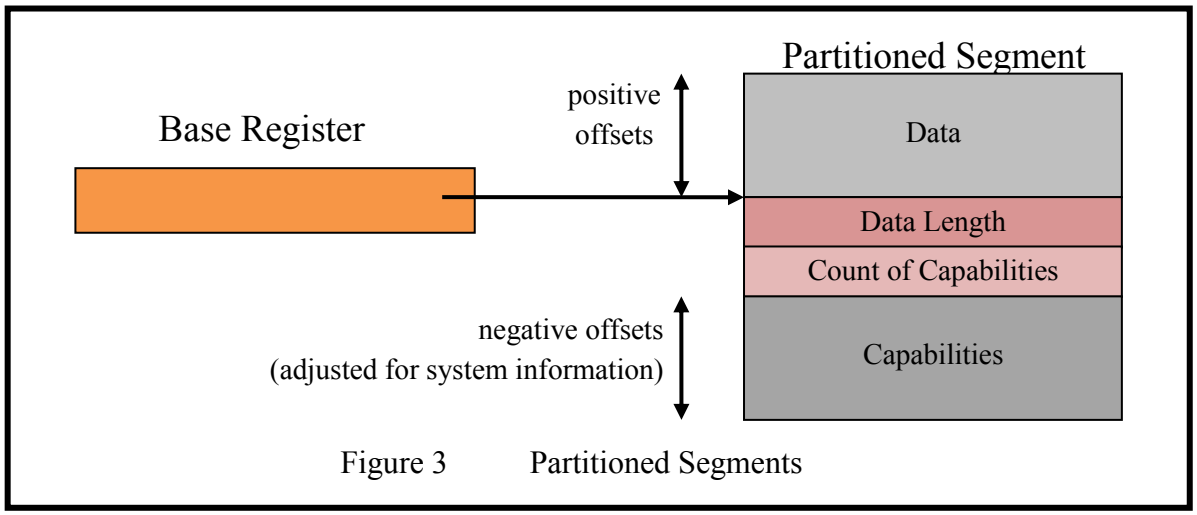

Figure 3 Partitioned Segments

Partitioned segments simplify the use of linked data structures in the application address space, and they have none of the disadvantages of the other solutions. There are no operating system tables or hardware tags. In this case the keys fit conveniently into your pocket and you can put them away as a bunch in a convenient filing cabinet or pocket, wherever is convenient.

2.5 Protecting Capabilities via Capabilities

Finally, there is another solution for protecting capabilities: use capabilities to protect other capabilities. This relies on the fact that another solution already exists, so it may seem to contain a circular argument. But if more than one kind of capability exists, as was the case in the MONADS-PC system [12, 13], then this solution also makes sense. This is like being able to lock a box which contains your keys for safe keeping.

3 RISC Based Systems

RISC systems rely on the use of a "load and store" instruction set architecture, which was first used in computers such as the CDC 6600 [14] and the IBM 801 [15].

The paper does not aim to describe how capabilities can be integrated into a particular RISC instruction set architecture (such as RISC-V[2]) but instead outlines a general model based on the RISC principles as set out in Hennessy and Patterson [1] (p. C-4), who invented the name RISC.

- Data operations are always applied to data in registers (32 or 64 bits per register).
- Only load/store instructions transfer data between memory and registers.
- Instructions have a uniform size with only a few instruction formats.

A large number of registers (typically 32) is usually available. In practice there may be an

[2] https://riscv.org

additional set of registers for floating point arithmetic.

ALU Instructions

As illustrated in Figure 4[3], ALU instructions operate on the values held in two registers or in one register and an immediate value; the result is stored in a third register.

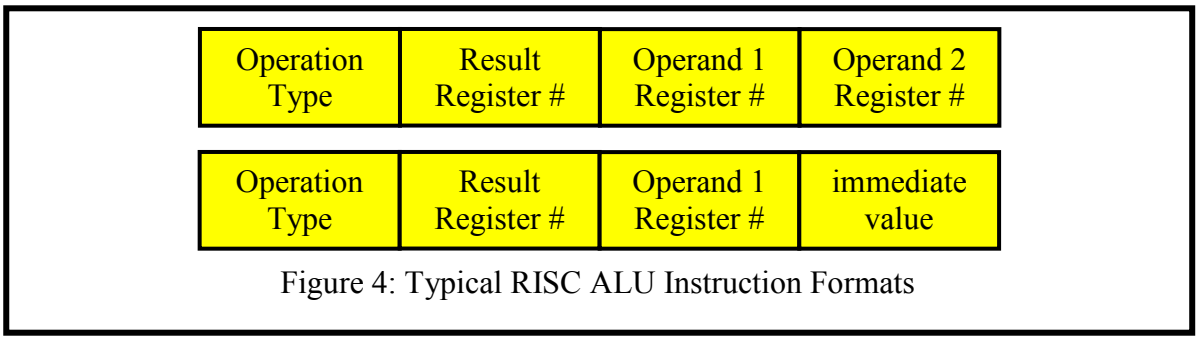

Figure 4: Typical RISC ALU Instruction Formats

Load and Store Instructions

The address held in a base register is added to an offset held in an immediate field or in an index register to produce an *effective address*, which is then used as the load or store address for the operand in the operand register (Figure 5).

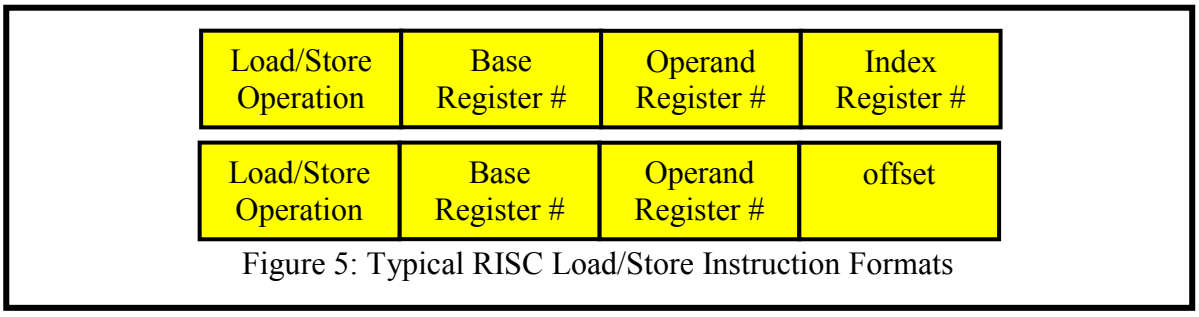

Figure 5: Typical RISC Load/Store Instruction Formats

Branch and Jump Instructions

These are used to control the sequencing of instructions (either conditionally or unconditionally). For conditional branches some RISC computers use condition codes, others use comparison instructions. The differences are not relevant to this paper. The branch destination is achieved by adding an offset to the current value of the program counter register.

4 Efficiently Combining Paging and Segmentation

In contrast with most of the earlier capability systems, which tend to use segmented memory, RISC systems tend to rely for their efficiency on the use of paged virtual memory. The issue is then to find a way of integrating capability systems into a RISC paging architecture.

The ideal situation would be to have a paged virtual memory with a mechanism for efficiently protecting both small segments (e.g. which can hold capabilities and also small program segments) and large segments (which also occur in programs). However, it is widely assumed that this is impossible. Eventually most computer architects, including RISC designers, decided not to support small segments. Some systems, such as the Borroughs B6700 [16], chose pure segmentation and made no attempt at supporting conventional paging. Others, such as Multics [17, 18] and the ICL2900 [19], implemented segmentation schemes in which the segments were large and paged; attempting to use these for small segments would lead to severe internal fragmentation and paging overheads. But most architects, including the RISC

[3] The following diagrams are oversimplified; in this general model issues such as lengths of the individual instruction fields are left open.

designers, chose to use pure paging.

Fortunately the quest to design secure architectures based on the RISC approach was not entirely abandoned, as we shall see in section 14. But this research has been hindered by the belief in the impossibility of combining small segments with paging.

In this section we show how fine grain protection can be attained in such a way that both small and large segments held in the same page can be separately protected while at the same time allowing segments to span multiple pages.

The author solved this problem and presented the solution in a paper presented at the 8th World Computer Congress, IFIP 1980 [20] in 1980. Soon after publishing the paper he concentrated mainly on programming language design rather than operating system design, although he and his former research students have published papers relating to the idea [21, 22, 23, 24, 12, 25, 26]. In view of the central role which this solution plays in the following proposal to integrate RISC systems and capability systems a description of the key mechanism now follows.

We start with two part addresses in the form «segment number, offset». Such "effective program addresses" say nothing about page boundaries and nothing about how the offset is derived. These addresses provide the starting point. They can be translated by reference to a segment table (see Figure 6).

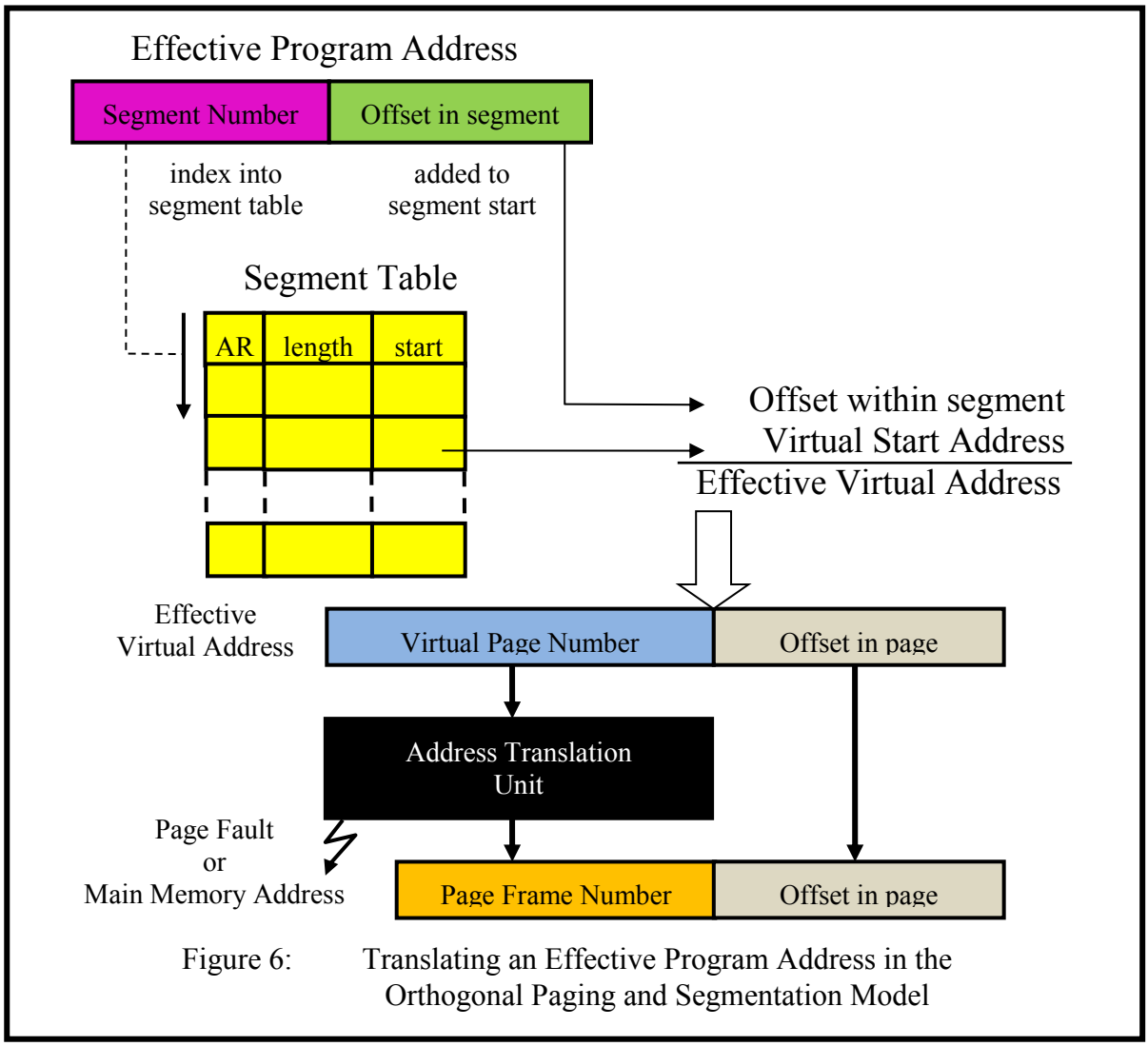

Figure 6: Translating an Effective Program Address in the Orthogonal Paging and Segmentation Model

But instead of regarding the entry in a segment table as containing either a main memory address or the address of a page table (as in conventional systems), we assume that it

contains a *virtual* address. This means that a distinction is being drawn between effective program addresses and virtual addresses.

The segment table holds a *virtual* address defining where a segment begins. To this is added the offset from the beginning of the segment, taken from the effective program address. This results in another address – this time the effective virtual address of the word to be addressed. That address must then be translated into a main memory address. Since the aim is to have a paged virtual memory, this translation can be achieved in one of the usual ways, by using either a conventional page table or an inverted page table, as is shown in Figure 6.

The segment table can contain information about the logical properties of the segment (i.e. its length and its access rights), while the ATU can hold information useful for paging, such as a use bit and a change bit.

To see what this means in terms of program layout, consider a simple program with three kinds of segments. Figure 7 shows how the segment table and the program both appear. From this it is clear that there is no difficulty in placing segments with different protection requirements adjacent to each other. It is also clear that a segment can span multiple pages and also that multiple segments can be placed in a single page, in any arbitrary combination.

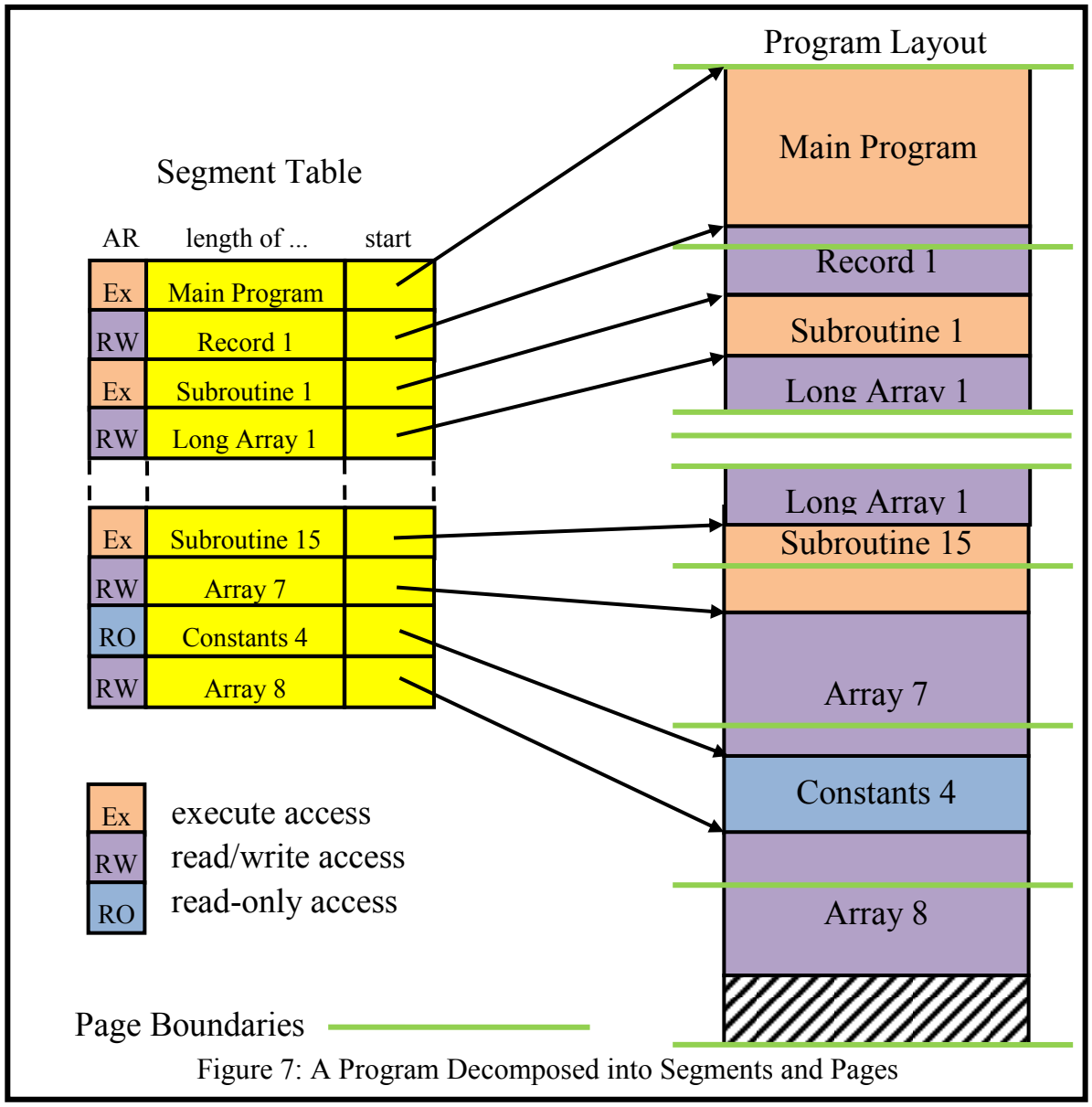

Figure 7: A Program Decomposed into Segments and Pages

Furthermore, the problem of internal fragmentation has been restricted to the final page. Like the conventional paging model, the orthogonal model achieves the minimum possible

fragmentation of a half page per program on average. This solution was successfully implemented in the MONADS-PC system in the 1980s [23]; this was a capability system, but not a RISC system.

The solution is an example of the principle of separation of concerns: the first concern is how segmentation can be organised; the second is how paging can be organised. We now consider how this orthogonal paging and segmentation model can be integrated into RISC systems.

5 S-RISC: The Proposed Mechanism

An important point about the orthogonal segmentation and paging model is that it does *not* define how the segment table is implemented. It only defines that entries should contain access rights, a start address and a length field. In a RISC environment the obvious way to provide this information is in an additional set of registers, which we here call "address registers". These have the format shown in Figure 8.

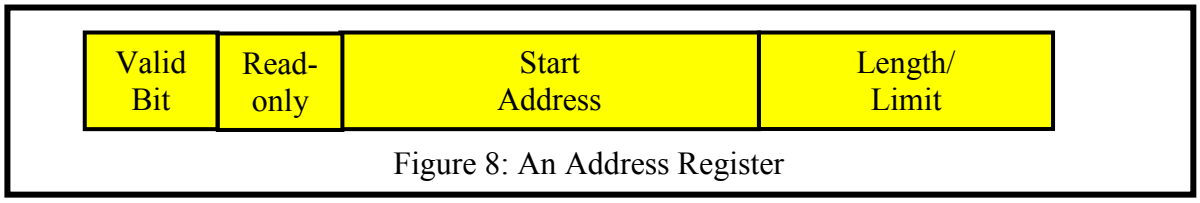

Figure 8: An Address Register

The start address is an address in the paged virtual memory, i.e. a virtual address in the sense of Figure 6. It need not coincide with the start of a page, and no further segmentation mechanism is needed at the hardware level, i.e. *a segment table in the conventional sense need not exist at all*. The length field defines the length of the permitted addressing range (typically a segment length). The address registers are used in load and store instructions, replacing the base registers shown in Figure 4.

The execution of a load or store instruction involves not only calculating an effective virtual address (using the start address as a base address) but also checking that this is in the range of permitted addresses. The length field is compared with the offset field or index register value in Figure 4. This comparison can be carried out in parallel with the generation of an effective address. Hence the speed of execution in the proposed model need not exceed that of a normal load or store instruction in a conventional RISC system.

Similarly the access rights field can be checked in parallel with the calculation of the effective address if the operation type field of load/store operations is appropriately encoded. (Loads correspond to reads, stores to writes.)

6 A Note on Indexing

Some RISC systems support indexing in load/store instructions, others (e.g. MIPS) do not [1] (pp. K5-K6). In the latter case, the proposed scheme could create inefficiencies for accesses to data arrays and, more generally, to items within a segment. These could arise because the instruction format which they use (the second format in Figure 4) would require a kernel call (see below) for each access to an array element, etc. to change the address register.

One solution is to add an indexing facility (either via a bank of explicitly addressable index registers or by using the general purpose registers for this purpose). In either case the required instruction format could take a form equivalent to the first format shown in Figure 4.

Alternatively it would be possible to define a fixed mapping between index registers (or a subset of the general purpose registers) and the address registers, cf. the technique used by the CDC 6600 [14]. In this case the appropriate value would be set in the index register before a load/store operation. If this option were chosen the first format shown in Figure 4 would not be needed, and using the second format would result in an operation which calculates an effective address by adding both an index value and an offset to the start address in an

address register.

7 Addressing Segments

Because the proposed segmentation scheme does not presuppose a particular hardware-defined table structure, the kernel can freely organise information about segments. We use the example of *partitioned segments* [5] (see section 2.4) to illustrate one possibility. The idea is that data and pointers (i.e. addresses of segments in an address space, used as capabilities) are stored in separate partitions of a single segment, which has a *red-tape* area defining the properties (e.g. lengths) of the two partitions (see Figure 9). A process can access segments via pointers, but only when the latter are loaded into address registers. The data partition of a segment can then be addressed using normal ALU instructions. Negative offsets cannot be used to gain direct access to the red tape area or to the pointers.

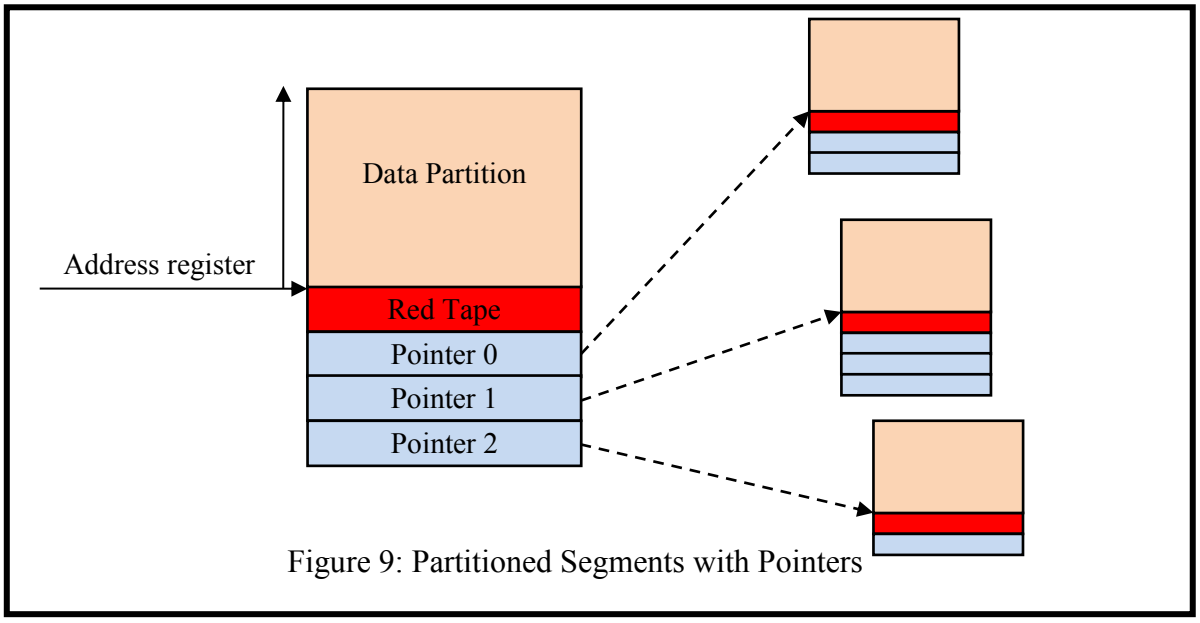

Figure 9: Partitioned Segments with Pointers

The first segment in an address space is created via a kernel instruction which has an operand specifying a length field for the data partition, the number of required pointers and an address register number. This instruction creates a root node and returns a pointer to the new segment loaded into the specified address register. More segments can be created by invoking a further kernel instruction using an existing valid address register, specifying a pointer number in its red tape area and providing as further operands the required data length and the number of pointers required in the new segment. The kernel's segment manager must ensure that overlapping segments are avoided.

Pointers can be followed by using a normal load/store instruction which uses a source address register number to specify a currently accessible segment and a pointer number for the destination segment. This instruction overwrites the current address register with the pointer value selected. The overwritten register can first be copied as a register-register operation if necessary.

Segments can where appropriate be set to read-only (i.e. the data partition can only be read, not written) by means of a kernel instruction which sets the read-only bit in an operand address register to read-only. This remains set until a further kernel instruction unsets it in the address register. In this way an entire linked list of segments can be set to read-only without having to use a bit in the individual pointers. Notice that because read-only status is a property of segment registers, the lists can be set to read-only for some threads/processes while remaining writeable for others.

Significant aspects of this and similar mechanisms are that pointers

- provide fine-grained protection (applying bounds checking);
- can have segments set to read-only for some (but not necessarily all) processes;
- need not be protected by tagged memory;
- can be followed using normal (non-privileged) instructions;
- are stored in a single word;
- fit naturally with compiler needs to create and follow complex data structures.

The proposed mechanism ensures that neither the red-tape nor the pointers of a segment can be directly accessed by the process, i.e. they are fully protected even though they are located in the same virtual address space. This approach, using registers similar to those proposed in this paper, was implemented in the experimental MONADS II [21] and MONADS-PC [23] (CISC) systems[4], which were designed and successfully implemented by the author's research team at Monash University, Australia in the late 1970s and early 1980s.

8 Organising Page Tables

The same technique can (but need not) also be used to store and protect the page table of a virtual address space (i.e. the mapping from virtual page numbers to secondary memory addresses) in the address space itself. (The page table cannot be accessed by unprivileged processes, since it cannot be addressed by them. This is only possible in kernel mode.) Notice that page tables can have different formats, even for the same secondary memory device. Thus if the pages of a large address space are stored in sequence on a secondary device, then the page table can simply consist of a secondary memory start address and length, with the addresses of individual pages being easily calculated as an indexing operation. Both segments and pages were organised in this way in the MONADS systems. This makes a virtual address space totally independent of its environment, which is also important in the Internet.

9 Instruction Execution

The execution of instructions is similar to the execution of instructions in normal RISC systems, except that a dedicated address register can be used to ensure that the program counter remains within the bounds of the current code segment. However, the code address register does not need an access rights field, assuming that code can only be executed. The kernel sets up the code address register when a switch of code segment is required.

Constant segments can be stored in a code virtual address space, addressable via a normal address register, with read-only access set. Immediate values which appear in instructions can of course be accessed directly in the instruction register (IR) as in normal RISC systems.

10 Granularity and Flexibility

The design outlined in this paper does not require restrictive hardware table structures for organising the virtual memory, nor does it define the granularity of segmentation for data or for code. Provided that the length field in an address register is not shorter than the length of a virtual address space then compilers and/or linkers can choose to implement very large segments (as in Multics [27]) or very small segments (as in the Burroughs B6700 [16]) or both (as in the MONADS-PC [23]). A code segment can address either an entire program or it can allow procedures to be individually protected; and data arrays can, but need not, be separately protected.

As outlined, the S-RISC scheme supports fine grain granularity, i.e. using secure pointers. This scheme can easily be supplemented with a kernel supported mechanism for switching domains, as is briefly described in section 14.4.

---

[4] Not to be confused with the Monash Capability System [7].

11 Controlling Access to I/O devices

A good protection scheme involves not only controlling access to segments and higher level software objects, but also to input-out devices. In a system where devices are memory mapped, the kernel could use address registers which provide direct access to the protected memory used for this purpose.

12 Compatibility with Existing RISC and non-RISC Systems

Since the proposed scheme follows the same instruction structure as that of existing RISC systems it should be possible to modify existing compilers in a relative straightforward way to produce machine code which executes correctly in a system based on the proposed model. There appears to be no reason why user application code (written in a high level language) would need to be changed.

13 Hardware Changes

Hardware changes (other than those described above to combine the orthogonal memory model with RISC) are not necessary. For example, the need to protect pointers by the use of tagged memory, which is a requirement in most other attempts to reconcile security with the RISC model, is superfluous. Similarly, individual pointers can be stored in single words, as the example of managing pointers in section 5 shows, thus eliminating the need for fat pointers.

14 Comparison with Other Work

14.1 Segmentation and Paging in Earlier Systems

Almost all earlier conventional systems which provide a hardware-based mechanism to access variable length segments (e.g. the Burroughs B6700 [16], Multics [27, 17, 18, 28, 29], the ICL2900 [30, 19] and the successors of the Intel 8086 [1] pp. B-51ff. have assumed that this should be integrated into the virtual memory mechanism. The result was quite complex systems, primarily because of the resulting relationship between segmentation and paging. The author's IFIP 80 paper [31] showed that the assumption behind this earlier work, briefly summarised in section 4 of this paper, was faulted.

14.2 Protection Schemes in Earlier Systems

The initial need was to protect the operating system from application programs. This was achieved, for example, in the HP2100A [32] by means of a "fence" register, which partitioned the main memory into two parts. Later, with the advent of multiprogramming and the introduction of virtual memory, some systems used base and limit registers to define the area accessible to a program. The GE-645/Multics system [27], followed by the ICL2900 [30, 19] and other systems, introduced a hierarchy of protected layers corresponding to a military-based hierarchical security scheme. Such protection mechanisms, and all earlier protection mechanisms known to the author, are more complicated than the present proposal. In fact, as D.A. Abramson pointed out in his PhD thesis [22], pp.92-3, all such mechanisms can easily be implemented by different kernels which have available a mechanism similar to that proposed in this paper, but none is so simple or flexible.

14.3 Capabilities in Earlier Systems

Although the paper has discussed how it can provide a flexible basis for developing capability systems, the proposed mechanism is *not* a capability mechanism. In fact it has almost none of the features associated with capability systems [33, 7, 34, 35, 36, 23, 37, 38, 39, 4], except that it provides a segment addressing mechanism. For example it does not use unique identifiers, it only addresses segments (not higher level objects), it has no fixed relation to capability lists, no attempt is made to solve the problem of dangling references, it neither creates nor

solves the problems of garbage collection, nor has it direct relevance to capability revocation or object deletion. And finally, it can be used to build non-capability systems.

Some capability systems (e.g. the Chicago Magic Number Machine [37]) support *capability registers*, but these serve as vehicles for supporting a purely segmented virtual memory scheme. The format of a Plessey System 250 capability [33], when it is stored in a capability register, consists of a base address, a limit, and an access rights field. In this sense it is similar to the mechanism outlined above. However the base address is a main memory address and a capability has a quite different format while stored in secondary memory. Although the CAP system [38, 36] used registers to hold capabilities, these were hidden from the user and were integrated into the virtual addressing mechanism.

14.4 The MONADS-PC System

The only capability systems which combined segmentation with paging using the orthogonal model were the MONADS systems [22, 21, 34, 40, 23, 12, 24]. The main difference between these systems and the present proposal is that they were CISC systems, whereas the aim of this paper is to show that the mechanism fits well into a RISC context.

The MONADS website[5] includes a description of the orthogonal paging and segmentation model and describes many other unconventional operating system ideas, including an early description of persistent virtual memory, an implementation of capabilities without the need for a central mapping table [34], distributed shared virtual memory, persistent protected processes, etc. Such features can easily (but need not) be supported in the S-RISC architecture.

In the MONADS-PC system, capabilities are used at two levels. The first level protects segments in a module (typically providing fine grain protection within a program or an information-hiding file) while that module is executing; this uses the addressing mechanism described in this paper. The second level uses "module capabilities" to control accesses between modules, with access rights that reflect the right to invoke specific entry points of an information-hiding module in the persistent virtual memory. Such modules eliminate the need for a conventional file system and thus avoid many risks threatening conventional systems.

14.5 Guarded Pointers and the M-Machine

The M-Machine [41] is a development from MIT which is based on an idea called *guarded pointers*. These are pointers that contain not only a 54-bit pointer to a segment in the virtual memory but also a 4 bit access rights field and a 6 bit segment length. They are protected by tagged pointers. In effect such pointers eliminate the need for segment tables. However, segments must to be a power of two bytes long, and must be aligned on their length, because the length field of the pointer holds the base-2 logarithm of the segment length. Hence segment lengths can range from a single byte to the entire 2^{54} byte address space in power of two increments. Furthermore modifying a guarded pointer (e.g. changing the access rights, or relocating it in the virtual memory) involves scanning the entire virtual address space.

The S-RISC proposal does not require such restrictions: segments can be of any length up to the size of an address space, modifying a pointer does not involve a scan of the entire virtual address space and tagging is not necessary. Furthermore S-RISC also shares the main advantage of bounded pointers: there is no requirement to store pointers in segment tables or capability lists (see section 7).

14.6 Mondrian Memory Protection

The Mondrian mechanism [42, 43] is based on the view, which we share, that it is important to provide protection within programs, not simply between them. However, the Mondrian

[5] www.monads-security.org

way of achieving this is to start with a conventional paging model and to superimpose on this a mechanism which uses a *protection lookaside buffer* (PLB) for managing permissions within a single multi-threaded address space. Each thread is associated with a protection domain and each protection domain within the address space can in effect define "segments" with different access rights, whereby these may overlap and may be defined on a word granularity. The Permissions Table is held in memory and is used to set up and manipulate the PLB.

The basic S-RISC mechanism can in principle provide all these features in a much simpler and more efficient way and with less overhead. Even overlapping segments can be supported, though we see no advantage in this. What it actually supports depends on a particular kernel design which is tailored for a particular purpose.

14.7 Hardbound

Hardbound [44] is motivated by the aim of adding a hardware-based bounds checking mechanism to arrays in C programs. It achieves this by hardware-tagging fat pointers using a mechanism which can compress the fat pointers. The basic S-RISC mechanism can in principle achieve the Hardbound aims in a much simpler and more efficient way and with less overhead, assuming that the C compiler is appropriately modified.

14.8 The CHERI Capability Model

As the title of [45] ("The CHERI Capability Model: Revisiting RISC in an Age of Risk") implies, a specific aim of the CHERI project [46] is to provide a platform which combines the performance of RISC with the protection which capabilities can provide. It uses a tagged memory scheme to protect pointers, and employs capability registers and a coprocessor to manage these.

A CHERI capability consists of a 32 or 64-bit address and a metadata section consisting of permission bits, an object type and bounds information of similar length, plus a protected tag bit. Thus in a 64-bit system a capability is of length 128 bits plus a hidden tag field [46, p. 8]. In contrast the scheme described in this paper requires only 64 bits (for a 64-bit architecture) for pointers and no hidden tags, since bound checking is achieved via the address registers.

The CHERI paper [45] compares CHERI with conventional MMUs, Mondrian memory protection [42, 43], Hardbound [44], Intel's iMPX Memory Protection Extension[6] and the M-Machine [41] in the protection categories "unprivileged use", "fine-grained", "unforgeable", "access control", "pointer safety", "segment scalability", "domain scalability" and "incremental deployment" (see [45], section 2 for definitions and 6 for the comparisons) and concludes that CHERI is the only system/mechanism which is ticked off in every case.

If we add three further categories ("simplicity", "flexibility" and "hardware implementation costs") then I suspect that only S-RISC would succeed in all categories[7], since it needs neither tagging nor a coprocessor to manage capability registers. Nevertheless this compari-

---

6 According to the Wikipedia article *https://en.wikipedia.org/wiki/Intel_MPX* "Intel MPX claimed to enhance security to software by checking pointer references whose normal compile-time intentions are maliciously exploited at runtime due to buffer overflows. In practice, there have been too many flaws discovered in the design for it to be useful, and support has been deprecated or removed from most compilers and operating systems. Intel has listed MPX as removed in 2019..."

7 It might be argued that S-RISC is not unforgeable because it does not use tagging, but just as for tags there must be a privileged level of software which can set the privileges. In the case of S-RISC this can be kernel software, as described in section 7, which must of course be secure.

son is not really fair, since in some cases it compares actual implementations with a model. It must be added that CHERI, a multi-million dollar project at Cambridge University, has many other aspects that are not relevant to this paper.

15 Conclusion

The paper describes a simple segment protection mechanism which accords well with the RISC philosophy. It relies neither on standard virtual memory segmentation schemes nor on conventional paging schemes.

The key features of the S-RISC proposal are its

- independence of memory management issues, including particular page table structures;
- protection at the segment level, which is independent of a particular segment table structure;
- use of registers as the only pre-defined structure;
- suitability for implementation in RISC systems.

The proposed scheme alone does not provide a panacea for the security and privacy problems to which Hennessy and Patterson referred in the quotation at the beginning of this paper. It needs to be complemented by a kernel which provides a foundation for a secure operating system, and then in turn good security strategies and policies built above such an operating system.

Acknowledgements

Special thanks are due to Professors David Abramson and John Rosenberg for their willingness to read several earlier drafts of this paper and for their many suggested improvements.

References

[1] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 5th ed., Elsevier, 2012.

[2] R. S. Fabry, "Capability Based Addressing," *Communications of the ACM,* vol. 17, no. 7, pp. 403-412, 1974.

[3] H. M. Levy, Capability-Based Computer Systems, Bedford, Mass.: Digital Press, 1984, 220 pp.

[4] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson and F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System," *Communications of the ACM,* vol. 17, no. 3, pp. 336-345, 1974.

[5] A. K. Jones, "Capability Architecture Revisited," *ACM Operating Systems Review,* vol. 14, no. 3, pp. 33-35, 1980.

[6] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM,* vol. 9, no. 3, pp. 143-155, 1966.

[7] M. Anderson, R. D. Pose and C. S. Wallace, "A Password Capability System," *The Computer Journal,* vol. 9, no. 1, pp. 1-8, 1986.

[8] G. Heiser, K. Elphinstone, S. Russell and G. Hellestrand, "A Distributed Single Address-Space Operating System Supporting Persistence," Sydney, march 1993.

[9] V. Berstis, "Security and Protection of Data in the IBM System/38," *ACM Computer Architecture News,* vol. 8, no. 3, pp. 245-252, 1980.

[10] M. E. Houdek, F. G. Soltis and R. L. Hoffman, "IBM System Support for Capability Based Addressing," in *Proceedings of the 8th SIGARCH Symposium on Computer Architecture,* 1981.

[11] E. Gehringer and J. Keedy, "Tagged Architecture: How Compelling are its Advantages?" in *12th Annual International Symposium on Computer Architecture*, Boston, Mass., 1985.

[12] J. Rosenberg, J. L. Keedy and D. Abramson, "Addressing Mechanisms for Large Virtual Memories," *The Computer Journal,* vol. 35, no. 4, pp. 369-375, 1992.

[13] J. Rosenberg and D. A. Abramson, "MONADS-PC: A Capability Based Workstation to Support Software Engineering," in *Proceedings of the 18th Hawaii International Conference on Systems Sciences*, 1985.

[14] J. E. Thornton, Design of a Computer: The Control Data 6600, Glenview, Illinois: Scott Foresman & Co., 1970.

[15] G. Radin, "The 801 Minicomputer," in *Proceedings of the First InternationalSymposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, 1982.

[16] E. I. Organick, Computer Systems Organization, the B5700/6700 Series, N.Y.: Academic Press, 1973.

[17] A. Bensoussan, C. T. Clingen and R. C. Daley, "The MULTICS Virtual Memory: Concepts and Design," *Communications of the ACM,* vol. 15, no. 5, pp. 308-318, May 1972.

[18] F. J. Corbato and V. A. Vyssotsky, "Introduction and Overview of the MULTICS System," *Proceedings of the 1965 Fall Joint Computer Conference,* pp. 185-196, 1965.

[19] J. L. Keedy, "An Outline of the ICL2900 Series System Architecture," in *Computer Structures: Principles and Examples (ed. Siewiorek D.P., Bell, C.G. and Newell, A.)*, N.Y., McGraw-Hill, 1982, pp. 251-259.

[20] J. L. Keedy, "Paging and Small Segments: A Memory Management Model," in *Proceedings of the 8th World Computer Congress*, Melbourne, Australia, 1980.

[21] D. Abramson, "Hardware Management of a Large Virtual Memory," *Proceedings of the 4th Australian Computer Conference*, pp. 1-13, 1981.

[22] D. A. Abramson, Computer Hardware to Support Capability Based Addressing in a Large Virtual Memory, Melbourne: Ph.D. thesis, Monash University, Dept. of Computer Science, available at https://espace.library.uq.edu.au/view/UQ:314180, 1982.

[23] J. Rosenberg and D. A. Abramson, "MONADS-PC: A Capability Based Workstation to Support Software Engineering," *Proceedings of the 18th Hawaii International Conference on Systems Sciences,* pp. 515-522, 1985.

[24] J. Rosenberg and J. L. Keedy, "Object Management and Addressing in the MONADS Architecture," *Proceedings of the International Workshop on Persistent Object Systems,* 1987.

[25] J. Rosenberg, "Architectural and Operating System Support for Orthogonal Persistence," *Computing Systems,* vol. 5, pp. 305-335, 1992.

[26] J. Rosenberg, "Architectural support for persistent object systems," in *1991 International Workshop on Object Orientation in Operating systems.*

[27] E. I. Organick, The Multics System: An Examination of its Structure, Cambridge, Mass.: MIT Press, 1972.

[28] R. C. Daley and J. B. Dennis, "Virtual Memory, Processes, and Sharing in MULTICS," *Communications of the ACM,* vol. 11, no. 5, pp. 306-312, 1968.

[29] J. Saltzer, "Protection and the control of information sharing in MULTICS," *Communications of the ACM,* vol. 17, Nr. 7, pp. 388-402, 1974.

[30] J. L. Keedy, "An Outline of the ICL2900 Series System Architecture," *Australian Computer Journal,* vol. 9, no. 2, pp. 53-62, 1977.

[31] J. L. Keedy, "Paging and Small Segments: A Memory Management Model," *Proceedings of the 8th World Computer Congress, Melbourne, Australia,* pp. 337-342, 1980.

[32] Hewlett Packard, "A Pocket Guide for the HP2100 Mini-computer," Hewlett Packard Co., California, USA..

[33] D. M. England, "Architectural Features of System 250," in *Infotech State of the Art Report/Operating Systems vol. 14*, 1972, pp. 395-426.

[34] J. L. Keedy, "An Implementation of Capabilities without a Central Mapping Table," *Proceedings of the 17th Hawaii International Conference on System Sciences,* pp. 180-185, 1984.

[35] H. M. Levy, Capability-Based Computer Systems, Bedford, Mass.: Digital Press, 1984.

[36] R. M. Needham and R. D. H. Walker, "The Cambridge CAP Computer and its Protection System," *Proceedings of the 6th ACM Symposium on Operating System Principles,* 1977.

[37] J. Shepherd, "Principal Design Features of the Multi-Computer (The Chicago Magic Number Computer)," University of Chargo, ICR Quarterly Report 19, November 1068.

[38] M. V. Wilkes and R. M. Needham, The Cambridge CAP Computer and its Operating System, Oxford: North Holland, 1979.

[39] W. A. Wulf, R. Levin and S. P. Harbison, HYDRA/C.mmp: An Experimental Computer System, New York: McGraw-Hill, 1981.

[40] J. L. Keedy and J. Rosenberg, "Support for Objects in the MONADS Architecture," in *Proceedings of the Internationl Workshop on Persistent Object Systems*, Springer Verlag, 1990, pp. 392-405.

[41] N. P. Carter, S. W. Keckler and W. J. Dally, "Hardware Support for Fast Capability-based Addressing," *SIGPLAN Notices, no. 11*, vol. 29, no. 11, pp. 319-327, 1994.

[42] E. Wichel, J. Cates and K. Asanovic, "Mondrian memory protection," *Communications of the ACM,* vol. 37, no. 10, 2002.

[43] E. Witchel, J. Rhee and K. Asanovic, "Mondrix: Memory isolation for Linux using Mondrian memory protection," in *Proceedings of the 20th ACM Symposium on Operating System Principles*, October 2005.

[44] J. Devietti, C. Blundell, M. M. K. Martin and S. Zdancewic, "Hardbound: architectural support for spatial safety of the C programming language," *SIGARCH Comput. Archit. News*, vol. 36, no. 1, pp. 103-114, March 2008.

[45] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton and M. Roe, "The CHERI Capability Model: Revisiting RISC in an Age of Risk," in *Proceedings of the 41st International Symposium on Computer (ISCA 2014)*, June 2014.

[46] R. N. M. Watson, S. W. Moore, P. Sewell and P. G. Neumann, "An Introduction to CHERI (Technical Report UCAM-CL-TR-941)," Department of Computer Science, University of Cambridge, September 2019.