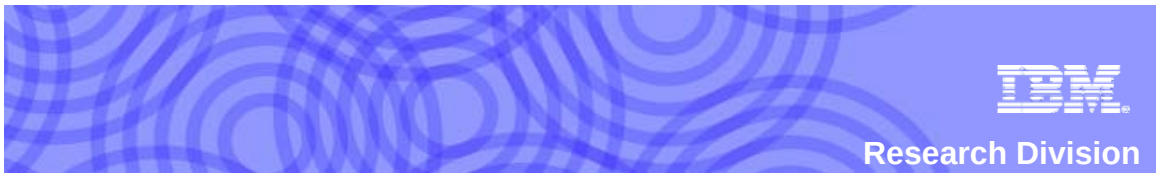
RC25155 (W1105-015) May 4, 2011
Computer Science

IBM Research Report

HWMAC: Hardware-Enforced Fined-Grained Policy-Driven Security

W. Eric Hall, Guerney D. H. Hunt, Paul A. Karger, Mark F. Mergen, David R. Safford, David C. Toll
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich



HWMAC: Hardware-Enforced Fine-Grained Policy-Driven Security

February 21, 2011

W. Eric Hall
Guerney D. H. Hunt
Paul A. Karger
Mark F. Mergen
David R. Safford
David C. Toll

IBM T. J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598
Email: (wehall, gdhh, mergen, toll)@us.ibm.com, safford@watson.ibm.com

Table of Contents

Abstract—A computer processing architecture with fine grain, programmable mandatory access control implemented in the processor hardware is presented. In this processing architecture, all processing contexts and all data are labeled. Depending on the instruction, and labels for the subjects and objects, the hardware enforces a loadable mandatory policy which specifies if the instruction is allowed access to the objects, and if so, also applies rules which may modify the context and output data labels.

1 Introduction

Historically, advances in systems security have accompanied architectural changes. Some of the major early architectural changes that enhanced the security of systems include: The descriptor concept in the Burroughs B5000 [50], [32]; paging in Atlas [27]; The original Multics architecture that featured segmentation and paging [15]; and hardware protection rings [30][37]. Since these innovations were broadly accepted only a few other hardware security architecture proposals have been implemented including: various smart card tamper protection [3] and side channel prevention techniques [1], [14]; TPM [36]; IBM's Secure Processor Architecture [22]; and IBM's Cell processor [44]. (We have excluded various cryptographic accelerators and co-processors as not addressing the core CPU architectures.) None of these has achieved the broad acceptance of the earlier work. Nevertheless, we believe that a radical new approach to building secure systems is required.

Security is one of the most significant issues in computer systems. Security features in systems are designed (architected) to protect the integrity, secrecy, and availability of the system and its data. These protections are typically provided by a combination of hardware methods, such as protected memory [45] in which page tables define the allowed access for a context to a given page of memory, and software techniques in the operating system, such as Lampson's access matrix [28]. Software mandatory access control systems often include mathematical models for secrecy protection, such as Bell and LaPadula [6], and integrity protection, such as Biba [7].

The existing hardware and software security systems have significant limitations. Support for finer grain, finer than process, thread, or page, labeling or access control does not exist in current systems. With typical hardware-based protected memory, while access is controlled to the data when it is in memory, once the data is loaded into registers in the processor, the information that was used to control access is no longer associated with the data, so security is dependent on the operating system software being correct. More specifically, for protection rings, and other similar mechanisms, the ring assures that that software of the appropriate level is running. But once the data is in registers, there is no hardware mechanism to remove (or protect the data) if the level of the software changes. Similarly, all software-based systems are dependent on correctness of their implementation. Typical operating systems have many millions of lines of code; consequently it is very difficult to ensure that all of this code is correct under all conditions.

In this paper we present changes to the computer architecture that can be exploited to increase the security of computer systems. We show how the proposed changes can be used to eliminate some common attacks on computer systems. Our focus in this paper is flexible hardware-enforced fine grained security policies.

2 Background

2.1 Hardware Access Control Background

As mentioned earlier, prior hardware processors have included access control features such as protected memory [45], but these systems only labeled the data in memory, and only at page table boundaries. There have been some prior hardware systems which involved data labeling, in which data words were labeled both in memory and in the processor registers, but these systems did not use the labels with as much flexibility, and loadable mandatory access control system such as contemplated in this paper. For example, Burroughs large systems [32], used three bit data labels, but used them only for type enforcement, not for mandatory access control, and the policy was fixed, not loadable. The Symbolics processors [4] used 4 or 8 bit tags for similar type enforcement.

Hardware capability systems [29],[19] use tag bits on data, but use the tags only to identify capability pointers. Capabilities can provide a form of mandatory access control in which the capability grants access to data. This access control has similar limitations to protected memory; while the capabilities themselves are protected by hardware, the data in general is not. While many capability system have been built and appear in the literature, most do not use hardware tagging, since tagging increases the size of every word in memory by at least one bit. Most capability systems have avoiding tagging, by segregating capabilities from ordinary data in separate memory regions. Even special-built capability machines like the Intel 432 [33] or the Cambridge CAP Machine [52] did not use tagging. Machines that did using tagged memory capabilities included the IBM System/38 [23] and its successors the IBM AS/400 [46] and IBM System i [47] as well as the BiiN system [8] which is the logical successor of the Intel 432, but with tagging.

Motobayashi, et. al. [30] describe the Hitachi Hitac 5020 which had a primitive form of protection rings in hardware, and Schroeder and Saltzer [41] describe the Honeywell Multics computer system with more generalized hardware hierarchical protection rings, which are a form of labeled pointers. Similarly, Schroeder in [40] describes an extension with non-hierarchical protections rings. Data General's *Fountainhead* prototype system [12], [26] was based in part on Schroeder's design. Again this approach does not provide both the labeling and mandatory access control protection of all data, throughout the processor and memory as contemplated by this paper.

In addition, these hardware designs all had fixed security models, such as capability labels, or type labels, or access control tables. They did not permit the definition of new

or different security models other than those explicitly anticipated by the processor architects.

2.2 Mandatory Access Control Background

Lampson's access matrix [28] models all access control policies by specifying the access rights of each subject to each object as an entry in the matrix. Traditionally, the matrix can be viewed either by columns (access control lists [16], [5] or rows (capabilities [18]). Harrison, Ruzzo, and Ullman [21] showed that the general confinement problem (determining that particular information can never leak to a particular bad individual) is undecidable for the general Lampson access matrix.[1]

The need for confinement in the military led to the development of mandatory access controls (MAC) for data secrecy [6] and data integrity [7]. The first general implementation of mandatory access controls was done for the Multics operating system [51]. Most subsequent implementations of mandatory access controls for systems such as Solaris, z/OS, etc. have used approaches similar to the original Multics design. More recently, SELinux implemented mandatory access controls [20], based on a type-enforcement approach.

Partially as a reaction to the complexity of SELinux, Schaufler [38], [39], developed Smack which implements a general Lampson access control matrix in Linux.

Karger and Herbert [25] and Boebert [9] identified problems with implementing mandatory access controls in pure capability systems and proposed restrictions to capability architectures to support MAC. A variety of approaches have been proposed for integrating MAC into capability systems, including HYDRA [54], PSOS [31], Honeywell's Secure Ada Target (SAT) [10], KeyKOS [35], Flex [53], SCAP [24], EROS [43]. It is beyond the scope of this paper to discuss the merits of all these approaches. However, we believe that any of them could be supported by the hardware MAC proposal in this paper.

Our architecture is distinguished from these prior systems both in that it applies these concepts in the processor hardware, rather than software. Our architecture allows for multiple labels per instructions (instructions often have multiple inputs and are often more than one byte long) and for a policy table per processor instruction. The response to a label/policy violation is handled by software. We also include instruction transition rules, which provide for hardware specific functions such as hardware trapping and context manipulation.

Our combination of a programmable hardware-based Lampson Access Matrix and per-instruction access control policy can support a wide range of both existing and new security models. For example, simply by changing the in-memory policy tables, it can enforce capability models by labeling pointers specially and by only allowing far or cross

[1] It is interesting to note that confinement is essentially the objective of the many new products appearing today that claim to provide data loss protection (DLP), although most of those products do not address the theoretical underpinnings of their objectives.

domain references through such typed pointers. It can also directly enforce any of the traditional mandatory access control designs, such as Bell and LaPadula and Biba, as described earlier.

Since the policy may depend upon the instruction, as well as all of the relevant MAC labels, one can create policies which combine the best of prior hardware-based capability systems, in which pointers are labeled, with the best of MAC context level enforcement, in which MAC can be used to control which process contexts are allowed to create pointers. While such combined models have been discussed for software in the past, our design can implement any of them directly in hardware simply by loading the appropriate policy.

3 Next Generation Secure Computer Architecture Project

The Next Generation Secure Computer Architecture Exploratory Research project has been investigating architectural changes that can be used to enhance the security of computer systems. IBM's success with its Secure Processor Architecture (Secure Blue) [22] shows that hardware changes can fundamentally improve the security of computer systems. When this project started our initial goals were to develop and evaluate a computer architecture which supports services orientated architecture, software as a service, while concurrently enhancing the security and isolation of all levels of software running on a system that exploits it. Our initial objectives were to be instruction set neutral (except for new features) and chip interface neutral. We wanted the resulting architecture to be transparent to applications, although we expected it to impact the underlying operating systems and hypervisors. One objective that drove our work was to reduce the size of the reference monitor by increasing hardware support for security. We also believe that these issues cannot be fixed solely by software or solely by hardware.

We began by looking at the collection of hardware functions that can affect the security of a system, including caches, buses, and inter/intra chip communications, considering how malware and malicious data can be used to compromise systems. Since we are concerned about physical as well as software attacks on systems, we also examined better methods for supporting isolation of software systems. This paper focuses on mechanisms that can be used to defeat some known attacks, such as injection. While we initially designed our support with one label per word, prior work [48] showed that by labeling each byte we could defeat injection. We designed our implementation of HWMAC to be flexible enough to allow us to experiment with the tradeoffs and optimizations that will have to be made to realize this idea in a processor. To fully benefit from these innovations, a project to build a high assurance hypervisor that exploits these architectural innovations must be completed–something to be addressed in future work.

Within the first year of the project we realized that it would not always be possible to retain the ideal of being chip interface neutral while concurrently making significant enhancements over existing systems. Consequently we relaxed this requirement, though when possible it remains very desirable. However, as illustrated in this paper, we have been able to achieve the goal of instruction set neutrality (except for new features).

4 MAC in Hardware Design

Originally inspired by Schaufler's Smack system [38], [39], our approach to hardware-enforced fine-grained policy-driven security was to modify the computer processing architecture to include an extended form of Lampson's access control matrix. However, as our design progressed, we realized that much finer-grained protection than Smack provided was essential. We will first describe the general approach and afterwards describe the subset that was implemented for this work. All data is labeled with an eight bit label. In our architecture there is one eight bit label for every byte of memory. [2] Similarly, all processing contexts are labeled. An extended Lampson matrix is stored in memory and provides the access control policy. For each instruction, and for each set of context and input operand labels, the matrix contains access control and transition rules. The access control rules imply the read, write, and execute permissions, along with an extended rule which specifies output operand labels, as well as transition rules for the operation, which may include instructions to trap, or to cause altering the label of the context, such as in support of a low-watermark Biba model. We have reserved one label indicating that a trap is required. An example of a simple Lampson Matrix is shown in Figure 1.

Subject\Object	1	…	N
1	RWX	RX	*
…	RX	RWX	W
N	*	W	RWX

Figure 1 Generic Lampson Access Matrix

To test our architectural approach we took an existing functional simulation of an existing PowerPC processor. The simulated PowerPC processor boots Linux. We then modified the architecture to incorporate our labels and policy tables. Figure 2 presents a high level architecture diagram for the PowerPC processor we modified. In this architecture, there is an Instruction Unit (instruction buffer and dispatch in Figure 2) which receives, queues, and dispatches instructions; maintains registers with Counter and Link addresses; and condition information (CR).

Instructions are dispatched as appropriate to the execution units (Integer, Floating Point, Load/Store, and Branch). Arguments and results of the instructions are stored in the General Purpose Registers (GPR), and Floating Point Registers (FPR).

[2] If a system is build where there is one byte of label for every byte of memory, this doubles the size of memory in existing systems, not counting the memory for policy tables.

The load/store unit and the instruction unit retrieve and store data to main memory. Addresses for the interactions are translated from logical to physical addresses in the Data Memory Management Unit (DATA MMU) and Instruction Memory Management Unit (INST MMU), and the data itself is cached in the respective data and instruction caches.

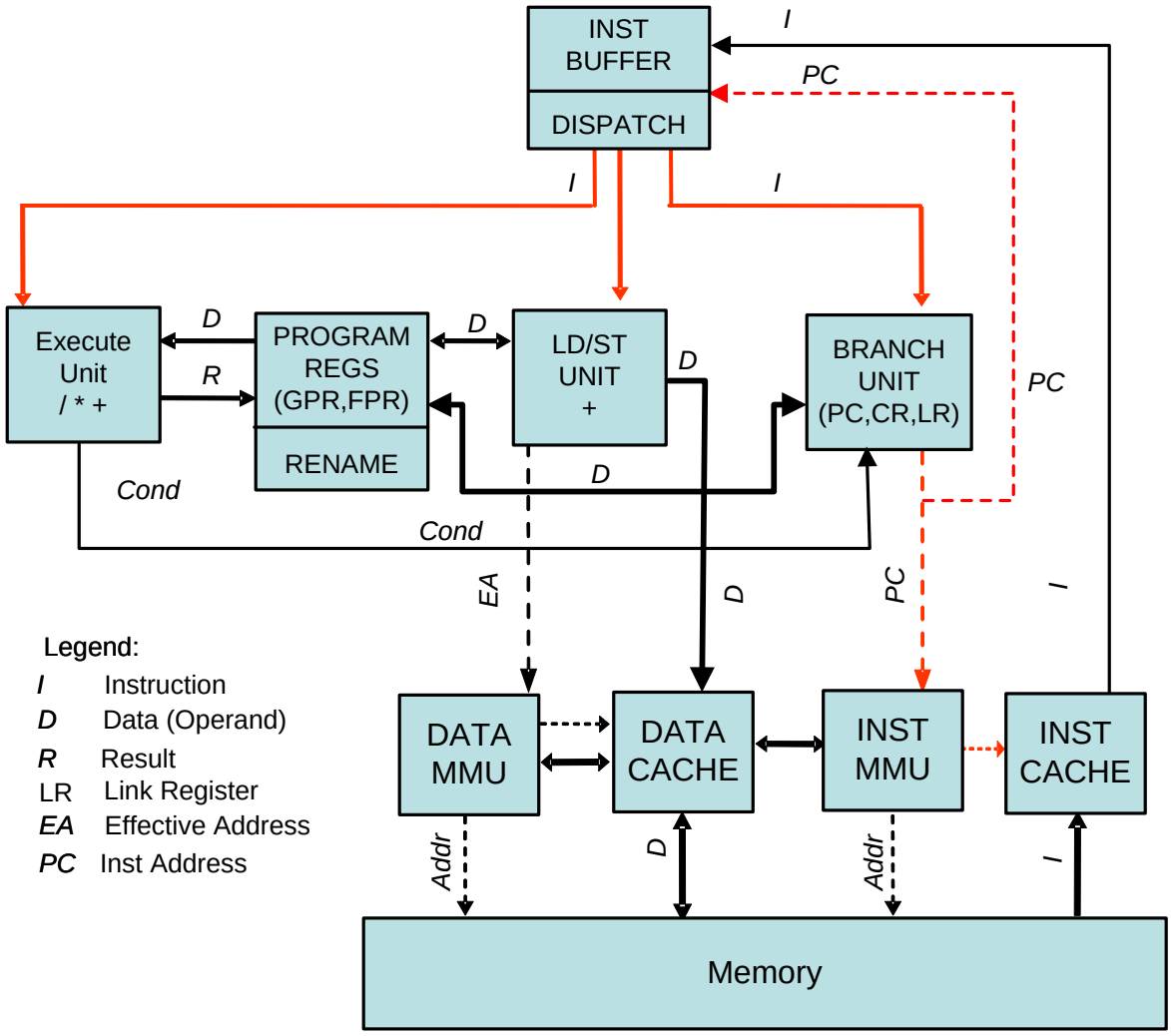


Figure 2 Base PowerPC Architecture Overview

Figure 3 shows how we modified this architecture to achieve our goals. The memory system is modified to provide 8 bit labels for every byte of data read from main memory. These labels are carried with each byte at all times; such as when loaded into cache or processor registers. The processor is modified to include the use of an extended Lampson access control matrix, stored in the Label Policy Cache. In the extended matrix, for each instruction, there is a table in memory which specifies, based on the input label or labels, and the context label, the access rights and transition rules which govern the execution of the instruction. The rules include additional operations, including the ability to trap the instruction, or to modify the context label as a result of the instruction. Policies are enforced by a policy engine that is attached to all major functional units of the

architecture. The policy engine receives the label on the instruction from the instruction decode unit, the label on the current context (PC) from the Instruction Unit, and labels on the operands from the respective data registers. The result of the policy lookup may include a new label for the output result in the register, a new label for the context (PC) in the Instruction Unit, or a signal to the Instruction Unit to trap the existing instruction.

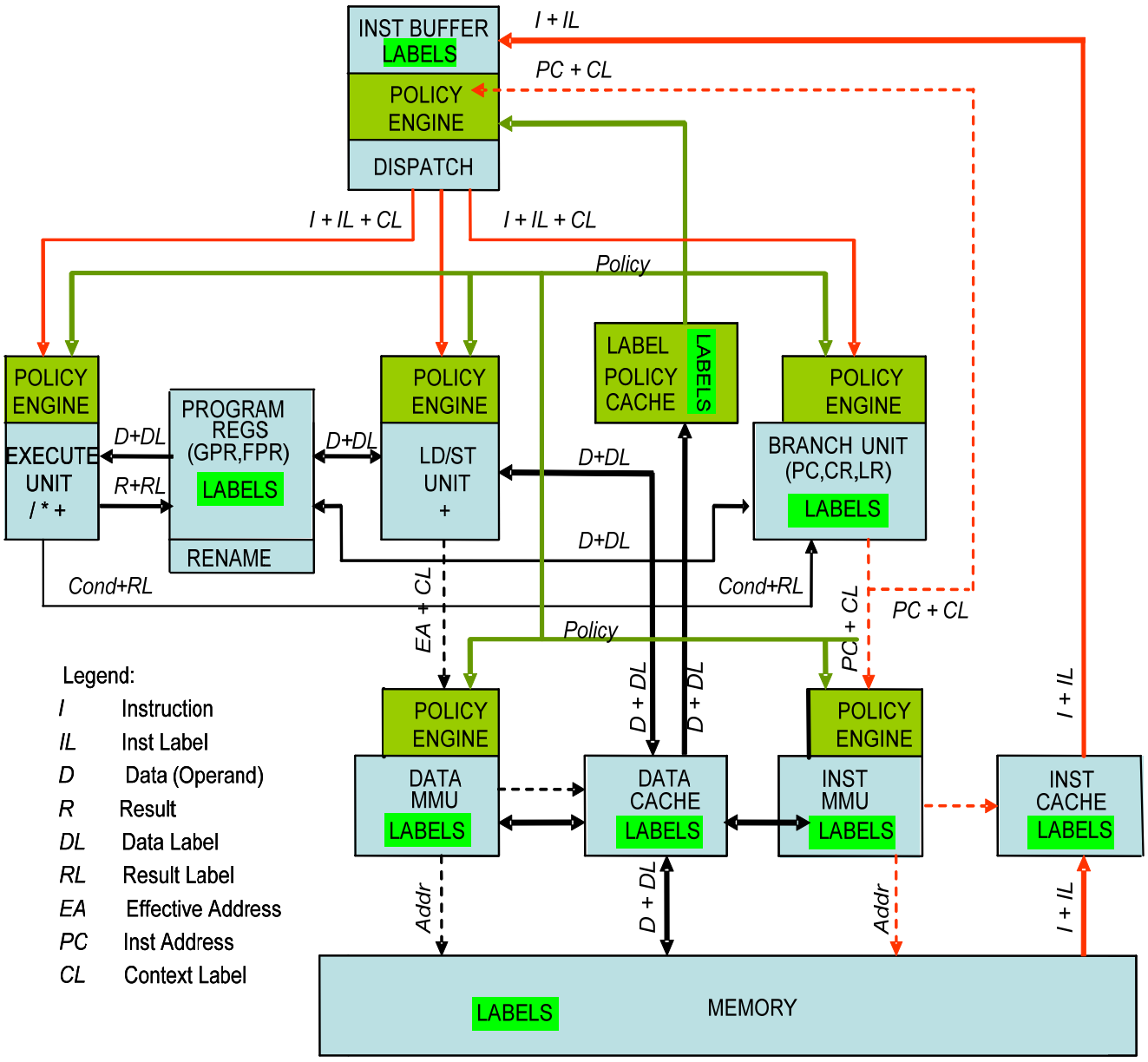


Figure 3 Hardware-enforced fine-grained policy-driven PowerPC architecture.

In addition to the labels on data in memory flowing into both the instruction unit and the registers, the data in the label policy cache and in the instruction and data MMU's are also labeled. Also, the logical addresses sent to the data and instruction MMU's carry the label of the current context, and the MMU's are extended to include allowed context labels in

the respective translation/access tables, so that a context cannot address memory whose page is not properly labeled.

For those readers more familiar with the Intel architecture, consider the simple example shown in Figure 4, of an instruction to load a register with an immediate value. Using the Intel 486 instruction nomenclature this can be represented as: `MOV EAX, 0` (load the 32 bit register EAX with the immediate 32 bit value 0). If our architecture were applied, there would be three sets of labels associated with this instruction: the label on the process context attempting to execute the instruction, the labels on the instruction itself (MOV)[3], and the label(s) on the immediate value. Also, there is the instruction ID (MOV). The policy table will reside in memory, and be accessed based on these labels and instruction ID. The output value will be either the new label to be assigned to the value moved into EAX, or a reserved label indicating a trap should occur.

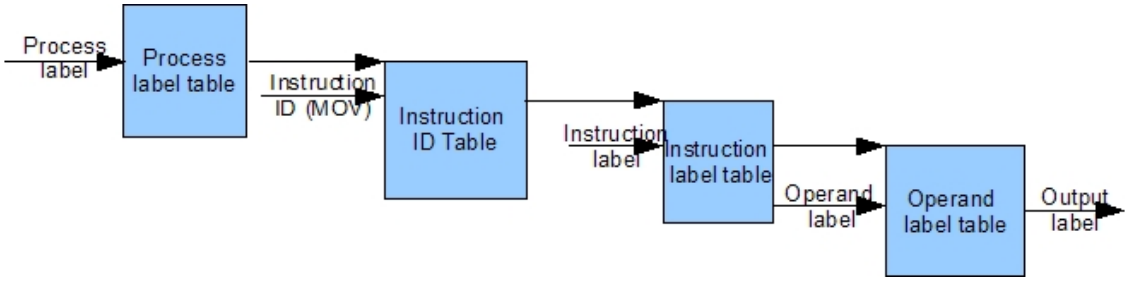


Figure 4 Policy Lookup Data Flow

In any architecture, there may be instructions, such as NOP (no operation), in which only the label of the process, and of the instruction are used, and in architectures with more complex instructions, there may be many operands and each operand may have multiple labels. Returning briefly to the example from the Intel 486 architecture, if this idea were applied, there would be labels for the Segment register, Base register, Index register, Scale value, and Displacement, so the policy table would vary in dimension based on the exact instruction and addressing mode involved.

For our initial PowerPC-based design we did not consider optimizations. However, the processor may cache all or parts of these tables for increased performance. Similarly, in the typical case, all of the labels are likely to be identical, and the implementation may handle this case directly (without policy lookup) for higher speed. Also, the policy could allow calculations on the relevant labels, including minimum, or maximum, to reduce the size of the tables for simple (hierarchical) policies. Additional hardware components could be added for enhanced performance, such as a mechanism for rapidly relabeling pages in memory without having to have the processor do this directly one byte at a time.

We have not yet implemented the full architecture shown in Figure 3in our simulator. We did not implement the policy engines on the data MMU, instruction MMU, or the label

[3] Intel architecture has variable width instructions. In our architecture there will be one byte of label for each byte of the instruction. Normally, we would expect all the labels on an instruction to be the same, but this is not an architectural requirement.

policy cache. We also did not implement the link between the label policy cache and the data cache. Consequently there was no policy-based control for storing data into memory locations. We did not address the issue of how to manage policies from software within the machine. In our initial implementation the policies are immutable after the machine is configured/booted; they appear as if they are in an SRAM or ROM.

5 Example Policies

This section discusses several different types of policies that could be supported by the same hardware MAC features.

5.1 Mandatory Access Control Background

Traditional Mandatory Access Control (MAC) systems enforce a security policy at the operating system level, assigning each process a label and enforcing its access to objects (e.g. files, sockets, devices) according to a policy. If a process is allowed to read an object, the data assumes the context label. In our design, CPU contexts are labeled, and accesses to objects at the memory byte level are controlled by the instruction level policy.

It is simple to write a policy that enforces a traditional MAC policy by defining what labels a given context label are allowed to read. For example, it is easy to create a MAC integrity policy by defining labels and policies which allow read access only to labels of equal or higher integrity, and trapping on attempted reads of lower integrity labels. This integrity policy is demonstrated later, where a high integrity SQL server traps if it tries to read low integrity data, while being allowed to read high integrity data output from a query sanitizer.

5.2 Data Loss Prevention

Data Loss Prevention systems attempt to control outbound disclosure of sensitive data. At the system level, this is equivalent to a secrecy mandatory access control model, and can thus be implemented in our system with a suitable policy as described above. For example, the network device can be labeled with an unclassified label, and sensitive data labeled as confidential, and the policy could prevent any writing of confidential data to the network device.

In addition, simply labeling sensitive data in the system can help detect leaks if output labels are monitored.

5.3 Preventing Code Injection

Some of the most prevalent attacks are injection attacks, in which malicious data or code is injected into a running process through an error or vulnerability in the code. The label architecture is able to detect and block all such attacks that we attempted in the simulated environment. The specific attacks attempted included SQL Injection, Shell Injection, Cross-Site Scripting Injection, and Pointer overflow. (We did not demonstrate code injection buffer overflows as they are well handled with the existing non-executable stack features in most processors.)

5.3.1 SQL Injection

SQL injection is an injection attack on web server programs which allow a user to supply inputs to a web form, where the CGI handler for the form takes the user supplied data, and uses it in a query to a backend SQL server which performs the actual lookup. If the CGI handler code is not well written, it might not properly sanitize the user input, allowing an attacker to inject malicious queries embedded within the allowed user supplied data.

A policy can be written which applies an untrusted label on all network supplied input, with labels flowing with the input data as it moves from the network, through the http daemon, and the CGI frontend. When the sanitizer views the data, the labels then distinguish which bytes in the SQL command originated with the user input, and which originated in the CGI program. Given these labels, a sanitizer can do a perfect job of sanitization of the query, with no modification to the web server, CGI program, or SQL backend. This is presented in more detail in the demonstration section.

5.3.2 Shell Injection

Shell injection is very similar to SQL injection. In this case, the CGI program executes operating system level shell commands based on the input user query, rather than SQL queries. For example, a CGI program might use a shell command to lookup a user's information and privileges as part of processing the request. Just as in the SQL injection case, a label-aware sanitizer can perform perfect sanitization as will be covered in more detail in the demonstration section.

5.3.3 Cross-Site Scripting Injection

Cross-site scripting attacks can occur if a CGI program returns any malicious scripting embedded in a query. Again, it is similar to the prior injection attacks, although it must be sanitized on the output returning to the browser rather than the query going to the backend. Again, a label policy can be written to flow the untrusted labels through the CGI program where the label aware output sanitizer can perform perfect sanitization.

5.3.4 Pointer Overflow

The original buffer overflows injected executable code into a vulnerable process, typically overflowing a stack variable. Modern hardware and operating systems now make the stack non-executable, so such code attacks are no longer effective. Now, rather than injecting code, overflow attacks try to overwrite an address, such as a return address on the stack or a function pointer on the stack or heap. Since an address is not executable, the non-executable stack protections do not block such overwritten pointers.

With our label system, a policy can be written that taints any such user supplied addresses, and refuses to branch or dereference such tainted pointers, thus defeating the attack. This is also covered in the demonstration section.

5.4 Capabilities

Our label system can also be used to enforce capability pointers. Capabilities must be unforgeable, and must be destroyed by assignment. This problem is similar to the pointer injection attack. By writing a policy that defines a capability pointer type, enforces that capabilities must be so labeled to be used, and which controls which trusted labels can create the capabilities, the system can enforce a capability model in hardware.

A capability-based system would need to be rewritten to use the hardware labels, and this is not covered in the detailed demonstration section.

5.5 Type Enforcement

Some programming robustness models seek to enforce strong typing of data, such as preventing integers from being used as pointers. The label system can easily enforce such data semantics in hardware by applying type labels to the data and restricting how the labeled types can be used in instructions. This does require label aware compilers and executable formats, and is not covered in the demonstration section.

One could go further and implement type enforcement in the style of the B5000 processors [32] in which a single ADD (or other arithmetic) instruction could handle any combination of fixed and floating point operands. However, that would require both instruction set and compiler changes.

6 Simulators

Tango is a full function simulator for modeling systems. It is similar to Mambo, IBM's full function simulator for PowerPC-based systems which has been widely used inside IBM for system development. [42], [11], [34] Tango differs in that it is focused on bus-centic functional modeling of heterogeneous combinations of CPU's, buses, networks and devices. It has not focused on cycle-accurate models of CPU's, but does model the register reservation effects of CPU loads and stores, and approximate timing effects of the parallel asynchronous characteristics of caches, buses, networks and devices.

Tango contains libraries of core models for CPU's, caches, buses, memories, network interfaces, UART's, disks and other devices. It also includes libraries of tcl source files for predefined system components like CPU system-on-chips (PPC405GP, PPC440EPx, etc.), memory SIM's and DIM's, and system boards like the AMCC ``Sequoia'' Evaluation Kit [2]. It has a tcl/tk command line interface for defining, loading and driving a target system model during simulation. The Tango software includes a seperate tapserver for routing packets between the network interface models and the host system, allowing host system browsers and file servers to communicate with the simulated system.

Two versions of the PPC440 CPU core were developed: a basic version, which simulates an unmodified PPC440, and a modified version, which simulates the PPC440 with our HWMAC extensions. A Sequoia system simulator was then defined to model an entire Sequoia evaluation kit with a PPC440 processor, 256 MB SDRAM, 64MB flash, ethernet, and PCI slots. The Sequoia board comes with a Linux 2.6 kernel and small Linux operating system in a 75MB ramdisk image. This software was taken from the board, and run unmodified on the Sequoia system simulator to verify the accuracy of the basic simulation.

The Tango core libraries were then modified to add the HWMAC extensions, including a memory model with an 8-bit label for every 8-bit data byte, an in-memory security policy, which specifies for every context label and every instruction what the output action should be (trap or apply label to output), and two new instructions, getmac and setmac, for reading and writing labels on bytes. The modified memory model includes page level label compression, so that if an entire page is labeled with the same value, then only one copy of the label is stored instead of one for each byte in the page. If the page has multiple labels, then a separate label is stored for each byte. This compression is tracked to provide statistics for later analysis, to help predict expected memory overhead if the labels are fully compressed. A “Demo” system simulator was then defined to again model an entire Seqouia system with the HWMAC CPU, memory and buses.

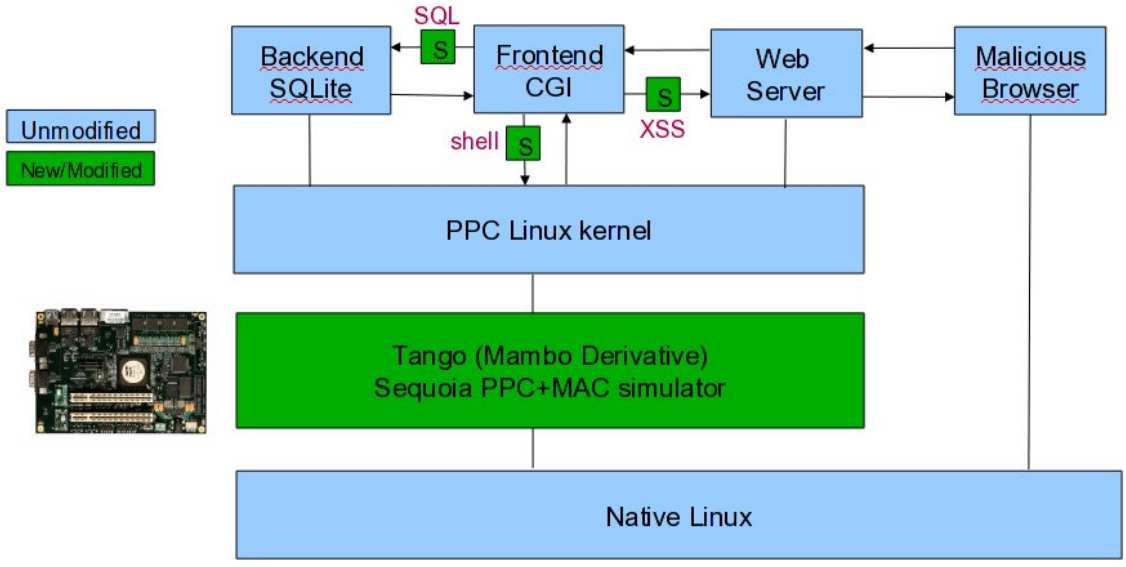


Figure 5 Demo Architecture: the sanitizers are the green boxes with an "S" in the center.

7 Demonstration

Two versions of the Tango simulator were used: a basic simulator, which simulates an unmodified board, and a modified version, which simulates the board with a modified PPC-440 with our HWMAC extensions.

7.1 Demonstration Configuration

The basic Sequoia Linux ramdisk comes with “thttpd”, a small web server. Two html pages with simple query forms were added, each pointing to corresponding CGI frontend programs. One CGI program calls a backend SQL database (SQLite) to perform the actual query, while the second issues a similar shell command to search for the results in a flat text file. Both CGI programs are vulnerable to the corresponding SQL or shell injection attacks. They are also vulnerable to Cross Site Scripting (XSS), and pointer overflow attacks.

The goal of the demonstration is to show how the label extensions can be used to block attacks on all of these vulnerabilities with:

- No modification to thttpd, the cgi programs, or to SQLite
- Transparent insertion of label aware SQL, shell, and XSS sanitizers at the appropriate place between the components
- "Perfect" sanitization (no false positives or negatives)
- A single policy which uses labels simultaneously for:
    - integrity mandatory access control (the high integrity SQL server cannot read untrusted bytes)
    - taint flow (labels on untrusted input bytes from the network flow through)
    - type enforcement/capabilities (untrusted character input cannot be used as a branch address)

Figure 5 shows the layout of the unmodified components, with the added label-aware sanitizers. The sanitizers can tell if they are running on the extended simulator, and perform sanitization only if the labels are available. This way the same demonstration Linux image can be run on both the unmodified and modified simulators to show the successful attacks on the unmodified system and the sanitization of these attacks on the modified simulator.

7.2 Mandatory Access Control

In this demonstration, the CGI program sends the “untrusted” label query bytes directly to the SQL or Shell backends, bypassing any sanitization. The label policy enforces an integrity MAC model, in which the “backend” labeled process is high integrity, and is not allowed to read low integrity “untrusted” labeled data. In this test, the result is that the SQLite backend process traps with a hardware “illegal instruction”, when it tries to read the first untrusted byte. The policy is mandatory because whether the untrusted data was sent deliberately, inadvertently, or from a malicious attack, the CPU hardware enforces the system-wide policy that high integrity cannot read low integrity.

7.3 SQL injection

In this demonstration, the CGI program is expecting the form's input data via the thttpd process, on standard input in the format `"lname=bar"`. The CGI program then reformats the data into an SQL query for the backend in the form `"select * from`

`phone where lname='foo'`". The CGI program is badly written, simply copying the form input into the query template, so if the attacker provides input of the form "`lname=x' or 'x' = 'x`", then the resultant SQL query becomes "`select * from phone where fname='x' or 'x' = 'x'`" which retrieves the entire database, not just one entry.

Simple SQL software sanitizers just escape special characters such as the single quote and semi-colon, but that could cause invalidation of any good queries with these characters (false positives). Perfect sanitization of this query is possible, as described in [48], if we have knowledge of which bytes are trusted (from the CGI program itself) and which are from untrusted user input. Given labels on the characters, we can distinguish good special characters from bad ones and sanitize correctly.

7.4 Shell Injection

In this demonstration, the CGI program is expecting the form's input data via the thttpd process, on standard input in the format "`lname=bar`". The CGI program then simply copied the data "`bar`" into a shell query of the form "`grep bar phone.txt`". If the attacker sends a query of the form "`lname=.</etc/passwd;ls -al;whoami;cat `" then the shell query becomes "`grep .</etc/passwd;ls -al;whoami;cat phone.txt`", which is certainly not what is wanted.

Simple shell sanitizers just escape the special characters less-than, and semi-colon, but again this is imperfect, as it can block valid shell commands. As in the SQL injection, Su and Wassermann [48] show how to do perfect sanitization, given labels on the characters, telling us which bytes came from an untrusted source, and which came from the CGI program itself.

7.5 XSS Injection

If a query contains embedded malicious javascript, and the CGI program somehow echos this javascript back to the browser, then the server enables cross site scripting (XSS). In this example, the CGI program is written to echo back to the browser any query which it does not understand, as a friendly error message. So if the SQL or Shell query is something of the form "`lname=bar\&oops=<script>malicious javascript</script>`" then the CGI program returns the "`oops=...`" part to the user's browser, enabling the attack.

Again, Su and Wassermann [48] show that with labels that distinguish the untrusted "`<script>`" from trusted ones, we can do perfect sanitization on the data being returned from the CGI program to the browser.

7.6 Pointer Overflow

Most processors and operating systems now support non-executable stacks, which block any code injected into a process's stack, due to a stack buffer overflow, from being executed. In response, attackers have come up with attacks which inject addresses instead of code, such as in *Return Oriented Programming* [13], in which a series of carefully selected return addresses are injected into the stack. Control “returns” to these locations, which point to valid instruction-return sections in the process's code space. A sequence of return address can thereby cause controlled malicious execution of existing valid code pieces, by injecting nothing more than a series of return addresses. This attack is not stopped by making the stack non-executable.

In this demonstration, we make the SQL and Shell CGI programs vulnerable to overflow replacement of a function pointer, so that the attacker can overwrite the pointer with an address of his choice, which is similarly not stopped by a non-executable stack. The attack demonstrates vectoring the function call to other code within the program and within the C library.

Blocking this attack is easy with the label support. The policy is simply defined so that pointers with any untrusted bytes cannot be used by a branch instruction. When the pointer injection is attempted, the process receives a hardware "illegal instruction" trap, when it attempts to branch to the tainted address.

7.7 Capabilities

Capabilities are easily implemented with this label support. As shown in the prior test, pointers to capabilities can be required to have a specific capability label, and so that they can only be created by trusted code with a context label allowed to create valid pointers. Forged pointers would then result in traps if they are not properly labeled.

8 Performance and Memory Consumption

We have shown the power and usefulness of HWMAC support in hardware, but the obvious question then becomes how does this support impact performance? There are two parts to the impact: the memory overhead for storing and retrieving the labels, and the time impact for adding the policy calculations into the instruction execution path.

While the HWMAC extended Tango simulator was not designed to be a cycle-accurate emulation of an actual processor implementation, it was extended to collect statistics for simple page-based label compression. The problem with this label compression is that it does not provide for any garbage collection, and over time, most pages eventually become uncompressed with multiple labels. (A reasonably efficient garbage collection would require help from the operating system to tell the simulated hardware when a page was being de-allocated, and the labels could all be reset and compressed.)

Running the demonstration software with Linux, web server, CGI programs, and SQLite backend, with labels for all of the demonstration processes active, initially shows roughly 30% of the pages being uncompressed, rising over time to roughly 90%. Without garbage

collection, we cannot say what the steady state would converge to, other than it is likely to be higher than 30% and lower than 90%.

As for the impact on instruction execution, Tango provides no real data for analysis, as it does not have cycle-accurate simulation of an actual CPU design. There will clearly be tradeoffs between performance and hardware size, as actual hardware is designed, and future work is needed to quantify these tradeoffs in an actual chip design.

9 Related Work

DIFT [49] described a hardware architecture which implements 1-bit tags per data byte, with fixed taint flow logic. Tainted bytes are blocked from being used as pointers or code. The design was validated with simulator based testing. While DIFT does block code and pointer injection attacks, it does not block SQL, shell or cross-site injection, and cannot implement other security models, such as multi-level security, data loss prevention, or capability models.

Raksha [17] presented an enhanced version of DIFT, with 4-bit labels per 32-bit data word. It was implemented in FPGA for testing. While more general than DIFT, its word-level labeling makes precise injection sanitization impossible, and it similarly is not able to support more general security models.

LOKI [55] presented an architecture with 32-bit labels on 32-bit data words, which was implemented in FPGA for testing. It implements label policies on memory accesses, and is used for isolation of a small security reference monitor. It has two limitations: its word-oriented labels do not support precise injection sanitization, and it enforces the labels only on memory access, not on register accesses, as the registers are not extended to flow the labels. Therefore, it cannot guarantee data confinement completely in hardware.

10 Conclusion

In this paper we have proposed HWMAC, a CPU architecture with hardware-enforced, fine-grained, policy-driven security. We have shown how this architecture can enforce diverse security models, and can prevent a wide range of attacks. We simulated the design and demonstrated a combined policy which simultaneously enforced integrity mandatory access control, precise taint flow protection against SQL, Shell, Cross-site and pointer injection attacks, and pointer type enforcement.

This design requires a substantial change to the CPU core, and will thus likely require a larger core size. In this era of chips with 64 or more cores, and little software that can utilize them, perhaps it is time to invest more chip area in security as we increase the number of cores.

References

[1] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi. The EM side-channel(s). In *Cryptographic Hardware and Embedded Systems –CHES 2002*, Lecture Notes

in Computer Science. Vol. 2523, Springer Verlag, pages 29-45, Redwood Shores, CA, 13-15 August 2002.

[2] AMCC PowerPC 440EPx evaluation kit. Applied Micro Circuits Corporation, Sunnyvale, CA, downloades 18 November 2009. http://www.appliedmicro.com/MyAMCC/retreiveDocument/PowerPC/440EPx/PPC440EPx_PB2041_EVK.pdf.

[3] R. Anderson and M. Kuhn. Tamper resistance – a cautionary note. In *Second USENIC Workshop on Electronic Commerce Proceedings*, pages 1-11, Oakland, CA, 18-20 November 1996.

[4] C. Baker, D. Chan, J. Cherry, A. Corry, G. Efland, B. Edwards, M. Matson, H. Minsky, E. Nestler, K. Reti, D. Sarrazin, C. Sommer, D. Tan, and N. Weste. The symbolics ivory processor: A 40 bit tagged architecture lisp microprocessor. In *Proceedings of the IEEE International Conference on Computer Design*, pages 512-514, Rye Brook, NY, 5-8 October 1987.

[5] D. W. Baron, A. G. Fraser, D. G. Hartley, B. Landy, and R. M. Needham. File handling at Cambridge University. In *AFIPS Conference Proceedings, Volume 30, 1967 Spring Joint Computer Conference*, pages 163-167, Washington, D. C., USA, 1967. Thompson Books

[6] D. E. Bell and L. J. LaPadula. Computer Security Model: Univied Exposition and Multics Interpretation. ESD-TR-76-372, The MITRE Corporation, Bedford, MA, USA, HQ Electronic Systems Division, Hanscom AFB, MA, USA, June 1975. http://csrc.nist.gov/publications/history/bell76.pdf.

[7] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, The MITRE Corporation, Bedford, MA, USA, HQ Electronic Systems Division, Hanscom AFB, MA, USA, Apr. 1977

[8] BiiN CPU architecture reference manual. Order Code: 6AM9000-4AB00-0BA2, BiiN Corporation, Hillsboro, OR, 1988. http://bitsavers.org/pdf/biin/Biin_CPU_Architecture_Reference_Man_Jul88.pdf.

[9] W. E. Boebert. On the inability of an unmodified capability machine to enforce the *-property. In *Proceedings of the 7th DoD/NBS Computer Security Conference, pages 291-293*, Gaithersburg, MD, USA, 24-26 September 1984. National Bureau of Standards.

[10] W. E. Boebert, W. D. Young, R. Y. Kain, and S. A. Hansohn. Secure Ada target: Issues, system design, and verification. In *Proceedings of the 1985 Symposium on Security and Privacy*, pages 176-183, Oakland, CA, USA, 22-24 April 1985. IEEE Computer Society

[11] P. Bohrer, M. Elnozahy, A. Gheith, C. Lefurgy, T. Nakra, j. Peterson, R. Rajamony, R. Rockhold, h. Shafi, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang. Mambo—a full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8-12, Mar 2004.

[12] R. G. Bratt, G. F. Clancy, C. J. Mundie, S. I. Schleimer, and S. J. Wallach. *Data Processing System Having a Memory Using Object-Based Information and a Protection Scheme for Determining Access Rights to Such Information*. United States Patent No. 4,525,780, 25 June 1985.

[13] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *Pr0ceedings of the 15th ACM conference on Computer and Communications Security*, pages 27-38, Alexandria, Virginia, USA, 27-31 October 2008. http://cseweb.ucsd.edu/~savage/papers/CCS08GoodInstructions.pdf

[14] S. Chari, C. Jutla, J. R. Rao, and P. Rohatgi. Towards sound countermeasures to counteract power analysis attack. In *Proceedings of Crypto '99*, Lecture Notes in Computer Science, Vol 1666, Springer Verlag, pages 398-412, Santa Barbara, CA, August 1999.

[15] F. J. Corbató and V. A. Vyssotosky. Introduction and overview of the multics system. In *Fall Joint Computer Conference*, volume AFIPS Conference Proceedings Vol. 27, pages 185-196, Las Vegas, NV, 30 November – 1 December 1965. Spartan Books. http://www.multicians.org/fjcc1.html.

[16] R. C. Daley and P. G. Neumann, A general-purpose file system for secondary storage. In *AFIPS Conference Proceedings, Volume 27, Part 1, 1965 Fall Joint Computer Conference*, pages 213-229, Washington, D. C., USA, 1965. Spartan Books.

[17] M. Dalton, H. kannan, and C. Kozyrakis. Raksha: A felxible informatoin flow architecture for software security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 482-493, San Diego, CA, 9-13 June 2007. http://csl.stanford.edu/~christos/publication/2007.raksha.isca.pdf

[18] J. B. Dennis and E. C. Van Horn. Programming semanticsa for multiprogrammed computations. *Commun. ACM*, 9(3):143-155, Mar1966.

[19] E. F. Gehringer. *Capability Architectures and Small Objects*. UMO Research Press, Ann Arbor, MI, USA, 1982.

[20] C. Hanson. SELinux and MLS: Putting the pieces together. In *2005 Security Enhanced Linux Symposium*, Baltimore, MD,28 February – 2 March 2006. http://selinux-sumposium.org/2006/papers/03-SELinux-and-MLS.pdf.

[21] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Comm ACM*, 19(8):461-471, Aug. 1976.

[22] G. D. H. Hunt. Secure processors for secure devices and secure end-to-end infrastructure. IBM Corporation, T. J. Watson Research Center, Yorktown Heights, NY, 30 May 2006. http://www.research.ibm.com/jam/secure-processors5-30-06.pdf.

[23] IBM System/38 technical developments. G580-0237-1, IBM General Systems Division, Atlanta, GA, USA, 1980.

[24] P. A. Karger. Improving security and performance for capability systems. Ph.D. dissertation, Computer Laboratory Technical Report No. 149, Computer Laboratory, University of Cambridge, Cambridge, England, Oct. 1988. http://domino.research.ibm.com/comm/research_people.nsf/pages/karger.pubs.html/\$FILE/trthesis.pdf.

[25] P. A. Karger and A. J. Herbert. An augmented capability architecture to support lattice security and traceabitliy of access. In *Proceedings of the 1984*

Symposium on Security and Privacy, pages 2-12, Oakland, CA, USA, 29 April – 2 May 1984. IEEE Computer Society

[26] TKidder. *The Soul of a New Machine*. Little Brown and Company, Boston, MA, USA, 1981.

[27] T. Kilburn, R. B. Payne, and D. J. Howarth. The Atlas supervisor. In *Computers—Key to Total Systems Control, Proceedings of the Eastern Joint Computer Conference*, volume 20, pages 279-294, New York, NY, USA, 12-14 December 1961. American Federation of Information Processing Societies, Macmillan Company.

[28] B. W. Lampson. Protectoin. *Operating Systems Review*, 8(1):18-24, Jan. 1974. Proceeding of the Fifth Princeton conference on Information Sciences and Systems, Princeton University, Princeton, NJ, USA, March 1971, pp 437-443.

[29] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, MA, USA, 1983. http://www.cs.washington.edu/homes/levy/capabook/.

[30] S. Motobayashi, M. Takashi, and N. Takahashi. The hitac 5020 time-sharing system. In *Proceedings of the 24th ACM Natioinal Conference*, pages 419-419, New York, NY, 26-28 August 1969. ACM.

[31] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Computer Science laboratory Report CSL-116, SRI International, Menlo park, CA, USA, May 7 1980. http://seclab.cs.ucdavis.edu/projects/history/papaers/neum75.pdf.

[32] E. I. Oranick. *Computer System Organization: The B5700/B6700 Series*. Academic Press, New York, NY, USA, 1973. http://bitsavers.org/pdf/burroughts/B5000_5500_5700/Organick_B5700_B6700_1973.pdf.

[33] E. I. Organick. *A Programmer's View of the Intel 432 System*. McGraw-Hill Book Company. New York, NY, USA, 1983.

[34] J. L. Peterson, P. J. Boher, L. Chen, E. N. Elnozahy, A. Gheith, R. H. Jewell, M. D. Kistler, T. R. Maeurer, S. A. Malone, D. B. Murrell, N. Needel, K. Rajamani, M. A. Dinaldi, R. O. Simpson, K. Sudeep, and Z. Lixin. Application of full-system simulation in exploratory systems design and development. *IBM Journal of Research and Devleopment*, 50(2/3):321-332, March/May 2006. http://www.research.ibm.com/journal/rd/502/peterson.pdf.

[35] S. A. Rajunas, N. Hardy, A. C. Bomberger, W. S. Frantz, and C. R. Landau. Security in KeyKOS. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 78-85, Oakland, CA, USA, 7-9 April 1986. IEEE Computer Society.

[36] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurements architecture. In *13th USENIX Security Symposium*, San Diego, CA, 9-13 August 2004. USENIX: The Advanced Computing Systems Association. Ttp://wwww.usenix.org/events/sec04/tech/full_papers/sailer/sailer.pdf.

[37] J. h. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278-1308, Sept. 1975.

[38] C. Schaufler. The simplified mandatory access control kernel. Technical Report, 7 March 2008. http//schaufler-ca.com/data/SmackWhitePaper.pdf.

[39] C. Schaufler. Smack in embedded computing. Technical report, Ottawa, ON, Canada, 23-26 July 2008. http://www.linuxsymposium.org/2008/ols-2008-Proceedings-V2.pdf

[40] M. D. Schroeder. *Cooperation of Mutally Suspicious Subsystems in a Computer Utility*. Ph. D. Thesis, Department of Electrical Engineering, Project MAC TR-104, Massachusetts Institute of Technology, Cambridge, MA, USA, Sept. 1972.

[41] M. D. Schroeer and J. H. Salzer. A hardware architecture for implementing protections rings. *Comm. ACM*, 15(3):157-170, Mar. 1972.

[42] H. Shafi, P. J. Bohrer, J. Phelan, C. A. Rusu, and P. J. L. Design and validation of a performance and power simulator for PowerPC systems. *IBM Journal of Research and Development*, 47(5/6):641--651, September/November 2003. http://www.research.ibm.com/journal/rd/475/shafi.pdf.

[43] J. S. Shapiro and S. Weber. Verifying the eros confinement mechanism. *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 166--176, Oakland, CA, 14--17 May 2000.

[44] K. Shimizu, H. P. Hofstee, and J. S. Liberty. Cell broadband engine processor vault security architecture. *IBM Journal of Research and Development*, 51(5):521--528, Sept. 2007. http://www.research.ibm.com/journal/rd/515/shimizu.pdf.

[45] L. Smith. Architectures for secure computing systems. Technical Report ESD-TR-75-51, The MITRE Corporation, Bedford, MA, HQ Electronic Systems Division, Hanscom AFB, MA, Apr. 1975. http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA009221.

[46] F. G. Soltis. *Inside the AS/400*. Duke Press, Loveland, CO, 1996.

[47] F. G. Soltis. *Fortress Rochester: The Inside Story of the IBM iSeries*. 29th Street Press, Lewisville, TX, 2001.

[48] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. POPL '06: Conference Record of the 33rd ACM SIGPLAN—SIGACT symposium on Principles of Programming Languages, pages 372--382, Charleston, South Carolina, USA, 11--13 January 2006. http://www.cs.ucdavis.edu/~su/publications/popl06.pdf.

[49] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85--96, Boston, MA, USA, 7--13 October 2004. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.111.6059&rep=rep1&type=pdf.

[50] The descriptor - a definition of the b5000 information processing system. BULLETIN 5000-20002-P, Detroit, MI, Feb. 1961. http://www.cs.virginia.edu/brochure/images/manuals/b5000/descrip/descrip.html.

[51] J. Whitmore, A. Bensoussan, P. Green, D. Hunt, A. Kobziar, and J. Stern. Design for Multics security enhancements. ESD--TR--74--176, Honeywell Information Systems, Inc., HQ Electronic Systems Division, Hanscom AFB, MA, Dec. 1973. http://csrc.nist.gov/publications/history/whit74.pdf.

[52] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and Its Operating System*. Elsevier North Holland, Inc., New York, NY, USA, 1979.

[53] S. Wiseman. A secure capability computer system. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 86--94, Oakland, CA, USA, 7--9 April 1986. IEEE Computer Society.

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

[55] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies using tagged memory. In 8th *USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 225--240, San Diego, CA, 8--10 December 2008. http://www.usenix.org/events/osdi08/tech/full_papers/zeldovich/zeldovich.pdf.