# A Security Architecture for Microprocessors

Doctoral Thesis

Jörg Platte

Genehmigte Dissertation zur
Erlangung des akademischen Grades eines Doktors an der
Fakultät für Elektrotechnik und Informationstechnik
Technischen Universität Dortmund

Abteilung Informationstechnik
Institut für Roboterforschung

06.11.2008

## Acknowledgements

It is my pleasure to thank all the people who supported me to make this thesis possible.

Prüfungskommission:

- Prof. Dr.-Ing. Christian Rehtanz (Vorsitzender)
- Prof. Dr.-Ing. Uwe Schwiegelshohn (Referent)
- Prof. Dr.-Ing. Christian Grimm (Korreferent)
- Dr.-Ing. Wolfgang Endemann

## Abstract

The Security Architecture for Microprocessors (*SAM*) is a lightweight and high-performance combined hard- and software security extension for microprocessors. *SAM* has been designed to provide a secure remote code execution environment. It can be used to implement effective copy-protection schemes and provides mechanisms to prevent data and algorithm disclosure. *SAM* provides protection even if an attacker has full access to both the operating system and hardware. *SAM* uses an enhanced processor core which can be used as a drop in replacement for a standard processor to provide transparent encryption and hashing of memory contents to prevent external tampering and sniffing attacks. Further internal security-related extensions support a secure operating system implementation. Both the hardware and software design are presented in this thesis.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

The digital revolution and the omnipresence of computers has changed our life and our sense of justice. With the mp3 compression format and the availability of small players digital music is now a part of everybody's life. The same applies for digital videos and movies as well as for computer games and software. One of the reasons why digital multimedia content is getting more and more successful is their availability. Everything can be downloaded from the Internet, regardless of copyright and distribution restrictions. Most people do not even feel that they are breaking laws when they are copying digital contents. One reason for this is that the ownership and distribution of physical goods can easily be defined, but this is not the case for digital data like programs or digital multimedia content. This can be illustrated by the following example: Digital data can be produced by companies, just like physical goods. When producing physical goods like cars, a lot of effort is required for both development and production. This is typically not the case for digital data. Here, the effort for development is much higher than for production. This is due to the fact that digital data can easily be copied or reproduced once it has been created. A car, for example, has to be physically moved to the customer, and the customer has to buy another car in case he needs another one.

However, this does not apply to digital data. It can be copied without notable effort, and broadband Internet connections have replaced previous distribution channels requiring physical data mediums like compact discs. Due to the success of CD and DVD burners and the growth of hard disk capacities, it is possible to permanently store huge amounts of data. This includes copyrighted work like programs or multimedia data.

The laws have been adjusted to meet the properties of digital data. But this is not sufficient, since people break the laws regularly, if it is easy (like driving faster than allowed) and hard to prove. Therefore, digital data requires other protection schemes than physical goods. While physical goods can easily be protected against theft, this is not possible for digital data, because digital data has to be copied all the time and it is not easy to distinguish between allowed and forbidden copying. Typically, data is copied from the hard disk or an optical medium to main memory and then copied into the processor.

There are copy-protection schemes relying on additional verifications during runtime. For example, a program may request permission from a license server before it starts. But these kind of verifications can easily be broken just by removing the check from the program code. Since this is mostly impossible for inexperienced users, a lot of programs designed to remove these copy-protection checks exist.

Protecting software against unauthorized distribution is one goal of the Security Architecture

for Microprocessors (*SAM*), but it is not the only one. In some cases the program itself has to be protected against external analysis. This is comparable to physical goods, where the production process is kept secret to prevent replicas of the same quality produced by competitors. For programs, all know-how is represented in the programs algorithms. If a competitor is able to analyze these algorithms, he is able to write programs of similar performance or quality. Especially multimedia content is distributed in encrypted form to prevent unauthorized copying. But the program used to display the content has to be able to decrypt the data. If it is now possible to analyze the program, an attacker may be able to analyze the decryption algorithm or to extract the encryption key.

Another scenario where digital data protection is required is GRID computing. In GRID environments, simulation data and programs are spread all over the world to be executed on computing clusters. Typically, the submitter of these programs cannot ensure that the executing computers are well administered and the simulation programs with the computed data cannot be analyzed or modified by third parties. Today, this limitation prevents the commercial usage of GRID computing, due to the probability of spy attacks by competitors.

There are other areas which would benefit from additional software protection, as for example in mobile agent scenarios, where programs (the mobile agents) are transferred and executed on several computer systems to perform tasks in behalf of the agent's owner. These agent-based systems can work reliably only if the agent can perform its tasks without any external tampering attempts. For example, it could be possible to modify the agent's collected results to spoof the agent's owner. There are solutions available for some of these problems [80], but there is no general approach.

Software only protection schemes are limited, because they cannot prevent memory-based attacks like memory dumps performed by the administrator. Even a protected execution environment which prevents execution of unauthorized software cannot prevent memory attacks if hardware supported bus sniffers are used. This kind of attack has been noticed by the broad public when the copy-protection of the XBox [36] has been analyzed by bus sniffing.

Likewise, a hardware-only approach has limitations in a multitasking environment as well, because programs interact with other programs and access data over the network or on the hard disk. Hence, in a multitasking environment any possible attack based on the (modified) operating system has to be prevented as well as hardware-based attacks. With a fully untrusted operating system, all tasks required to support secure multitasking would have to be implemented in hardware. This might be possible but requires more complex and expensive chip designs. Hence, hardware only protection schemes can be used only for small systems like smart cards or microcontrollers. But these approaches are restricted due to limitations in terms of available memory, computing power or network and file system access.

Most new technologies require some time before they are accepted and widely used. Therefore, most new technologies are first developed as an optional add-on to existing technologies. Hence, the acceptance of a security extension for computer systems can be improved if it is implemented as an optional extension, which can be used but is not mandatory to be activated to raise their acceptance. Furthermore, more security should not result in less flexibility from the user's point of view. A secure system which limits the number of programs to be executed or requires only new and adjusted software without compatibility to existing software is not likely to be accepted.

Hence, the Security Architecture for Microprocessors (*SAM*) has been developed as a *drop in* replacement for a standard RISC processor by providing full downward compatibility. Other hardware parts remain unchanged, only the operating system has to be adjusted to take

advantage of the new protection mechanisms for a secure user environment. The processor has been extended to provide strong cryptographic functions used to transparently encrypt, decrypt and hash memory contents to protect all program-related data.
In the following, the term *protected program* refers to a program designed to use the *SAM* protection mechanisms. An *unprotected program* is a normal program which uses none of *SAM's* protection parts.
This work is structured as follows. Part I provides an overview of existing technologies that can be used to protect programs. It starts in chapter 2 with basic protection mechanisms on the processor level. Chapter 3 shows threats for current computer systems and software. A short overview of cryptographic functions is given in chapter 4, followed by security architectures based on cryptographic algorithms in chapter 5.
Part II describes *SAM* in detail. At first the *SAM* design goals are presented in chapter 6. Chapters 7 and 8 describe the architecture and the memory protection mechanisms. The next chapters describe sample implementations of *SAM*. This includes two processor designs in chapter 9, an operating system design in chapter 10, a cache implementation in chapter 11 and a sample application design in chapter 12. Chapter 13 analyzes the security of *SAM*. The simulation environment and simulation results are presented in chapter 14. Finally, chapter 15 concludes this work. The appendices A and B list simulation results and configuration parameters.

# Part I

# Software Protection Schemes

# Chapter 2

# Computer and Operating System Architecture from a Security Point of View

This chapter gives a brief overview of modern microprocessors and operating system kernels. Beside a presentation of the basic functions of a processor and an operating system, this chapter provides an overview of the software protection mechanisms of current operating systems.

## 2.1 Processor Overview

Today's processor architectures can be divided into CISC (Complex Instruction Set Computing) and RISC (Reduced Instruction Set Computing) architectures [33]. CISC architectures have a longer history and their instruction set provides a lot of functionality. The instructions have different sizes based on their information content. Since not all instructions have the same complexity, their execution time is different. Many CISC architectures provide a small number of general-purpose registers but allow memory contents as operands.
In contrast to CISC processors, RISC processors have a simple instruction set. For example, the first versions of the SPARC [85] architecture did not provide multiplication or division instructions. These operations had to be emulated using a couple of more simple instructions. To simplify memory access all instructions have the same size, a mostly similar runtime and instruction format. These properties are then used to increase execution performance by using pipelining. Compared to CISC processors, RISC processors typically have a large number of general-purpose registers, but operands stored in memory cannot directly be addressed by instructions except for dedicated load and store instructions (Load/Store architecture).
Along with the general-purpose registers most processors contain a number of special registers. Sometimes the special registers are accessible by dedicated instructions only or they are for internal use only. Examples for values stored in special register are:

- The Program Counter (PC) or Instruction Pointer (IP) which always points to the currently executed instruction.
- Interrupt-related registers to enable or disable interrupts.

- The current privilege level, because some instructions require special privileges to be executable. Most processors provide at least two levels: supervisor mode and user mode.

Many modern processors cannot be classified as RISC or CISC, because they use a hybrid approach. Intel, for example, translates CISC instructions to one or more so-called "*μOps*" [9]. These *μOps* are less complex and can therefore be processed much faster.

### 2.1.1 Instruction Set Protection Mechanisms

Most processor architectures provide protection mechanisms on the instruction set level by providing at least two modes of operation: user and supervisor mode. User mode defines a normal operation mode for programs without granting additional privileges like direct access to all instructions, memory and processor configuration registers used for memory management or interrupt handling (see below). These operations are limited to supervisor mode, and this mode is dedicated to the Operating System (OS).

### 2.1.2 Virtual Memory

A processor supporting virtual memory distinguishes between addressable and physically available memory. For 32-bit architectures the addressable memory region covers $2^{32}$ bytes, but there may be more or less physical RAM (Random Access Memory) available.
The virtual address space is typically divided in pages with a typical size of at least 4096 bytes, and they can be mapped to physical memory using a page table. Beside the mapping information the page table contains additional data for memory access permission like read, write or execute permissions. All of these functions are provided by the Memory Management Unit (MMU). To speed up mapping of virtual pages to physical pages, a Translation Lookaside Buffer (TLB) can be used. This buffer stores a certain number (for example 16) of the last accessed page mappings to prevent page table accesses when these mappings are needed again. Some architectures provide direct hardware support for several processes, called "contexts", running at the same time. Read or write access can then be granted individually for each context and each page. For architectures supporting only one context a new page table has to be provided manually and the TLB has to be flushed.
Each time a virtual page is not mapped to a physical page or on insufficient access permission an interrupt is raised. This mechanism can be used to further extend the physical memory by for example storing pages on hard disk and loading them to Random Access Memory (RAM) on access.

### 2.1.3 Memory Hierarchy

All available memory in a computer is organized hierarchically, because the speed of today's processors is growing faster than the memory access speed. Sometimes even parts of the main memory are shared by the processor and the graphic card, thus further limiting the memory bandwidth. With SRAM there is a fast RAM technology available, but SRAM (Static Random Access Memory) is more expensive than the commonly used DRAM (Dynamic Random Access Memory) technology. Hence, memory is organized in layers, where each layer closer to the processor has a smaller size but provides a better performance than a farther layer.
Typical modern processors contain at least one cache. A cache provides a small amount of fast memory used to buffer parts of the main memory for faster access. A cache is organized

in small chunks of memory called "blocks" or "lines". Cache lines can be placed in a cache based on the memory address (direct-mapped), freely (fully associative) or in a combination of both (x-way set associative) [33]. Set associative caches have another level of freedom, because they have to choose a cache line to replace within a given set. Typical algorithms for cache line selection are FIFO[1], LRU[2] and random selection. A cache line can be loaded into the cache on each read or write access (write allocate) or on read access only (no write allocate). Data written to the cache can be passed directly to memory (write-through) or on a cache line exchange (write-back).

Each cache stores mainly user data but additionally a small amount of data used for cache management. The memory used to store this data is called "TAG-RAM". It typically contains cache line identifiers, flags to mark lines with valid and modified data and cache line usage information used by the line replacement algorithm.

A processor typically contains a cache hierarchy with a cache for data (L1 data cache) and another cache for instructions (L1 instruction cache) and another and much larger but slower combined data and instruction cache (L2 cache).

### 2.1.4 Interrupts

Interrupts, sometimes called "TRAP" or "fault", can be divided into hardware and software interrupts. Hardware interrupts are raised by hardware, for example when new data is available from the hard disk or to report failures. Software interrupts are raised by dedicated instructions.

The processor stores the base address of the interrupt table (TRAP table) either in a special register or at a given position in memory. The interrupt table typically contains the addresses of the corresponding interrupt handlers or a couple of instructions to be executed on each interrupt.

An interrupt suspends the current program flow to execute the interrupt handler. To be able to continue execution at the interrupted position the processor stores at least the program counter on each interrupt of the interrupted instruction and – based on the architecture – additional information in registers or dedicated memory.

After each interrupt further interrupts are temporarily disabled by the processor. When the state of the interrupted program has been saved successfully, the interrupt handler can reenable interrupts. In cases where priorized interrupts are supported only interrupts with a higher priority than the current one can be raised. Cases where an interrupt handler is interrupted by another interrupt are called "nested interrupts".

With each interrupt, the current privilege level is changed to a higher one. An interrupt handler is left with a dedicated instruction which adjusts the privilege level and resumes the interrupted program.

The same mechanism is used to provide breakpoints to support program analysis. Breakpoints are markers which interrupt program execution if the marked instruction is to be executed. There are hardware and software breakpoints. For hardware breakpoints the instruction address is stored in a processor-specific register. Software breakpoints are realized by replacing the marked instruction by a software interrupt instruction.

[1] First In, First Out.
[2] Least Recently Used.

## 2.2 Operating Systems

The task of an Operating System (OS) is to manage the hardware and software resources of a computer. This section gives a brief introduction based on UNIX-like multiuser and preemptive multitasking operating system kernels. These kernels are implemented using a monolithic design. In a monolithic kernel all core functions like memory and process management and drivers are executed with full access to all hardware and software resources. On the one hand this increases performance, but on the other hand the system stability may decrease, since a faulty driver may accidentally overwrite vital parts of the operating system.

### 2.2.1 Memory Layout

A user process does not have full access to the whole virtual address space. A contiguous part, typically located at higher addresses is used by the kernel only and is shared between all processes. The corresponding pages are not directly accessible by user processes. This kernel address space contains the kernel and memory used to store kernel data like process and memory descriptors and kernel stacks. The remaining virtual address space can be used by programs. Each process has its own virtual address space but it is possible to share memory between two or more processes.
Some of the actual addresses used by a program are not determined by the kernel but by the linker at program creation time. The task of the linker is to take one or more objects generated by compilers and assemble them into a single executable program [48]. The ELF (Executable and Linkable Format [8]) file format is presented as an example of a program file format. An ELF file contains – among others – the following sections:

- Initialization routines (`.init` section).
- The program code (`.text` section).
- Program constants (`.data` section).
- Zero initialized constants (`.bss` section).

The definition of these sections is used only by the linker and debugger. For the kernel all sections are mapped to segments. Segments are contiguous page-aligned regions in an ELF file which can directly be mapped to memory. Each segment provides information about its start address and size.
Programs can be statically or dynamically linked. For statically linked programs all subroutines provided by external libraries are stored with the program code, resulting in a larger program file. This is in contrast to dynamically linked programs. They consist of the program data only, and all used libraries have to be supplied at program start resulting in smaller program files and a smaller memory footprint.
Dynamic program data can be stored at two different places: on the heap or on the stack. The stack is typically located below the kernel address space and grows to lower addresses. The heap is located between the executable and the stack and it grows to higher addresses. Memory on the heap has to be explicitly allocated during runtime by the program using `mmap`, `malloc`, `new` or a similar function. The stack is used to store function parameters, the return address of a subroutine and automatic variables, i.e., variables which are only valid in the current context of a function. Additionally, the stack is used as a temporary storage for

register values, for example when a subroutine is called. The programmer typically does not have direct access to the stack; instead, the stack is managed by the instructions generated by the compiler.
The stack is normally located below the kernel address space and is growing to lower addresses. Environment variables and command line parameters are stored at the beginning of the stack.

### 2.2.2 Interrupts

The operating system handles software and hardware interrupts differently.

- Hardware interrupts are handled as described above. Interrupt handling is getting more complex if the kernel is preemptive, i.e., it can be interrupted as well. Then, the operating system has to distinguish if a user program or the kernel itself has been interrupted.

- Software interrupts are mostly used to implement system calls. The kernel provides different interfaces for process-kernel communication, but they can all be reduced to system calls. Since direct access of the kernel code is not possible, another method to access kernel functions is required and serves two purposes:

  - Privilege expansion by activating supervisor mode. Without supervisor mode direct access to the kernel address space is not possible.
  - A standardized interface used for access control and parameter verification.

  The supervisor mode can be activated by raising an interrupt. Hence, a system call can be invoked by raising a dedicated software interrupt. The parameters passed to the kernel can be stored in registers and on the stack. These parameters may contain references in cases where additional data is located in user memory. A system call basically performs the same actions as an interrupt, but additionally the user-supplied data is verified. Otherwise, for example, it would be possible for a user process to access kernel memory by passing references to kernel memory with a system call.

### 2.2.3 Virtual Memory Handling

The operating system uses virtual memory for different purposes:

- It allows a better memory utilization, because a virtual contiguous memory area can be mapped to arbitrary non-contiguous physical pages.

- The available physical memory can be extended by storing currently unused pages on external memory like hard disks and loading them on access back to RAM. This is called “paging”. The operating system is notified by an interrupt each time a page not located in physical memory is accessed. The same mechanism can be used for programs to load the parts to be executed from the program file on demand.

- The memory access permissions can be used to provide shared read-only memory used for example by shared libraries. Then the same physical pages are mapped to different virtual pages to save physical memory.

- Virtual memory provides Copy On Write (COW) functionality. This is used for example when a program requests a large memory region initialized with zeros. Then the same physical page containing zeros can be mapped to several virtual pages with no write permission. On write access to one of the virtual pages the operating system allocates a physical page, copies the contents of the read-only page to the newly allocated one and maps it to the accesses virtual page with read and write permission.

- With execute permissions parts of the virtual address space can be explicitly reserved for program code. Memory regions containing data and marked non-executable like the stack or the heap can then not be used to execute injected program code using buffer overflows (see section 3.2).

- Physical memory consumption can be lowered by storing libraries used by different dynamically linked programs only one time in physical memory and mapping them into the virtual address space of each process.

## 2.3 Protection Mechanisms

This section discusses protection mechanisms provided by modern UNIX-like[3] operating systems which do not require additional cryptographic units. Other operating systems provide similar protection mechanisms.

### 2.3.1 Multiuser Environment

A multiuser operating system can assign different processes to different users where each user may have different permissions. Users can be assigned to groups, and group membership may give additional permissions. Permissions can be granted on the file system level or directly on the process level. Each file and each process belongs to at least one user, and this user is called the "owner" of the file or process.

### 2.3.2 File System

File system permissions include read, write and execute access to files as well as permissions to create or delete files or directories. A user is not allowed to change access permissions of files if the owner of the file does not explicitly allow this. Therefore, file system permissions can be used to protect sensitive information from unauthorized user access.

### 2.3.3 Process Management

The operating system ensures that each process running on a computer is separated from other processes on the same computer. Each process has its own virtual address space and has no access to memory belonging to other processes. Processes can typically be altered by the owner of a process but not by other users. This greatly improves system stability, because a failure in one process cannot affect other independently running processes on the same computer. System security is improved as well, because sensitive information like passwords stored in memory by a process cannot be read by another process.

[3] UNIX is a multiuser operating system written by the Bell Laboratories.

### 2.3.4 Administrator Access

Most operating systems provide a fine-grained permission design, but there is at least one user which has full access to the system or which can grant itself full access. In the following, this user is called "administrator". Due to its permissions, each administrator of a multiuser system has to be trustworthy, because he is able to bypass all operating system protection mechanisms.

### 2.3.5 Sophisticated Access Control

The NSA (National Security Agency), for example, has developed a security extension for Linux, called "SELinux"[4]. This extension adds mandatory access control mechanisms to Linux and therefore can be used to provide a better separation of information on Linux systems. However, SELinux is only a tool which has to be set up properly to provide enhanced security. Hence, sophisticated usage of access permissions can be an effective protection against unauthorized access as long as the administrator is trustworthy and there are no known weaknesses to bypass the operating system security. Possible weaknesses are described in the following chapter.

[4] `http://www.nsa.gov/selinux/`

# Chapter 3

# Security Issues

This chapter gives a brief overview of attacks to computer systems. Weaknesses in computer systems can be classified in architectural and implementation weaknesses. Architectural weaknesses are the worst case, because they cannot easily be fixed without losing compatibility to existing implementations. For example, the wireless encryption standard WEP (Wired Equivalent Privacy) has several vulnerabilities. The last discovered one allows sending data over a WEP protected link without knowing the secret key [6] or can be used to directly compute the key within 60 seconds [91]. WEP's vulnerabilities cannot be circumvented without loosing compatibility to existing hardware. Fortunately, these kinds of problems are rare compared to implementation errors. Therefore, implementation errors are the main target for adversaries, and some of them will be discussed below.

## 3.1 Hardware Access

All operating system protection mechanisms can be bypassed if an adversary has direct physical hardware access. This kind of attack is not based on a real vulnerability, because most computer systems are intentionally not designed to prevent this kind of attack.

For example, it is possible to bypass file system protection by directly accessing the hard disk by another operating system or by using direct block access. The easiest protection against this attack is a physical protection of the particular hardware. In cases where this is not possible other protection mechanisms using special hardware or cryptography can be used (please refer to chapter 5 for further information). When using cryptography to encrypt all sensitive data, other problems arise. Encrypted data can be accessed only after decrypting it with the corresponding key. Therefore, this key should not be stored on the same medium. Hence, encrypting data to prevent hardware-based attacks is effective, but mostly requires user interaction for decryption.

In cases where sensitive data has to be transferred over busses hardware-based sniffers can be used. This technique has been used to break the security of the XBox [36] by capturing data transferred over a bus using an FPGA (Field Programmable Gate Array) to pre-process the bus signals.

## 3.2 Software-Based Attacks

In cases where direct hardware access is not possible or too complicated software-based attacks are used.
These attacks are mainly used remotely to break into computer systems and to increase privileges in cases where unprivileged access is already available. For this kind of attacks the attacker mainly tries to execute his code with the unprivileged user's permission to access user data or to exploit errors in system software to get administrator access.
In the simplest case the adversary manages to let the user execute a malicious program. In cases where this is not possible, the adversary may try to exploit software implementation errors. Common attacks are:

- *Buffer overflow*: Here the attacker exploits errors where the programmer has forgotten to check if the length of the input data does not exceed the size of the buffer located on the stack which is intended to hold the data. In these cases the adversary tries to replace the return address stored on the stack with an address pointing to the overflowed buffer containing its own injected code. For example, this code can then be used to copy passwords or other sensitive information to an adversary.

  Several techniques exist to prevent buffer overflows [97, 99]. Even when buffer overflows cannot be prevented, the execution of the injected code can be prevented on some processors by marking pages containing the stack or the heap non-executable. Another approach is based on programming languages providing runtime size checks of buffers. Other programing languages like Java [30] or C# [15] do not even allow direct stack access.

- *Heap overflow*: These attacks are based on exceeding buffer sizes, too, but this time the buffers are located on the heap. The heap does not contain return addresses, but in most implementations the actual buffer and additional administrative data like the size of the buffer and pointers to the next and the previous buffer are stored in the same chunk of memory. The attacker can then try to overwrite one of the pointers to point to the return address on the stack. This overwrites the return address the next time the buffer is freed. There are several approaches to prevent this attack, like storing the buffer and administrative data at different positions [98].

  Other approaches are based on stack and heap randomization where the beginning of the stack and heap is selected randomly on program start-up and runtime-allocated memory is selected randomly as well. Both techniques are, for example, part of the Grsecurity Linux kernel extension[1] and have been analyzed in [96]. They can help to defeat some attacks where the knowledge of absolute addresses is required for the attack to succeed.

- *Double free vulnerability*: This attack is similar to heap overflows, but this time the adversary exploits a request to free heap memory which already has been freed, thus overwriting administrative data structures. This attack can for example be prevented by using execution filtering approaches [65].

- *Format string attack*: The function `printf` is used to print data based on a format string. The format string may contain wildcards which are evaluated during runtime to

[1] `http://pax.grsecurity.net/docs/index.html`

read or write to variables located on the stack. If an attacker is able to alter the format string, he may be able to overwrite stack contents to execute his own code. This error can be detected at compile time and the programmer can be informed.

- *Temporary file vulnerability*: Many programs create temporary files to write data. If the program does not choose an unpredictable file name, the attacker may try to place a symbolic link with the expected file name to force the program to write to an existing file the attacker cannot access but the attacked program has write access to. This attack can for example be used to modify the password database to get administrator access.

- *Brute-force attacks*: Most password-based authentication schemes are vulnerable to brute-force attacks. Here the adversary simply tries a huge number of user name and password combinations. Most current authentication schemes try to slow down brute force attacks by putting additional latencies in the authentication process and by disabling an account after a given number of unsuccessful authentication attempts. This countermeasure can be circumvented by an attacker by using the same password but different user names. This attack is applicable in cases of a large user base, because inexperienced users tend to use simple passwords. In cases where the attacker tries common words as passwords the attack is called "dictionary attack".

  Brute force attacks can be used as well to find overflowed buffers or possible format string vulnerabilities by passing random or specially crafted data to programs until the program crashes.

## 3.3 Side Channel Attacks

For this class of attack properties of a computer system or program are analyzed to get additional information. Therefore, this attack is not based on theoretical weaknesses. For example, this kind of attack can be used to distinguish illegal user names and illegal passwords in authentication systems if both cases result in a different time until an error is reported. The Secure Shell (SSH) was vulnerable to this kind of attack (see CVE-2003-0190)[2].
Side channel attacks based on cryptographic algorithms are described in section 4.5.7.

## 3.4 Program Analysis

The attacks described above can be used to get access to computer systems or to extract cryptographic keys. Another kind of attack is used to analyze program code to be able to extract algorithms or to modify the program. This attack can be used to remove restrictions from programs like a copy-protection. In other cases software is analyzed to reverse engineer algorithms and protocols.
Typical tools to analyze programs are debuggers and decompilers. A debugger is used to stop the program at arbitrary places to read or modify program variables and instructions and to analyze the program flow. However, when debugging a program without additional debug information, only the disassembled instructions without variable and function names are available, which makes further analysis difficult. Programs can further be analyzed by

[2]Common Vulnerability Exposure (CVE) ID, `http://cve.mitre.org/`

linking the program to tampered libraries to record internal data or by monitoring all system calls.

A decompiler can be used to generate high-level program code based on the program code. But missing variable and function name information complicates further analysis here as well. Decompilers are more successful when transforming byte code like interpreted by the Java virtual machine [52] back to high-level code. Here most information like variable and function names is still contained in the byte code.

Program analysis can be further complicated when using code obfuscation. A code obfuscator typically renames all variables and functions to meaningless names in case of interpreted languages. In case of directly executed machine code this information is already lost and code obfuscation is here focused on hiding algorithms by adding useless instructions to the code and by reordering or replacing instructions in the program. Barak *et al.* [3] has proven that obfuscation can never transform all programs into a "virtual black box", which hides all internals. Hence, obfuscation can slow down program analysis, but not prevent it.

There are further approaches to prevent program analysis. For example, the Voice over IP program Skype[3] uses many of them, and they have been partly analyzed in [5]. Skype encrypts most parts of its obfuscated program code and decrypts it during program execution to complicate program file analysis. Parts of the program are deleted in memory and Skype randomly computes checksums over parts of the code. They are used to detect software breakpoints and other manipulations to the original machine code. Some running well-known debuggers are detected by Skype, and on detection Skype clears register values and jumps to an arbitrary page in memory to hide the actual stack frame. Therefore, Skype makes it very difficult to analyze the program, but it cannot prevent it.

Another approach for program analysis is virtualization. Virtual machines like the open source QEMU [4] can emulate a whole computer, including processor, memory and peripheral devices. A modified version of QEMU has been used in this work to log memory accesses to a trace file. The same technique can be used to analyze programs running in the virtual machine. Online debugging is further simplified by providing a debugger interface. Therefore, the whole emulated computer system can be stopped and all executed instructions can be analyzed easily, and it is very difficult for an application to detect a virtual machine, since detection mechanisms can be circumvented in many cases by small modifications to the virtual machines' source code.

In case of open-source operating-system kernels like Linux [54] the kernel can be used as well to analyze a running program. The kernel knows all page mappings, can interrupt the program at any place, and has full access to all memory and register contents.

In [60] an approach to protect scalar and array computations is presented. It is based on a compile time transformation of these calculations to prevent reverse engineering. While this approach works for this special class of calculations, it is not applicable for general-purpose programs. However, it can be used in addition to other protection mechanisms to further make reverse engineering harder, but it cannot prevent manipulations.

As a result, an effective program analysis prevention is mostly impossible without dedicated hardware, as described in chapter 5.

[3] `http://www.skype.com`

## 3.5 Copy Protection

Another important part of software protection is copy-protection. Commercial software products are valuable goods and, therefore, unauthorized distribution has to be prevented. There are dozens of different copy-protection schemes. In the following a few of them will be presented.
Copy-protection schemes can be roughly divided into those dealing with hardware and those with software. When using a software-only protection scheme the program can easily be copied to other computers. As soon as the program is executed the program performs checks to verify if further execution is allowed. Common checks are so-called "product key verifications". A product key is typically a unique alphanumerical string which has been generated using a secret algorithm. The program can then read the product key and perform other secret computations on the key to check if it is valid. The verification process can be centralized when using a license server approach as well. In case of a license server the program may ask the server for permission each time it is started. These types of verification can easily be bypassed, either by modifying the program to skip the check or by copying a valid product key with the program.
Since software-only protection schemes can easily be broken, additional hardware support is required to strengthen them. Simple protection schemes were trying to prevent the making of copies of the installation medium. For example, several Compact Disc or DVD protection schemes use partly corrupted data or invalid directory entries to prevent simple copy attempts. However, specialized programs like CloneCD or CloneDVD from Slysoft exist to copy those disks.
Most current copy-protection mechanisms are using a combination of hardware- and software-based protection schemes. State-of-the-art Compact-Disc or DVD copy-protection schemes use physical properties of the medium which are verified during runtime. Physical properties can be read errors or data density. The copy-protection system SecuROM[4] uses these techniques. This protection can be bypassed using virtualization techniques by reading the physical properties of the disk and creating an image containing this information. A virtual drive can then provide access to the image while following the timing constraints.
There are other hardware and software copy-protection schemes available, but they suffer from the same problem, namely that they can be bypassed. Each program has to be copied to memory which always allows further analysis of the used protection schemes and allows modifications to the program to bypass them. It is just a matter of time until a copy-protection scheme is broken.

## 3.6 Sandbox Security

Modern web browsers are able to execute program code supplied by the currently visited website to provide more interactivity. For example, data written into a form can be verified before sending it back to the web server. Each time program code is executed on foreign computers the security of the host system has to be guaranteed. Examples for programs executed within the web server are Java or Javascript [39] programs. For this kind of programs direct file and network access has to be prevented, because otherwise an attacker could simply copy private files possibly containing passwords or other confidential data. A possible solution

[4] `http://www.securom.com/`

provided, for example, by the Java Virtual Machine [52] is the sandbox approach. A sandbox is a limited execution environment which provides full programming capabilities within the environment but no or only limited access to the host computer. Java programs executed by a web browser are restricted in many ways. They are, for example, only allowed to write temporary files in a given directory and to start network connections with the web server they have been downloaded from. When implemented properly a sandbox can prevent many attacks requiring direct access to the computer.
Unfortunately, a sandbox approach cannot prevent all attacks. At first, the implementation of the sandbox may contain exploitable programming errors or logical errors, which allow the program to escape from the sandbox[5]. Second, a sandbox does not necessarily prevent denial of service attacks like consuming host resources like processing time or memory.
Hence, the sandbox approach provides more protection for the host system than a direct execution of unknown program code, but an adversary may still find ways to break out of the sandbox.
As a result, even a well-administrated computer system can be successfully attacked by a sophisticated attacker. The attacker may get full access to all files and may be able to analyze or alter running programs.

[5]Common Vulnerabilities and Exposures (CVE): CVE-2005-3904, CVE-2005-3905, CVE-2005-3906, CVE-2005-3907.

# Chapter 4

# Cryptography

The last chapter has shown that most security threats regarding computer software cannot be prevented using standard hardware. The usage of cryptography offers additional potential for software protection. Therefore, this chapter gives a brief introduction to cryptographic algorithms and cryptographic attacks and is based on [81] and [56].

## 4.1 Symmetric Cryptography

Cryptographic algorithms can be divided into symmetric and asymmetric ones. Symmetric algorithms are based on a shared secret, the secret key $k$, which is similar or the same for encryption and decryption. They can be further divided into stream ciphers and block ciphers. A stream cipher is encrypting bit-wise, whereas a block cipher encrypts a couple of bits (a block), typically 64 or 128-bits, at the same time.
The following formal notation is used:

- The plaintext is denoted by $P$.
- The encryption algorithm encrypting with key $k$ is denoted by $E_k$.
- The decryption algorithm decrypting with $k$ is denoted by $D_k$.
- The resulting ciphertext is denoted by $C$.

In cases where the cryptographic operation is applied to a block the block number is denoted as $i$:

$$C = E_k(P); \quad C_i = E_k(P_i)$$
$$P = D_k(C); \quad P_i = D_k(C_i)$$

In cases where the plaintext is smaller than a block a pad of random data has to be appended to the plaintext before encryption.

### 4.1.1 Random Number Generation

Random numbers play an important role in cryptography. For example, symmetric keys should be chosen randomly to prevent attacks. According to [81], the following types of random number generators exist:

1. Pseudo random number generators are often used in computers. Computers are deterministic, and therefore the generated sequence of "random" numbers is periodical. A good pseudo random number generator meets a lot of criteria. For example, the generated sequence should not be compressible.

2. Cryptographic random number generators are periodical, too, but the generated random numbers should be unpredictable as well, even when the algorithm and all previously generated bits are known.

3. True random number generators produce random numbers which are not reproducible. Even when using the generator with exactly the same input data the generated numbers are different.

True random number generators require dedicated hardware using random physical processes like resistor noise to generate random numbers.

### 4.1.2 Algorithms

Many symmetric algorithms have been invented with different goals in mind. Some algorithms are optimized for a software implementation, others are faster in hardware, but all of them claim to be secure. However, there is only one provable secure algorithm, the One Time Pad (OTP) algorithm. Here, the key has the length of the plaintext, and each plaintext bit is encrypted with the corresponding bit of the randomly chosen key using the Exclusive Or (XOR) operation. Therefore, each key can be used only once and has to be generated using a true random number generator. Despite its security, the algorithm has a lot of drawbacks. In particular, the algorithm cannot be used for high throughput communication channels because of the resulting size of the key, and the key distribution process itself requires a secure channel.
Since all other algorithms can be broken at least theoretically due to the limited key size, it is advisable to choose a well-known and analyzed algorithm. In the past this was for example the block cipher DES (Data Encryption Standard) and later 3DES [62]. However, DES should not be used any longer due to its key length of 56 bits, which is too small for current security needs.
In 2000, the National Institute of Standards and Technology presented its successor, the Advanced Encryption Standard (AES, [63]). AES is based on the Rijndael [17] algorithm but with 128-bits it supports only one block size. The key can have a length of 128, 192 or 256 bits. The algorithm uses several rounds of key additions, substitutions, and shift- and mix-operations. The number of rounds depends on the key length and lies between 10 and 14.

### 4.1.3 Modes of Operation

Cryptographic block algorithms can be used in several ways. In the following, some modes are discussed. Among others, the following modes are used widely:

- Electronic Codebook Mode (ECB) or direct encryption: Here a block of plaintext is directly encrypted to a block of ciphertext:

$$C_i = E_k(P_i)$$
$$P_i = D_k(C_i)$$

A drawback of this mode is that the same plaintext block always results in the same ciphertext block when the key is unchanged. Nevertheless, this mode is suitable for random access configurations like database accesses because it allows independent encryption and decryption of arbitrary blocks.

- Cipher Block Chaining Mode (CBC): This mode uses a feedback mechanism. The previous ciphered block is XORed with the current plaintext block before encryption:

$$C_i = E_k(P_i \oplus C_{i-1})$$
$$P_i = (C_{i-1} \oplus D_k(C_i))$$

The first block is XORed using a random pattern called "initialization vector" (IV). This mode encrypts the same plaintext blocks differently, but arbitrary decryption of random ciphered blocks is not possible.

- Output Feedback Mode (OFB): Here the result of the previous encryption step is encrypted again for the next step using a block cipher. The result of each step is XORed with each corresponding block of plaintext:

$$C_i = P_i \oplus S_i; \quad S_i = E_k(S_{i-1})$$
$$P_i = C_i \oplus S_i; \quad S_i = E_k(S_{i-1})$$

In the first round an initialization vector is encrypted. Note that encryption is involutive, so a double application recovers the plain text again. Hence, for encryption and decryption this mode uses only the encryption engine of the block cipher. This step can be calculated in advance without knowing the plaintext.

- Counter Mode (CTR): This mode is similar to OFB, but here a unique value like a counter $\mathcal{C}$ is encrypted for each block of plaintext and the result is XORed with the plaintext:

$$C_i = P_i \oplus E_k(\mathcal{C}_i)$$
$$P_i = C_i \oplus E_k(\mathcal{C}_i)$$

The counter can be public, but it has to be unique, otherwise an attacker might be able to decrypt the ciphertext easily (see section 4.5.5). An application of AES in Counter Mode is described in [20].

## 4.2 Data Integrity

Cryptographic algorithms can be used to hide sensitive information, but they are not designed to prevent manipulations in the first place. Therefore, modifications to encrypted data may remain undetected if no further verification checks are used.
In contrast to cryptography, where the size of the plaintext equals the size of the ciphertext for many algorithms, data integrity can be ensured only by additional redundancy. There are many algorithms used to detect transmission errors in communications engineering like checksums or Cyclic Redundancy Checks (CRC), but they are mainly designed to detect random transmission errors. For cryptographic systems, data integrity has to be ensured

even if a sophisticated adversary is able to modify a protected message. Therefore, better algorithms, called "one-way hash functions", are required. In formal notation a hash function $H$ transforms a message $M$ into a hash $h$ with a fixed size of $m$ bits:

$$h = H(M)$$

One-way hash functions should have the following properties [81]:

- For a given $M$ it is easy to compute $h$.
- For a given $h$ it is hard to compute $M$, so that $H(M) = h$.
- For a given $M$ it is hard to find another message, $M'$, so that $H(M) = H(M')$.
- It is hard to find two random messages, $M$ and $M'$, so that $H(M) = H(M')$.

A hash function with these properties should generate a hash with a reasonable size $m$. Otherwise, brute-force attacks exploiting the birthday paradox might be able to find two colliding messages $M$ and $M'$ with the same hash value $h$ after $2^{\frac{m}{2}}$ tries with a probability of 0.5.
Current popular hash functions are SHA-1 [61] ($m = 160$) and MD5 [76] ($m = 128$), but weaknesses have been found for both algorithms [94, 95]. They permit to find collisions much faster than using a brute-force attack.
Due to the weaknesses the usage of SHA-256, SHA-384, or SHA-512 is recommended. The resulting hash length $m$ equals the number in the algorithms' name.
Since encryption provides the properties listed above, standard block ciphers can be used to compute hashes as well, but due to their lower performance, dedicated hash functions are more common. There are several methods to turn a block cipher into a hash function. The AES hash [7], for example, applies the Davies-Meyer [57] method to calculate a hash of $M$. At first, $M$ is split into the following $i$ blocks of equal size: $x_1 \dots x_i$. Then, the hash is calculated as follows:

$$H_i = E_{x_i}(H_{i-1}) \oplus H_{i-1}$$

Related to hashes are Message Authentication Codes (MAC) [45]. They combine a cryptographic hash function and a secret key to build a hash that can only be verified by the owners of the secret key. A simple MAC computes a cryptographic hash which is then encrypted using a block cipher with the secret key.

## 4.3 Asymmetric Cryptography

Asymmetric cryptography does not require a shared secret. Instead, the secret is split into a private and a public part. The private part has to be kept secret. The public part $k_{pub}$ is used for encryption

$$C = E_{k_{pub}}(M)$$

while the private part $k_{priv}$ is used for decryption

$$M = E_{k_{priv}}(C).$$

Asymmetric algorithms are much slower than symmetric ones. Therefore, many cryptosystems use a hybrid approach: all data is encrypted using a fast symmetric algorithm with a random secret key, and this symmetric key is encrypted asymmetrically. This approach is much faster, because the size of the encrypted data is typically much larger than the size of the key.
The most popular asymmetric cryptographic algorithm is RSA [77]. For RSA the public and private keys can be used both for encryption and for decryption, which permits the usage of RSA for digital signatures.
Digital signatures are used to prove the origin of digital data, and they are required for distributing public keys via insecure channels. To prevent attacks, where an adversary has exchanged a public key with its own key a digital signature can be used to protect the original public key. Signatures typically require a trusted third party which is used to prove that a public key belongs to a specific person. A document can be signed using RSA by encrypting it, or a one-way hash of the document, with the private key. The signature can then be verified by decrypting the document or the one-way hash with the public counterpart.

## 4.4 Key Exchange Protocols

A key exchange protocol is required each time a random encryption key is required. This can be the case when two parties want to start an encrypted communication without a common secret key. A widely used protocol is the Diffie-Hellman (DH) key exchange protocol [19]. The protocol permits two parties to compute a common secret key while transferring all related data via an insecure channel.

## 4.5 Cryptographic Attacks

Parallel to the development of new cryptographic algorithms, methods to break them have been invented. Due to the large number of attacks and due to algorithm-specific attacks, this section will name only a few common attacks. An adversary may try to attack the algorithm directly or try to attack related parts like a protocol using the algorithm or use a side-channel attack to get further information. Some cryptographic algorithms are kept secret to make attacks more difficult. This does not increase security in many cases, because secret algorithms have not been analyzed by many cryptographic experts and may contain weaknesses. They are therefore not considered in this section.

### 4.5.1 Brute Force

As stated above, provided it be used carefully the only secure cryptographic algorithm is the one-time pad, because here each possible message $M$ with the length of the ciphertext $C$ has the same probability. All other algorithms can be broken due to the limited key size, if the encrypted message is reasonably long. The simplest attack is to test all possible keys. Therefore, using a long and true random key is prerequisite for defeating this brute force attack. Then, the matching key with a length of $n$ bits can be found with a probability of 0.5 after approximately $2^{n-1}$ attempts which can be considered impossible with $n = 128$ bit keys in a reasonable time frame and a massive parallel approach. Hence, cryptographic algorithms which can be broken faster than using a brute force approach are considered insecure and

should not be used. distributed.net[1], for example, uses the computing power of personal computers spread all over the world to decode RC5 ciphertext using a brute force approach. The project gives an overview of the time required for brute force attacks. A 64 bit RC5 key, for example, was found after 1757 days.

### 4.5.2 Known Plaintext

Knowing the plaintext $M$ and $C$ may help an attacker who is trying to get the key $k$. Without any information about the plaintext it may be impossible to find a matching $k$ if $M$ contains random binary data only. In other cases the adversary may try to find the plaintext by using statistical assumptions about $M$ to find the key.

### 4.5.3 Related-Key Attack

Here the attacker knows that at least parts of $k$ are the same without knowing the exact values of these bits. For example, this kind of attack has been used to break WEP-protected wireless networks [24]. This attack can be prevented by using true random keys.

### 4.5.4 Replay Attacks

A replay attack is used to alter ciphertext unnoticed without being able to decrypt it. As described above, ciphertext can be protected against modifications using one-way hashes. However, if an attacker manages to replace the ciphertext and the corresponding hash, he might be able to replace a new message with an older one. This attack can be prevented by adding additional redundancy like timestamps or counters to the message.

### 4.5.5 XOR Security Considerations

OFB mode and CTR mode both encrypt a block of precomputed data and XORing the result with the plaintext. This decouples the slow encryption step from the plaintext and allows encryption of single bits using a block cipher without padding. The security of these modes is as high as in ECB mode as long as the encrypted values, here denoted as $\mathcal{C}$, are always unique for a given $k$.
Assume the same $\mathcal{C}$ is used as a counter for two plaintext blocks $P_1$ and $P_2$:

$$C_1 = P_1 \oplus E_k(\mathcal{C})$$
$$C_2 = P_2 \oplus E_k(\mathcal{C})$$

If $\mathcal{C}$ is public, the attacker knows that both plaintext blocks are encrypted with the same counter. He is then able to eliminate $E_k(\mathcal{C})$ by XORing $C_1$ and $C_2$:

$$C_1 \oplus C_2 = P_1 \oplus E_k(\mathcal{C}) \oplus P_2 \oplus E_k(\mathcal{C}) = P_1 \oplus P_2$$

With the XOR combination of two plaintext blocks the original plaintext can be recovered in many cases. In the simplest case one block contains only zeros, which directly reveals the other block contents. In all other cases statistical analysis can be used to reveal the plaintext [25].

[1] `http://www.distributed.net`

When using XOR to decrypt plaintext, an attacker is able to flip bits in the ciphertext, which results in the same bits flipped in the decrypted plaintext. This attack can be used in cases where the ciphertext is not protected against manipulations.

### 4.5.6 Man-in-the-Middle

For this attack an adversary tries to intercept a communication between two parties by pretending to one party to be the other party and vice versa. The only possibility to defeat this attack is a proper authentication scheme in conjunction with a good protection of an established connection. For example, the public RSA key has to be signed to prevent this attack if it is distributed over an insecure channel. The same applies for the messages required for the DH key exchange.

### 4.5.7 Other Attacks

Cryptographic algorithms are often implemented in software and therefore the same attacks as described in chapter 3 can be used to break them.
Side channel attacks are very common in case of cryptographic algorithms, because for example the runtime of a cryptographic operation may reveal information about the key or the decrypted data [43]. Even the processor cache can be used to reveal cryptographic keys used by another running process, as described in CVE-2005-0109 and [68]. Here the attacker exploits that both processors use a common L1 cache in Intel's Pentium 4 processor with hyper-threading enabled. This attack can be prevented by disabling hyper-threading. Other attacks based on the power consumption [42] or on electromagnetic radiation [46] are possible.

## 4.6 Standard Protocols

Cryptographic algorithms and protocols are subject to several attacks, and their implementation may contain errors. In case of cryptographic protocols, wrong usage of algorithms may result in information leakage or may allow further attacks. Therefore, the usage of standard protocols and algorithms and standard implementations is advisable. These implementations may contain errors as well, but the probability of errors in a new and not widely analyzed implementation is much higher.
For network connections, the Transport Layer Security (TLS) protocol [18], the successor of the Secure Socket Layer (SSL) protocol, can be used. It provides symmetric encryption for data transfer, asymmetric cryptography for client and server authentication and key exchange methods. Furthermore, it is standardized for a wide range of TCP-based Internet protocols like HTTP (Hyper Text Transfer Protocol) or IMAP (Internet Message Access Protocol).

# Chapter 5

# Security Architectures

This chapter presents sophisticated hardware solutions designed to further prevent program analysis and program manipulations as described in chapter 3.

## 5.1 Memory Protection Schemes

Many software protection schemes can be broken by direct manipulation or reading of memory contents. Therefore, a memory protection is the first step for additional security. Existing solutions cover the following three aspects:

1. Memory integrity protection using hash values.

2. Memory disclosure prevention using memory encryption.

3. Address bus protection to prevent further program-flow analysis.

### 5.1.1 Hash Trees

Using hash values to guarantee memory integrity has been proposed by Gassend *et al.* in [27]. The protected memory area is divided into small chunks of memory with the size of a cache line, and each line is protected by a hash value. To further speed up memory verification, memory areas containing hash values only can be cached as well (cache lines storing hash values will be denoted as "hash lines" in the following). Therefore, the size of the hash value has to be a fraction of the size of a cache line for efficiency reasons. Using a single hash value to protect a memory region is not sufficient, because it does not prevent replay attacks. Static program data cannot be replayed, but an attacker could try to replace parts of the stack or the heap with out-of-date data to alter a program. Hence, Gassend proposes the usage of hash trees (sometimes called "Merkle trees") as described in [58] to prevent replay attacks by protecting hash values with additional hash values. For this scheme only the top-level hash called "root hash" needs additional protection, since replay attacks would still be possible if this hash value could be replaced. A possible protection mechanism is a dedicated on chip memory which cannot be tampered. A hardware implementation of this approach has been proposed in [88].

### 5.1.2 Memory Encryption

Hash values prevent any kind of program and data manipulations, but memory contents can still be read by an adversary. Therefore, transparent memory encryption techniques can be used to hide memory contents. Rogers *et al.* [78] uses a technique called "memory predecryption" to protect memory contents. Memory contents are split into chunks with the size of a cache line and these chunks are encrypted directly. Rogers tries to hide the additional encryption latency by using a prefetching approach. The future memory access is predicted using standard prefetching techniques. The performance of this approach depends on the quality of the prefetcher and on the executed programs, because not all memory areas patterns can be predicted well by a prefetcher.
Direct encryption of memory contents can be too slow for high bandwidth applications. Therefore, memory scrambling techniques can be used [75]. For example, memory scrambling can be implemented by using a simple and fast scrambling function like a random number generator and XORing the random number with the data to protect. This method is similar to counter mode, but much faster due to the lack of a complex cryptographic algorithm.
Further memory protection schemes are described in section 5.2, because they are parts of a complex security architecture.

### 5.1.3 Hiding Address Information

Encrypted memory hides memory contents, but it does not hide memory addresses. Memory addresses may give an attacker a lot of valuable information in cases in which the attacker is able to identify the encrypted algorithm. Zhuang *et al.* [100] describes a statistical approach to identify the execution of standard cryptographic libraries based on the memory access pattern. Once identified, the memory access pattern can be used to reveal cryptographic keys, because many algorithms are implemented using several conditional branches which are used to distinguish only two values. In these cases an attacker can directly disclose the corresponding value which might be a part of a key. A solution called "HIDE" is presented by Zhuang *et al.* as well. HIDE is a cache extension and hides the memory access pattern by dividing memory into blocks with the size of a cache line and permuting these blocks in memory. Before data is written back to memory, all remaining blocks of a larger memory region (e.g. 64k) are read and then written back to memory using a new permutation scheme. This is similar to Oblivious RAM [29], but it consumes less memory for the permutation information and performs better.
These memory access-hiding schemes can hide all fine-grained memory access, but due to the limited cache size and the performance penalty these schemes can only be applied to memory regions with a size of a few hundred kilobytes. Therefore, coarse-grained memory access cannot be hidden. Hence, these schemes can prevent the described statistical attacks but may reveal enough information for other classes of attacks.
A memory decryption attack is presented in [84]. The attack requires memory encryption using counter mode, because in counter mode each flipped bit of the ciphertext results in a flipped bit at the same position of the decrypted plaintext. The attack is based on an opcode manipulation whereby each instruction is changed into a *jump* instruction, which, when executed, gives the attacker the opportunity to record the target address on the address bus, thus revealing more bits of the modified instruction. Even when using HIDE, parts of the target address may remain unmodified and may give the attacker enough information

to reveal parts of a cryptographic key. For example, when using HIDE with a 32-bit RISC processor to hide the addresses of a 1 Mbyte ($2^{20}$ byte) memory region, there are at least $32 - 20 = 12$ bytes unchanged. This might be enough information for a attacker if these bits belong to a secret key. Note that in most cases the protected memory region will be much smaller, resulting in more disclosed bits.

## 5.2 Security Architectures

A security architecture combines many protection schemes to prevent manipulations and attacks. This section describes several approaches to provide a secure execution environment.

### 5.2.1 Smart Cards

The most widely spread secure execution platform are smart cards [75]. A smart card may contain a processor, memory (RAM, ROM, EEPROM) and input and output controllers for communication. They can be used as an electronic purse system, for identification purposes or as a secure data storage.
Smart cards often contain cryptographic units. They are used to digitally sign data like e-mails or can be used for encryption purposes. Smart cards are more secure than computer-based solutions, because the cryptographic secret or private key does not leave the card, and the user of a card has to authenticate himself before using the card. To further complicate stealing keys and other private data stored on smart cards, some of them can encrypt all data before storing it in memory.
The security of smart cards has been analyzed and many security flaws and attacks like side channel attacks have been found [32, 47]. Other attacks force computation errors to get further information about cryptographic keys [44]. Most of the attacks presented have been successfully demonstrated on smart cards, but they apply to other hardware cryptosystems as well. Hence, the countermeasures which have been invented for smart cards can be used to defeat attacks on other cryptosystems.
Software can be developed for smart cards, but due to hardware restrictions like limited processing power, small memory, a limited communication protocol, and complex programming these cards cannot be used for general-purpose applications. To overcome the complex programming interface, SUN[1] has developed the JavaCard [90], a smart card which is able to execute java byte code [52]. Java as a programming language simplifies the usage of smart cards but this card is not widely used.

### 5.2.2 Secure Co-Processors

Personal computers' security can be improved using secure co-processors. IBM[2] has developed the 4758 [16] secure co-processor as a PCI (Peripheral Component Interconnect) card. The card contains a processor, memory, a random number generator, and cryptographic devices housed within a sealed tamper-responding environment. Recently, IBM has presented a new co-processor architecture [2] which provides a faster processor and an increased security. Most secure co-processors follow the security requirements defined in [22] by the National Institute

[1] `http://www.sun.com`
[2] `http://www.ibm.com`

of Standards and Technology (NIST). The standard defines four security levels and covers physical security, authentication, operating system requirements, and key management.
Secure co-processors can be considered as a more powerful smart card. They embed many parts of a computer system and the whole module is protected against physical attacks and software attacks. They are used in automated teller machines and other systems where sensitive data has to be protected and security is more important than costs. The main drawbacks of secure co-processors are the additional costs (a processor card costs more than 20,000 US-$) and increased software development complexity. The co-processor runs embedded Linux but for communication with the host system special communication protocols have to be used. Therefore, all the software for co-processors has to be newly designed and existing software cannot be reused.

### 5.2.3 LaGrande

Intel released their preliminary architecture specification of the LaGrande technology in 2006 [38, 37]. This technology is connected with Intel's virtualization attempts to provide a hardware protected operating system running in parallel to an unprotected one. LaGrande consists of a processor extension which works in conjunction with a cryptographic enhanced chipset, input and output devices to provide a secure execution environment, for example for the Next Generation Secure Computing Base (NGSCB) by Microsoft [59], with protected input and output paths. The whole boot process including the BIOS (Basic Input Output System) and the bootloaders are protected as well. This results in a very complex system with a huge protected codebase, which complicates code auditing. Memory contents are not encrypted, but the chipset is used to control memory access.
Closely related to this architecture is the Trusted Platform Module (TPM) [92] developed by the Trusted Computing Group [93]. The module itself is not able to encrypt memory contents. This has to be done in software by the main processor. The module can only be used to store small amounts of information like encryption keys. Additionally, a TPM provides cryptographic functions which can be used to protect the boot process and for identification purposes.
Microsoft's Windows Vista operating system can use the TPM to protect the boot process and provides trusted video and audio output paths to protect digital content. The 64-bit version allows loading of signed drivers only to protect the operating system from malicious drivers. However, this design is mainly built to protect the operating system and digital media on the operating system level. As described in chapter 2, this software-only approach cannot defeat sniffing attacks as long as no additional cryptographic hardware like LaGrande is used.
There are other research projects like perseus [69][3], an open-source security kernel, or Nizza [31], a small secure kernel, which are based on the LaGrande architecture. Those trusted operating systems can then be used with virtualization technologies to protect for example middleware systems used for Grid computing [53] to provide a secure execution environment.

### 5.2.4 Digital Rights Management

Digital Rights Management (DRM) describes methods of copyright owners to restrict or control the distribution of their digital content. For example, DRM techniques are used to protect

[3] `http://www.perseus-os.org`

digital music or movies. Most DRM systems encrypt the digital data, and only dedicated players implementing the required cryptographic algorithms and knowing the required decryption keys are able to process the protected data. Popular examples for DRM systems are Apple's FairPlay technology for digital music or the Advanced Access Content System (AACS) for BluRay or HD-DVD movie discs. Like all other copy-protection schemes described in chapter 3 both technologies have been broken. FairPlay's encryption has been analyzed using reverse engineering by Jon Lech Johansen[4], who also wrote the program DeCSS to decrypt DVD's, to allow playing of protected songs with players not licensed by Apple. By time of writing this thesis, AACS has not directly been broken, but there exist methods to extract encryption keys from popular players which can be used to decrypt and copy-protected media. Shortly after this attack, Slysoft started selling a program to decrypt AACS protected media.

### 5.2.5 X-Box

The X-Box was Microsoft's first attempt to build a video game console based on the X86 architecture. To prevent the execution of unlicensed programs or pirate copies Microsoft created a security system which implements a security model similar to some parts of the LaGrande architecture. However, the security system was less effective and several parts of it have been broken. Microsoft uses a chained boot code, where beginning from the first executed code all further code is first verified before it is executed. Due to software errors and misuse of cryptographic algorithms, this chain has been broken. Additionally, a symmetric key used to encrypt one of the bootloaders could be sniffed even on high-speed HyperTransport bus by using dedicated hardware [36]. A summary of all security flaws of the X-Box design can be found in [86].
The new X-Box 360 uses a different hardware and software architecture which has not been deeply analyzed up to the time of writing this thesis. Therefore, there is no public information about the new security design available.

### 5.2.6 XOM

The eXecute Only Architecture (XOM) [50, 51] is a processor extension which supports execution of encrypted programs and encrypted program data. In contrast to a co-processor, XOM enhances an existing processor design with additional security functions. Each program is executed within its own compartment, and all data leaving the processor can be encrypted transparently. The encrypted part of a program is entered and left by executing a special instruction.
XOM uses ECB encryption to store data encrypted in memory. Encrypted data is additionally protected against modification by using a MAC which incorporates the virtual base address of the data into the hash computation. To prevent replay attacks the author suggests to compute a MAC over all protected data by the program itself, but this decreases the performance on frequently changed data. To overcome this issue the usage of a hash tree approach is suggested, but not implemented.
Access to a compartment is granted using a unique symmetric session key $k$ assigned to each program. A special null compartment without encryption capabilities exists to share data between processes. All processed data stored in registers or memory is tagged with a process-dependent identifier, and each process can access data tagged with its own tag. All

[4] `http://gigaom.com/2006/10/02/dvd-jon-fairplays-apple/`

other accesses result in an exception which terminates the current process. XOM does not trust an operating system therefore the operating system does not have access to tagged registers. However, the operating system requires methods to save and restore the processor status including the register contents on context switches. Therefore, XOM provides two instructions to save and restore the processor state to encrypted memory.
To each processor a public/private key pair is assigned by the manufacturer, and the private key can only be accessed by the processor but not by any program running on the processor. The public key is used to encrypt the session key so that only the dedicated processor is able to get $k$ and to execute an encrypted program. XOM switches between the execution of encrypted and unencrypted instructions by executing two dedicated instructions. Unencrypted data can be accessed only by dedicated instructions.
To limit the required amount of additional hardware XOM can be implemented using a so-called "XOM Virtual Machine Monitor" (XVMM). This virtual machine consists of on-chip firmware which handles most security parts like cryptography and tagged register handling. The drawback of this approach is the limited performance in contrast to a hardware only implementation. Therefore, a hardware only implementation is proposed as well.
In [49] an operating system design for XOM is presented. The design covers process creation, shared libraries, interrupt handling and paging. The whole operating system design is rather complex and requires changes compared to the first XOM implementation, since the OS has to save registers which might be marked with different tags. All MAC's have to be stored at dedicated physical addresses, which limits the flexibility of the operating system, because a certain amount of memory has to be reserved for these MAC's. Paging is possible, but the data and the corresponding MAC's have to be paged out and in at the same time. The usage of shared libraries is possible, but they cannot be protected, and therefore it is possible to alter the unprotected parts of a program with malicious libraries. The biggest drawback is the lack of secure system calls. Therefore, a protected algorithm has to be left each time the operating system is called. This could be used by an attacker to alter the program flow. This does not decrease the security of the protected part, but affects the security of the protected program as a whole. The XOM architecture can thus be used to protect small parts of a program like secret algorithms.

### 5.2.7 AEGIS

Suh's AEGIS [89] architecture can be considered as a successor of the XOM architecture. AEGIS shares some design principles with XOM like the ability to protect only parts of a program and dedicated instructions to enter and leave a protected part. Other parts have been greatly improved.
AEGIS consists of a hardware part and a software part. The software part acts as a secure operating system kernel which enhances a normal operating system. The secure kernel has to be activated before any protected program can be executed. The processor computes a hash over the protected part and other sensitive values like the entry point and stores it in a secure on-chip register. A similar approach is used each time a protected part of a program is entered. AEGIS contains a private/public key pair as well, but in contrast to XOM this key can be used by the processor to sign the program and the protected kernel hashes to prove that both the program and the kernel are unmodified.
In contrast to other architectures like XOM, the secret key is not directly stored in on-chip memory like ROM or an EEPROM (Electrically Erasable Programmable Read-Only

Memory), because this kind of memory can easily be read by an attacker with physical access to the processor core if no further hardware mechanisms prevent this attack. Instead, AEGIS uses Physical Random Functions (or Physical Unclonable Functions, PUF) [28]. They use variations of physical properties of a silicon chip, which are different even for chips on the same wafer produced with the same mask. The output of a PUF is then used to encrypt a private key.
AEGIS provides encrypted, verified read-and-write and verified read-only memory. Verified read-only memory is protected with MAC's, for read/write areas a cryptographic hash function and Merkle trees are used. Data is encrypted using AES counter with a time stamp as a counter. The time stamp has a size of 32 bits and is stored in a dedicated memory region, thus increasing the memory footprint by approx. 6 %. This potentially gives rise to more misses during memory operations. Depending on the number of memory access, this counter can overflow resulting in a time-consuming re-encryption, using a new key, of all program-related data. Longer counters can prevent this for most programs, but they consume more memory.
The required hashing step before starting a program and the need for re-encryption during runtime are making the whole architecture more complex, since these actions take a long time and many memory accesses, while the program has to stay suspended during that time. A larger counter value can prevent counter overflows in most cases at the cost of an increased memory demand. AEGIS uses speculative execution to speed up program execution. To circumvent the attack presented in [84] AEGIS suggests the usage of memory access hiding schemes like HIDE or oblivious RAM. But as described in section 5.1.3, this cannot prevent this attack entirely and therefore AEGIS is still vulnerable to this attack.
AEGIS requires a protected operating system kernel to be able to execute protected programs. This kernel requires exclusive access to memory management, otherwise the security of a protected program cannot be ensured, since memory protection is implemented in the TLB. An unprotected operating system could alter the page table to circumvent the protection. The drawback of this approach is the performance degradation even in cases where no protected program is to be executed due to the additional security checks required for the protected operating system. AEGIS can leave a protected execution environment and later enter it again using dedicated suspend and resume instructions. They are used for example by the operating system to implement system call handlers in unprotected memory. AEGIS memory verification is based on physical addresses thus making paging difficult. Therefore, AEGIS can page only the whole data protected by a hash tree to memory and later load it to a possibly different address, but paging only arbitrary pages is not possible.
Programs for the AEGIS architecture require a modified compiler. The compiler has to support protected, encrypted and unprotected memory and transitions between these memory areas. For each variable and function the target memory area has to be specified by the programmer based on the required protection. Hence, the programmer has to analyze his program to split it into protected and public parts. This is the main disadvantage of the proposed AEGIS programming model [79], because most programmers are not aware of possible attacks, and a classification of sensitive and public parts is very difficult for inexperienced programmers, because side channels or other attacks can easily be overlooked. Furthermore, three different memory areas require three different stack pointers which complicate the whole design.
Due to the complexity of the processor's security functions, most instructions are implemented using an internal firmware. This reduces the hardware costs at the expense of additional delays, which occur when longer tasks like re-encrypting on counter overflows have to be performed

by the firmware. Additionally, the internal firmware consists of normal program code which may contain programming errors. Of course, erroneous implementations are possible on the hardware level, too, but a design with less complexity can be verified more easily.

# Part II

# The Security Architecture for Microprocessors (*SAM*)

# Chapter 6

# *SAM* Design Goals

This chapter summarizes the design goals for the *SAM* architecture.

## 6.1 Motivation

The first part of this work has shown threats to software and possible solutions. It has been shown that software-only solutions cannot provide a secure execution environment. Only the presented hardware-based security architectures are able to provide a secure execution environment. However, each architecture has disadvantages which prevent their usage for general-purpose applications. Smart cards are limited in terms of computing power and flexibility, and secure co-processors are too expensive. Intel's and Microsoft's plans for a secure execution environment require a lot of changes to existing hardware, but they are likely to be implemented in future computers. Their scope is focused on digital content protection and Digital Rights Management. Therefore, they require cryptographic enhancements even for peripheral devices like sound cards, which makes the architecture rather complex. Though, memory protection is not covered by LaGrande, which makes the architecture vulnerable to hardware-based sniffing attacks.
XOM and AEGIS are designed as a drop-in replacement for a standard processor and they provide a memory protection scheme as well as an operating system design to make the security functions available to user programs. As described in chapter 5, AEGIS has improved many concepts of XOM, especially the usage of physical random functions greatly improves the security of the protected programs. However, AEGIS uses counter mode encryption without proper countermeasures to prevent memory decryption attacks due to speculative execution of instructions.
Unfortunately, the whole AEGIS architecture is rather complex. There are a lot of new instructions and differently protected memory areas which require modifications to both the compiler and the assembler. AEGIS allows to partially protect and encrypt a program on a function and variable basis. This gives more freedom to a programmer, but in security environments more freedom often results in more mistakes. For example, the programmer can overlook security-related dependencies by not protecting a vital variable resulting in an exploitable flaw in his program. Therefore, most programmers will choose to protect the whole program, and this can be done with a less complex architectural design.
Due to the complexity, some instructions are not implemented in hardware but in firmware, which may lead to longer stalls in which the processor is not able to execute user instructions

or operating system instructions. This may happen in cases where a counter overflow occurs, because it results in a full re-encryption of the program. Another problem may arise if in those cases parts of the protected parts of program are paged out to external storage and are not directly available to the processor. Another drawback of the architecture is the increased memory footprint, as both encrypted memory and hashed memory require additional data like the hashes and the counter. Furthermore, different handling of read-only and read/write-areas requires two different memory verification units, which complicates the design.
Additionally, parts of the AEGIS kernel have to be permanently protected to be able to run protected programs. This results in a permanent performance degradation even if no protected program is executed. Furthermore, each time a new protected part is created the processor has to compute the hashes for this memory area. Therefore, the whole protected part has to be loaded to memory at once and memory-efficient loading of protected pages on demand is not possible. This further limits the flexibility of the operating system.
In this thesis, the *SAM* security architecture is presented. *SAM* has been designed to overcome some of the disadvantages of the existing security architectures described above while providing at least the same level of security. The architecture's design goal is to provide a secure execution environment based on an enhanced microprocessor. The architecture should be mostly independent from specific processor architecture to be portable to different processor architectures. To be able to use this processor as a replacement for a normal processor the number of architectural changes is kept low. The design should be less complex to support a direct hardware implementation without additional firmware requirements and without reducing the security level.
Hardware-only security architectures are possible [49], but they complicate the hardware design and have limitations in terms of a flexible operating system design and performance. Therefore, *SAM* is based on a combined hardware- and software-based architecture. In the following the security requirements for *SAM* are listed.

## 6.2 Requirements for Secure Computing

The *SAM* architecture has been designed with the following goals in mind:

1. *Protection of program code and data.* The program should act as a black box, which hides all executed instructions as well as all processed data. Even administrators or direct hardware access should not reveal any information about the program.

2. *Prevention of any unintended modifications during program execution in a multitasking environment.* A program has to be executed in the way intended by the programmer or automatically terminated immediately. Of course, program interruptions should be allowed since they are required in a multitasking environment.

3. *Parallel execution of protected and unprotected programs.* Typically, not all programs have to be protected. Hence, protected and unprotected programs should be executable in parallel without lowering the security for protected programs.

While some of the attacks employed to break one of these goals cannot be prevented directly, (for example, parts of the executed program code may have been changed in memory or a power blackout occurs) they must be detected so that the affected program is aborted immediately. If any tampering attempt results in a termination of the program, then each

program that exits normally has not been tampered and it has been executed in the intended way. However, *SAM* does not try to prevent any kind of programming errors. It is up to the programmer to ensure proper prevention of buffer overflows and other possible attacks when processing external data.
These goals can be used for efficient copy-protection schemes, because programs cannot be modified to bypass them. The same applies for DRM implementations, because a program that cannot be analyzed can be used to hide audio and video codecs or encryption keys. However, *SAM* does not try to make program execution itself more secure for example by preventing buffer overflows or other attacks based on eternal data, but existing countermeasures against this kind of attacks can of course be combined with *SAM*.
In the following, the hardware- and software-related requirements to meet the goals for secure computing are summed up.

### 6.2.1 Hardware Requirements

The following prerequisites can only be guaranteed in hardware:

- *Tamper-resistant processor core*: All data inside the processor has to be protected against external physical attacks to prevent manipulations used to reveal data or manipulations related to the execution of programs [22]. As a direct consequence, all data leaving the processor has to be protected, since modifications or sniffing attacks to external data cannot be prevented physically by the processor.

- *Tamper evidence*: The processor has to be able to detect any kind of tampering attempts before they can be exploited by an attacker. That means that any kind of tampering does not have to be detected immediately, but in any case before an adversary could take an advantage from it.

- *Restricted execution*: If a program is intended to be executed on a tamper-resistant processor core only, then the execution on other, not protected processor cores, has to be prevented. Hence, only the intended secure processors shall be able to execute the program code. Otherwise, attacks may be possible, especially if a simulated processor could be used to execute the program. Furthermore, this results in an effective copy-protection.

- *Algorithm and data protection*: Programs and processed data are typically too large to allow for direct storing inside the secure processor caches. Hence, all data and instructions (in the following the term "data" refers to instructions and other data) stored outside the secure processor in RAM must be protected against tampering attempts. The following two attacks have to be prevented:

    1. *Data disclosure*: Data may be read by an attacker to analyze algorithms or processed data. Possible attacks are, for example, memory dumps by bus master devices or hardware-based sniffers.
    2. *Data modifications*: Even if data disclosure attacks are prevented, the normal program operation can be altered by an attacker by modifying memory contents. This includes replay attacks, where data is replaced by older, but at that time valid, data.

- *Prevention of program flow analysis*: A (statistical) analysis of the program flow may reveal used algorithms in a program or may be used to restore cryptographic keys by examining the program flow of identified algorithms.

The requirements mentioned above provide a secure execution of one program in a single task environment. In this case, the whole program is executed in a secure execution environment, which prevents any kind of external modifications.
However, these requirements are not sufficient for a multitasking environment. In these environments, programs can be interrupted at any time by the operating system or they may issue a system call. In both cases the protected execution environment is left. This allows new attacks addressed below.

### 6.2.2 Software Requirements

Attacks in a multitasking environment can be prevented if the whole operating system is protected as described in the previous subsection. But this would result in protecting all drivers resulting in a huge protected codebase. More code contains more security-related programming errors and therefore the protected codebase should be kept as small as possible. Another positive aspect of a smaller protected codebase is the improved performance: indeed any additional security check performed by the processor is likely to decrease the execution speed. Furthermore, the unprotected part can easily be changed or enhanced without affecting the security of the whole system.
Therefore, a *SAM*-enabled system should meet the following additional goals to provide the required security:

- *Proper interrupt handling*: Interrupting a protected program in a multitasking environment and restarting it later without the possibility of modifying any program-related data in the meantime must be ensured. Hence, the state of the program must be fully reconstructed after an interrupt in every case without the possibility to modify program data in an unauthorized way. This leads to the following requirements during an interrupt:
  - *Prevention of program flow manipulations*: Any manipulation to the program flow could be used to execute the program in the wrong order, to skip parts of the program or result in false computations.
  - *Prevention of unauthorized memory access by the operating system*: The OS has full access to the whole memory area of the program and knows all dynamically requested memory regions as well as the size of the stack. Therefore, any kind of manipulation or examination of memory areas by the operating system has to be prevented. However, some memory access should be allowed, since, for example, the register values have to be stored in memory during a context switch.
- *Access to external data*: A program typically relies on data read from files or the network or user-provided data and creates new data. Hence, the security architecture has to provide mechanisms to pass external data to a protected program. However, external data can always be manipulated and, therefore, each program dealing with external data has to verify this data by its own by using cryptographic protection mechanisms. For example, data received over the network is outside the scope of a secure processor.

In the following chapters the *SAM* architecture is presented.

# Chapter 7

# Processor Architecture

This chapter describes the main properties of the *SAM* processor architecture. The properties here are mostly independent of the underlying processor architecture. Proposed implementations of *SAM* based on the SPARC and on the IA-32 architecture are described in chapter 9. In the following the computer system in which a protected program has been compiled and prepared for *SAM* is referred to as *build host*, and the system which executes protected programs is called "target host".

## 7.1 Overview

The *SAM* architecture provides memory encryption and memory verification in conjunction with an enhanced and tamper-resistant processor core to prevent sniffing and tampering attacks. As can be seen in figure 7.1, *SAM* adds several components to a normal processor, all of them will be addressed below. *SAM* provides protected and unprotected processes, distinguished by their context identifier. An unprotected process behaves like a normal process executed on a processor without *SAM* enhancements. Therefore, *SAM* can be used as a direct replacement for an existing unprotected processor, and protected and unprotected programs can be executed in parallel.
*SAM* does not require a protected operating system start-up like AEGIS or the LaGrande architecture. Instead, a design which protects parts of the operating system and the protected process only during the runtime of the protected process has been chosen. This helps to prevent performance degradations in cases where only unprotected processes are to be executed and simplifies the boot process of the operating system.
The current *SAM* architecture described in this work does not support dynamic linkage. That is, all protected programs have to be statically linked. This design simplifies the whole architecture and gives the author of a program more flexibility, because it is not required to use specific protected library versions provided by the target host. In the following, all *SAM* components are described in detail.

## 7.2 Cryptographic Keys

In connection with cryptographic functions keys have to be exchanged and stored. Like XOM, *SAM* uses symmetric and asymmetric cryptographic functions for protection. Each processor $P$ has its own private key $k^P_{priv}$ and public key $k^P_{pub}$. The private key remains in the processor

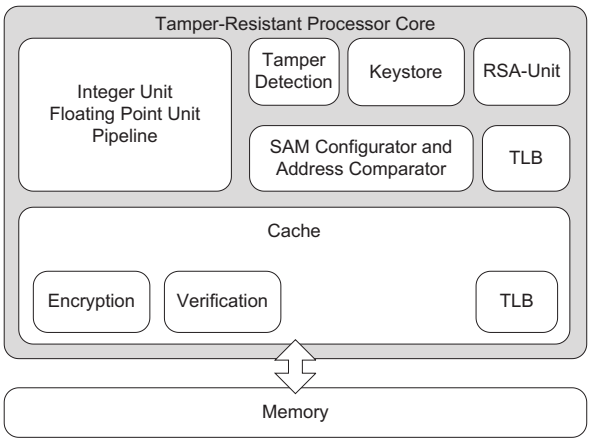



Figure 7.1: Schematic *SAM* processor overview

core and cannot be read by software. The corresponding public key can be read, and a copy of this key should be signed by the manufacturer to prove the origin of each manufactured processor. The public key can then be used to encrypt data that should only be readable by the corresponding processor. The usage of physical random functions is possible, too, but it is not considered in this thesis.
Each executable $\mathcal{E}$ is encrypted using $k_b$, a unique secret key randomly chosen at program creation time resulting in an encrypted executable $\mathcal{E}'$. In addition to this key, a random value $r$ is generated for each program for memory verification purposes and hash tree generation (see section 8.1). Both $k_b$ and $r$ are encrypted with $k_{pub}$ and attached to the encrypted program (where || denotes a concatenation):

$$\mathcal{E}' = E_{k_b}(\mathcal{E})||E_{k^P_{pub}}(k_b||r)$$

If $\mathcal{E}$ should be executable on more than one processor, $k_b$ can be attached multiple times encrypted for each processor:

$$\mathcal{E}' = E_{k_b}(\mathcal{E})||E_{k^{P_1}_{pub}}(k_b||r)||\cdots||E_{k^{P_n}_{pub}}(k_b||r)$$

Internally, the processor has to store $k_b$ in addition to other configuration parameters described below for each executed protected program. This information is stored in on-chip tables and the tables are indexed using a program-specific identifier, the context number.
*SAM* implementations are supposed to support additional, possibly user-configurable, asymmetric key pairs, for example a unique key stored on all *SAM*-enabled processors to simplify distribution of protected programs. Using this key, a program could be executed on all *SAM*-enabled processors without the need to prepare it for each specific processor.
Of course, the standard tool chain (compiler, assembler and linker) does not generate protected programs and is not able to generate the required keys or encrypt programs. To limit the number of changes to the tool chain, the *SAM Linker* is used to transform the previously statically linked program into a protected program by encrypting the program and by adding keys and program-verification-related data. A more detailed description of this program is given in section 12.2.

## 7.3 Tamper Detection Unit

This unit handles all detected physical and logical attacks. In the following, logical attacks are denoted as security violations. As described below, they include memory verification errors or invalid usage of instructions. Each detected security violation results in an immediate termination of the currently executing protected program and program-related data deletion, such as fault addresses. Additionally, the cache clears all lines owned by the faulty context and the key store deletes the corresponding key $k_b$. Security violations can only affect protected programs. They are ignored when executing unprotected programs only.
Physical attacks to the processor core are always handled, and they result in the immediate deletion of all keys including the private key $k^P_{priv}$, as described in [2].

## 7.4 RSA Unit and *SAM* Configurator

The RSA unit has exclusive access to $k^P_{priv}$ and is used to decrypt $E_{k_{pub}}(k_b)$. After decryption, the *SAM* configurator stores $k_b$ for the given context in an internal key store. Then, the root hash for the corresponding context is loaded and stored decrypted in the hash store. Both the hash store and the key store can only be read by the cache: they are not readable by instructions, but their configuration can be controlled using memory-mapped registers.

## 7.5 Cryptographic Functions

All internal cryptographic units, like AES or RSA, and a hardware random number generator can be made available to user programs. This further increases security, because this can be helpful to avoid statistical attacks using the memory access pattern, such as those cited in [100]. Of course, access has to be implemented in a way that access to all *SAM*-related keys is prevented.

## 7.6 Security-Aware Cache

The cache is a major part of *SAM's* security architecture. It contains all cryptographic parts required for memory integrity verification and memory encryption and decryption. The basic cache architecture has been published in [73], but the current architecture described in chapter 11 has been enhanced to provide a better performance by using prefetching mechanisms.

## 7.7 Memory Layout

A *SAM*-enabled processor has to be able to know if memory contents are encrypted or if their integrity has to be verified. For AEGIS and XOM dedicated instructions are used to enable this for smaller parts of a program. The *SAM* architecture does not support partly protected programs, because in these the integrity of the whole program cannot be ensured. For example, the program flow of unprotected parts can easily be changed resulting in an unexpected program behavior. A fully protected program simplifies program development, because many existing functions and libraries can simply be used without any modifications by statically linking them to the protected executable.

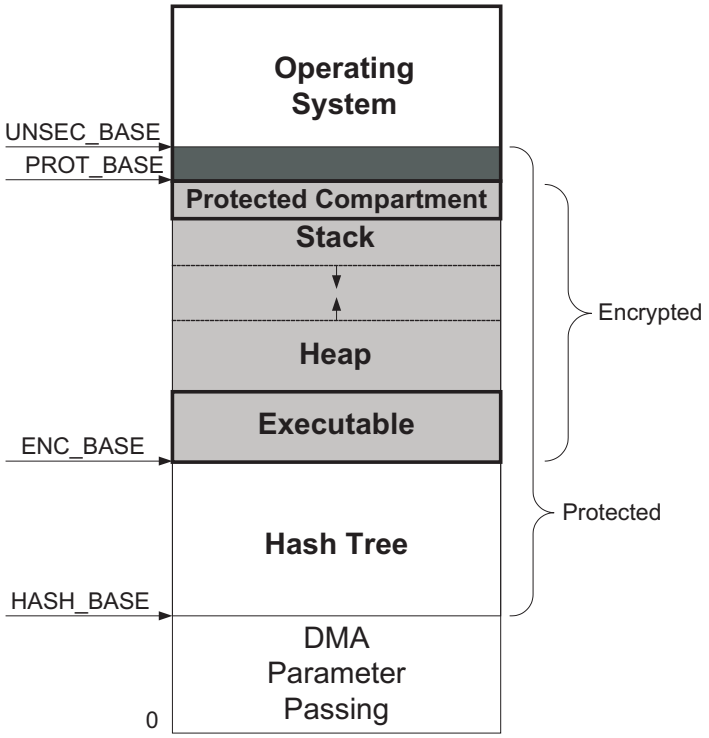



Figure 7.2: Memory Layout for each protected process

Since no partly protected programs are permitted, no dedicated instructions to enter a protected part are required, and memory protection can be enabled based on the virtual address. Figure 7.2 shows *SAM's* relatively fixed and simple memory layout for a protected program chosen with hardware implementation in mind. All addresses are virtual, resulting in a free mapping from virtual to physical addresses, with a double positive effect: avoiding contiguous physical memory clusters and allowing for paging of unused parts.
For each protected context only four base addresses are stored, which are marked in this figure with the following labels:

- *HASH_BASE* denotes the position of the root hash and therefore the beginning of the hash tree. This section is called "hash tree" or "hash memory area" in the following.
- All code and data between *ENC_BASE* and *UNSEC_BASE* is protected by hash values. This memory region can be used as read-only shared memory, for example, for protected parts of an operating system. In the following, this memory area is called "protected memory area" or region.
- Between *ENC_BASE* and *PROT_BASE* the cache transparently decrypts all read code and data and decrypts all written code and data. This section is referred to as "encrypted memory area".
- All memory above *UNSEC_BASE* and all memory below *HASH_BASE* is unprotected. These parts are dedicated for major parts of the operating system, DMA areas and parameter passing between the operating system and the protected program.

The corresponding memory layout for each protected process is set by the operating system when loading a protected program, and it remains static during the whole execution (see chapter 10).

### 7.7.1 Memory Views

Access to decrypted parts cannot always be granted and hash values must not always be computed or checked: If for example, memory contents are decrypted and hash values are computed during write-back only if the line was written by an instruction protected by hash values of the same context. Likewise, access from arbitrary instructions to decrypted memory regions must be prevented, because otherwise the administrator of a computer system may access any part of the memory contents by simply reading them.
This results in the internal classification of protected or unprotected instructions resulting in two different *views* of the main memory:

- Protected instructions can access all memory regions, and encrypted main memory contents are decrypted in *SAM's* cache. On write-back hash values are computed and the hash tree is updated.
- Unprotected instructions can access the whole memory area, too, but encrypted parts remain encrypted and hash values are not generated during write-back of dirty parts.

In order to be able to distinguish memory access by protected and unprotected instructions on write-back and cache line replacements, the memory system has to be enhanced to support tagging of memory regions. This memory view design has been chosen instead of simply preventing memory access to protected parts, because the page mapping can freely be modified by the OS. Therefore, it would still be possible to access a protected physical page by mapping this page to an unprotected address space. Of course, this page can then be accessed only encrypted and hash values are not updated. This is in contrast to AEGIS, which requires that the parts of the operating system maintaining the page mapping be trustworthy and protected.

## 7.8 Protected Operating System

The memory view design can prevent access to protected memory regions by unprotected instructions, but it does not prevent the unintended execution of protected instructions. An attacker who is able to modify the operating system can still *jump* to protected parts of the program to execute them. This would allow an attacker to skip or re-execute parts of a program and therefore it has to be prevented. *SAM* prevents this attack by distinguishing between protected instructions executed in user mode and in supervisor mode. This is realized using several flags (listed in table 7.1) representing the current protection level. The `PM` and `PE` flags are set based on the current value of the program counter, and the `PV` flag is set by the cache system.
The `PI` flag is used to restrict instruction execution in supervisor mode. When set, the currently executed instruction has full access to memory, and encrypted memory contents are transparently decrypted. In user mode `PI` equals `PM`, resulting in full access for all protected instructions as described above. In supervisor mode, the flag `PT` has to be set as well to enable full access. This flag is automatically set when the processor performs a transition from

| Flag | Description |
|---|---|
| S | *Supervisor Mode.* Set each time the processor operates in supervisor mode. |
| PP | *Protected Process.* This flag is set during execution of a protected process. If it is unset, the following flags are always unset. |
| PS | *SAM-enabled.* This flag is set when the first protected process is configured and cleared when all protected processes have ended. |
| PT | *Protected TRAP.* This flag is set on each TRAP for a protected process. |
| PV | *Protected Data Verified.* Each time all current data has been verified, this flag is set. |
| PM | *Protected Memory.* Set if the current instruction is located in protected memory. |
| PE | *Encrypted Memory.* Set if the current instruction is located in encrypted memory. |
| PI | *Protected Instruction.* PI = PM $\wedge$ ($\overline{\text{S}} \vee$ PT). This flag reflects the special handling of protected instructions executed in supervisor mode and equals PM for user mode instructions. The processor does not store this flag, its value is always computed using the given Boolean expression. |

Table 7.1: *SAM's* processor flags

user mode to supervisor mode (TRAP), but it can be cleared by the operating system to drop its privileges using the *st* instruction described in section 7.9. This instruction has to be executed before leaving the protected part of the operating system to clear PT to disable access to encrypted memory regions. Each time the operating system wants to enable full access to encrypted memory regions *st* has to be executed again to set PT. Means to prevent unauthorized setting of PT are described in chapter 10. This definition of PI serves two purposes: It prevents direct data access by the unprotected parts of the operating system as well as execution of encrypted instructions.
To prevent attacks initiated in supervisor mode the *SAM* architecture implements additional checks listed in table 7.2[1] with the S flag in the processor. In order to prevent arbitrary transitions from supervisor to user mode, this flag can only be cleared if, for the clearing instruction (for example an instruction which returns from a TRAP), the PI flag is set. Therefore, only protected instructions can return to a protected user space.
Additionally, in supervisor mode the execution of encrypted instructions is prevented. Otherwise a malicious operating system may try to directly jump into user mode to execute arbitrary instructions.

### 7.8.1 Protected TRAP Table

To support a protected operating system kernel, *SAM* ensures that the TRAP table, which is the entry point for all operating system calls, is located in protected memory (see table 7.2). However, this requirement (listed in [72]) is not sufficient to prevent attacks, since the pointer to the TRAP table base address $T_{base}$ might have been changed to point to another location in protected memory. Although it is very likely that this attack would crash the system, *SAM* must prevent it. Hence, the current value of $T_{base}$ is incorporated into the memory verification

[1] In the table the notation $\rightarrow$ X has been used to represent a write access to X.

| Condition | Description |
|---|---|
| $\rightarrow$ S $\wedge$ PP $\wedge$ $\overline{\text{PI}}$ | Prevention of execution of unprotected instructions in protected TRAP mode. |
| S $\wedge$ PM $\wedge$ PE $\wedge$ PP | Prevention of execution of encrypted instructions in supervisor mode. |
| S $\wedge$ PT $\wedge$ $\overline{\text{PM}}$ $\wedge$ PP | Prevention of execution of unprotected instructions in protected TRAP mode. |
| S$\wedge$ $\rightarrow$ $T_{base}$ $\wedge$ PS | Prevent any modification of $T_{base}$ when in *SAM* mode. |

Table 7.2: Supervisor-mode-related security violations

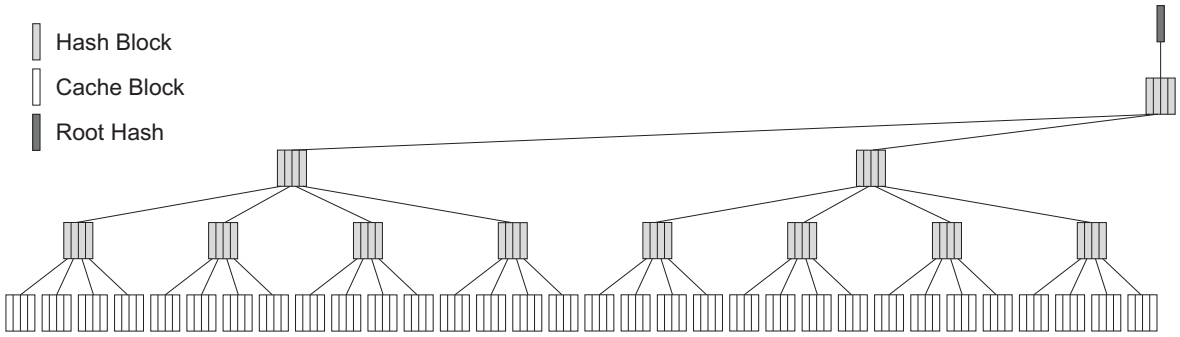



Figure 7.3: Hash tree layout

(see section 8.1 for further details), so that changing $T_{base}$ while PS is set is reported as a security violation.

### 7.8.2 Sparse Hash Tree

Hash values are stored in a cached hash tree [27], as can be seen in figure 7.3. The hash tree protects the whole virtual memory area containing the executable, stack and heap and parts of the OS (see figure 7.2). The size of the hash tree directly depends on the size of the protected memory region: the tree always consumes 25% of the whole protected virtual address space due to the size of a hash value as explained in section 8.1. *SAM* requires that all hash values have been precomputed at program creation time for all protected memory regions. This increases the protected program size, but disk space is cheap and it prevents time-consuming computations on program start-up and obviously simplifies the processor design. However, parts like heap or stack that are unused at time of program start do not need precomputed hash values. These values can be calculated the first time a corresponding region is used. Hence, hash values used to protect initially unused memory parts can be omitted in the executable to reduce its size.
In order to allow the cache to distinguish used and unused memory areas, unused areas are marked with a hash value of zero[2]. Fortunately, all memory pages containing zeros only need not to be stored within the executable, because the OS can map pages containing only zeros to these regions. Hence, the size of the executable can be reduced if the protected address space is mostly unused during program start-up.

[2] All hash value bits are set to zero.

Each time a hash value of zero is read, the protected content is assumed to be valid, regardless of the actual contents. In order to prevent attacks the cache still verifies this zero value using the corresponding parent in the tree. To cope with the unlikely case of a computed hash value of zero, the hash value is set to one[3]. Algorithm 7.1 illustrates the recursive hash verification procedure `verifyCacheLine` used for memory integrity verification. Of course, each time data is written back to memory the processor updates the hash values.
The sparse hash tree design requires some operating system modifications (described in section 10.2.3). They are required to guarantee that always all required pages possibly accessed by the cache to verify memory contents are mapped to physical memory. In cases where the cache is unable to resolve a virtual address pointing to a required hash value, the processor instantly reports a security violation to terminate the protected program.

**Algorithm 7.1**: Function verifyCacheLine(Address, CacheLine)

```
computedHash=computeHash(CacheLine);
rootHash=getRootHash();
if computedHash==0 then
    computedHash=1;
end
if Address==BASE_HASH then
    if computedHash ≠ rootHash then
        security_violation();
    end
else
    parentAddress=computeParentAddress(Address);
    memoryHash=fetchHashValue(parentAddress);
    if (memoryHash==0) or (computedHash==memoryHash) then
        verifyCacheLine(parentAddress, memoryHash);
    else
        security_violation();
    end
end
```

## 7.9 *SAM* Instruction Set

The extended security functions of *SAM* require new instructions. Compared with [72] the number of required instructions has been notably reduced with a simplified hardware design as a by-product. The current design requires only the following *Secure TRAP* (*st*) instruction. This instruction sets or clears the `PT` flag based on the register value of the operand to request or drop additional privileges. In the following, setting `PT` is referred to as *sts* (Secure TRAP Start), whereas clearing `PT` is referred to as *ste* (Secure TRAP End). This instruction can only be executed in supervisor mode and applies only to protected processes. It behaves like a *nop*[4] instruction for unprotected processes or when located in unprotected memory.

[3] Only the least significant bit of the 128-bit hash is set to one.
[4] No Operation.

Chapter 9 lists more additional instructions, but they are implementation-specific and not necessary for an abstract architecture overview.

## 7.10 Speculative Execution

Checking the data integrity is a very time-consuming task. Hence, it is desirable to execute most instructions speculatively prior to their verification. This improves the performance, but an attacker might exploit the small time frame between executing tampered instructions and detecting them to alter the program. As speculative execution cannot prevent tampering attempts, the architecture must ensure that these attempts cannot be used to reveal any secrets. For example, an attacker might try to replace protected but unencrypted instructions to copy secret data to unprotected memory.

### 7.10.1 Memory Decryption Attack

The *SAM* architecture as described in [73] prevents many attacks, but was, like AEGIS and XOM, vulnerable to an attack used to decrypt memory contents. The attack is described in [84] and is even easier to accomplish due to the SPARC instruction format which uses only two bits for a *call*[5] instruction opcode (see section 9.1 for more details of the SPARC instruction format). The attack is based on an opcode manipulation whereby each instruction is changed into a *call* instruction, which, when executed, gives the attacker the opportunity to record the target address on the address bus, thus revealing more bits of the modified instruction. This attack is possible because the adversary gets direct feedback on his attacks. Therefore, it is not sufficient to stall the processor on interrupts until all instructions are verified as suggested in [89] or [73], because sensitive information may still be stored in registers containing for example the faulting address on page faults. Even if these registers are cleared on verification errors, an attacker may try and sniff the address bus directly. Even with address bus randomization schemes, like HIDE, the attacker gets useful address information, since these schemes typically randomize small chunks of memory. For example, if the access to 64k Byte chunks ($2^{16}$ Bytes) of memory is randomized by HIDE only the information in the least significant 16 address bits is lost. The remaining 16 most significant bits are unchanged and can be used to reveal two bytes of memory.

### 7.10.2 Secure Speculative Execution

In order to prevent this attack more steps are required to not provide any kind of feedback to the attacker when executing instructions speculatively. Hence, *SAM* has the following limitations while executing instructions speculatively:

1. Instructions can be executed speculatively until an interrupt is executed or the context is to be changed. In both cases the processor stalls until any read data has been verified and then continues execution. Hence, modified instructions or data are detected at least before the next system call or context switch is executed. Therefore, modified program parts cannot be used to transfer data to disk or over the network, since this would require a TRAP.

[5] A *call* instruction is used to jump to arbitrary instructions in the virtual address space.

2. The cache defers program-caused fetches until all pending verifications have been finished successfully. This prevents data disclosure by address bus sniffing attacks. This does not decrease the cache performance in cases where there is enough time between cache misses.

3. All protected instructions trying to write to unencrypted memory or special registers are stalled until the cache has finished all pending cache line verifications.

# Chapter 8

# Memory Protection

This chapter describes *SAM's* memory protection scheme.

## 8.1 Memory Integrity Verification

Since hash values are cached, their size has to be a fraction of a cache line and a power of two for efficiency reasons. In *SAM's* case a cache line consists of 512 bits and each hash has a size of 128-bits. Thus, four hash values (called "hash blocks") can be stored in a cache line. Other hash sizes are possible, but smaller hashes provide less security against attacks, and larger hash values result in an increased memory footprint.
When computing hash values for encrypted data, the hashing function is typically applied to the encrypted data and not to the unencrypted data. This has the advantage that it is impossible to leak additional information by the hash value. For example, when applying a hash function to unencrypted data, an attacker could try to decrypt the data by performing a brute force attack using the hashing function. This attack is not possible when hashing encrypted data.
However, when designing *SAM*, one goal was the resulting performance of the cryptographic tasks. Hence, the additional amount of data required for encryption and hashing should be reduced compared with memory encryption schemes like used by AEGIS. Applying the hash function to unencrypted data has the advantage that the hash value can be used as a counter value for encryption (see section 8.2). Consequently, an additional counter value is not required, thus saving memory. Therefore, *SAM* always computes hash values on the unencrypted memory contents. To prevent information leakage the algorithm has the following properties:

- Each cache line $C$ is XORed with a random, program-dependent secret 512 bit value $r$ before computing the hash $H$. This value makes it impossible for an attacker to compute $H$ even when $C$ is known. Therefore, the hash values cannot be used as a "fingerprint" to identify known functions in a program just by comparing the hash values with precomputed ones. Furthermore, identical memory contents of two programs result in different hashes due to different $r$ values. The size of $r$ has been chosen to successfully prevent brute force attacks.

- Identical memory contents inside one program at different addresses always have to result in different hash values. Otherwise, an attacker would be able to move a cache

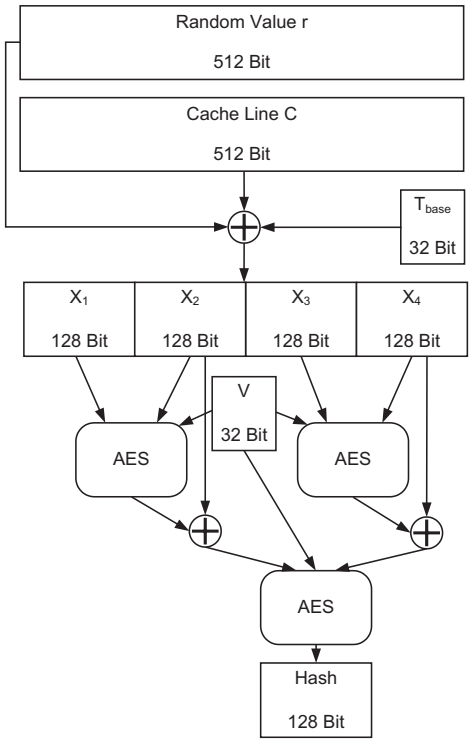



Figure 8.1: Hash value computation

line and its corresponding hash to a different address without notice. Furthermore, equal hash values could give an attacker valuable information about encrypted memory. For example, if hash values were computed over unencrypted and encrypted parts, equal hash values could directly be used to reveal parts of the encrypted memory. Therefore, *SAM* uses the virtual base address $V$ of the corresponding cache line to make the hash-value computation location-dependent.

- As described before, *SAM* requires a fixed address of the TRAP or interrupt table $T_{base}$. To ensure the correct value during protected program execution this value has been incorporated into the hash-value computation as well.

These requirements lead to the following hashing algorithm illustrated in figure 8.1. At first

$$C' = C \oplus r \oplus T_{base} \tag{8.1}$$

is computed. In the following, $C'$ is further divided into four 128-bit blocks $X_{1\cdots4}$, for convenience:

$$C' = X_1||X_2||X_3||X_4. \tag{8.2}$$

The hashing scheme has been suggested in [74, §2.4.4] and is based on a tree structure and provides the properties listed in section 4.2. Let

$$f(x, H) = E_x(H) \oplus H \tag{8.3}$$

be the rounding function, where $E_k(y)$ represents the application to $y$ of the AES function with key $k$. AES has been chosen to reduce the amount of additional hardware required to compute hash values, since AES is used for encryption as well.

The hash is computed as follows: First the following operations are performed in parallel to compute the two sub-hashes $H_1$ (over $X_1$ and $X_2$) and $H_2$ (over $X_3$ and $X_4$):

$$H_1 = E_{X_1 \oplus V}(X_2) \oplus X_2, \qquad H_2 = E_{X_3 \oplus V}(X_4) \oplus X_4. \tag{8.4}$$

When the previous step is completed, the following computation is carried out:

$$H = E_{H_1 \oplus V}(H_2) \oplus H_2. \tag{8.5}$$

The hashing scheme has been changed compared to the one described in [70] by incorporating the virtual address $V$ into the hash computation instead of XORing it with the plain cache line due to the following reason: Let $C_1$ and $C_2$ be two different cache lines located at the virtual addresses $V_1$ and $V_2$. Then $C_1'$ and $C_2'$ are computed using the old scheme as follows:

$$C_1' = C_1 \oplus r \oplus (T_{base}||V_1)$$
$$C_2' = C_2 \oplus r \oplus (T_{base}||V_2).$$

It can be observed that $C_1'$ and $C_2'$ are equal if

$$C_1 \oplus V_1 = C_2 \oplus V_2$$

resulting in an undesired information leakage. This case may happen quite frequently, because, for example, often data is stored in linked lists which may contain virtual memory addresses of other list elements. If these addresses are stored at the address of the last word of a cache line, the hashes may be equal. The new scheme does not have this undesired property, because the virtual address is directly incorporated into the hash-value computation instead of only modifying the plaintext.

Computing a hash over the plaintext further reduces the probability that an attacker may take advantage of the very unlikely case of equal hash values located at different addresses. Assume that $H_{C_1}$ and $H_{C_2}$ are the hashes of the cache lines $C_1$ and $C_2$ located at the virtual addresses $V_1$ and $V_2$. If $H_{C_1} = H_{C_2}$ then $C_1$ and $C_2$ cannot be exchanged without notice. The cache always recomputes the hash values over the plaintext for comparison, and due to the different virtual addresses $V_1$ and $V_2$ the hash values computed with equations 8.4 and 8.5 would differ from the values stored in memory.

Another advantage of this hashing scheme compared to the one published in [72] is the performance gain by parallelizing hash-value computation. The old scheme was based on a sequential approach requiring five sequential AES applications before the final hash value has been computed. The new scheme reduces this to roughly two sequential applications, resulting in a performance gain of 5/2.

The secret random value $r$ can be used to further optimize integrity checks. The knowledge of $r$ is required to compute valid hash values. Due to the one-way properties of the hashing function, it is not possible to compute $r$ based on the unencrypted plaintext and the corresponding hash value. A brute force attack would be possible, but due to the size of $r$ (512 bits) it is not feasible. Without the knowledge of $r$ it is impossible to compute a valid hash value for a given cache line, even if the line is not encrypted. This property can be used to omit a full hash tree walk when verifying static data like instructions, because replay attacks on static data are not possible as long as $r$ is different for different programs. When using the same $r$ value for different programs, an attacker could replace protected memory

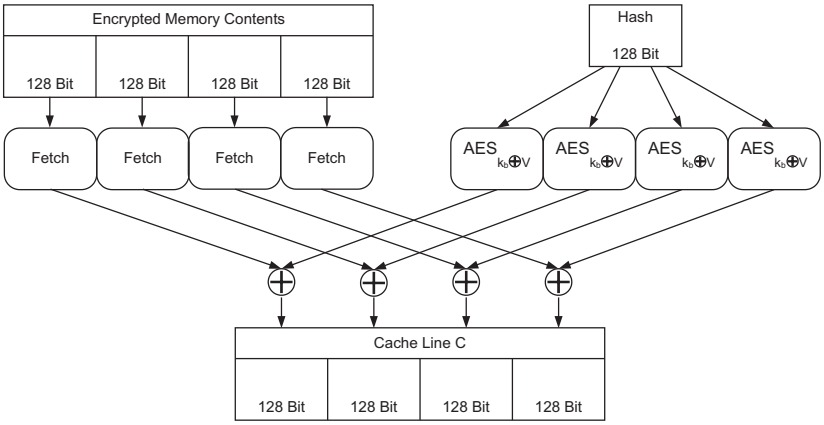



Figure 8.2: Memory decryption

contents from one program with contents located at the same address of the other program. Admittedly, omitting a full hash walk is only possible when executing no self-modifying code. Furthermore, in contrast to the first design of *SAM* [72], it is no longer required to store the root hash encrypted within the executable. This does not lower the security, because an adversary cannot compute valid hashes, and the processor prevents read access to already stored and updated root hash values.

## 8.2 Memory Encryption

*SAM* encrypts all data located in encrypted memory areas transparently before storing it in memory. Since encryption adds another latency to each memory access, special care has been taken to minimize the overhead. Instead of encrypting the memory contents directly – which would sum up the fetching and encryption latency –, a parallel approach based on the counter mode [20] is used. In contrast to the AEGIS architecture a dedicated counter value is not used. Instead, the corresponding 128-bit hash value (see section 8.1) is used as a counter value. This has the following advantages:

- The memory footprint is reduced, since there is no need for an additional counter value.
- The hash value is already stored in the cache when the cache verifies the integrity. This helps to speed up integrity verification.
- A full re-encryption of all encrypted memory contents on counter overflows is not required.
- For direct encryption the latency to fetch the cache line $C$ and the latency to decrypt it sum up. This is not the case for counter mode, since the encryption function is applied to the counter and not to the actual data. Hence, both steps can be processed in parallel if the counter value is known in advance, as shown in figure 8.2. Therefore, both latencies do not sum up if a reasonable fast AES implementation, such as [34], is used. The probability of a hash line already present in the cache is between 0.5 and 0.9 for the simulated workloads presented in section 14.6. This is due to the fact, that most memory accesses are local, both in space and time. Hence, an already fetched hash

line can be used to decrypt up to three newly fetched cache lines, since each hash line protects four cache lines. In cases where this is not possible, the latency is around twice the latency of an unencrypted fetch, similar to the latency of direct encryption. Hence, using counter mode is typically faster than direct encryption schemes.

- Approaches like memory pre-decryption [78] can hide the latency only in cases of successful predictions but result in much higher latencies in case of mispredictions. It is therefore advisable to use cryptographic schemes which provide a good overall performance in all cases regardless of predicted memory accesses. However, when using the presented design it is possible to use memory access predictors as well to further speed up memory access.

Indeed, as described in section 4.5.5, special care has to be taken for the counter value in counter mode. Equal counter values for different data to encrypt have to be prevented. Since *SAM* uses 128-bit hash values, the probability of equal hashes is very low, because the value $r$ is unknown to an adversary and therefore he is is not able to compute valid hash values by himself. Instead, he has to wait until two hashes are equal. According to the birthday paradox described in section 4.2, this may happen after approximately $2^{\frac{128}{2}} = 2^{64}$ hash-value computations with a probability of 0.5.
The only information that cannot be hidden with this scheme is that data at a given address is unchanged compared to a previous point in time. However, in cases where the programmer wants to hide this information as well, he or the compiler could enclose the data to protect with random values changing their values with each write access.
To further decrease the impact of equal hashes *SAM* uses virtual address-dependent encryption keys based on a base encryption key $k_b$ chosen at compile time. The encryption will be performed now at the block level, using a different key to encrypt each block; this encryption key will be derived from the virtual address of the block to be encrypted, using $k_b$ as a parameter. This "derived key" will be the actual encryption key for the block to encrypt. The usage of a base key has the desirable property that only hash values for the same virtual address have to be unique. This further reduces the probability that two equal hash values computed over two different cache lines $C_1$ and $C_2$ can be exploited to simplify their decryption.
More precisely, let $\mathcal{K}$ be the space of encryption keys, let $\mathcal{K}^\star$ be the space of base keys, and let $\mathcal{V}$ be the space of virtual addresses; then, given a block $X$, which belongs to cache line $C$ and is located at virtual address $V$, the encryption function is

$$E_{g_{k_b}(V)}(H(C)) \oplus X. \tag{8.6}$$

In this expression, $g\colon \mathcal{K}^\star \times \mathcal{V} \to \mathcal{K}$ is a suitable transformation function, such that, for each value of the parameter $k_b \in K^\star$, $g_{k_b}(V)$ supplies a usable AES encryption key. This function should satisfy the following property: for any $k_b$, $k'_b \in \mathcal{K}^\star$, there do not exist $V$, $V' \in \mathcal{V}$, $V \neq V'$, satisfying $g_{k_b}(V) = g_{k'_b}(V')$. In practice, such function exists since $|\mathcal{V}| \ll |\mathcal{K}|$, but then, of course, the base key space is smaller than the original, namely $|\mathcal{K}^\star| < |\mathcal{K}|$. Initially, the requirement of hardware simplicity compels us to use a simple $g$, such as XORing $k_b$ and $V$. But, then, the system could be liable to a related-key attack, as described in section 4.5.3.

# Chapter 9

# *SAM* Implementations

Most parts of the *SAM* architecture as described in the previous chapters are independent of the underlying processor architecture. This chapter describes architecture-related adaptation of *SAM* for the SPARC and the IA-32 architecture. Since the main changes compared to a standard processor are located in the cache, the modifications to a standard processor can be kept small.

## 9.1 *SAM* for SPARC

This section provides a short overview of the SPARC V8 architecture based on the descriptions in [67, 85]. A SPARC processor provides 32 logical registers, separated in four different register types containing eight registers each: `global` (*%g0-%g7*)[1], `local` (*%l0-%l7*), `in` (*%i0-%i7*) and `out` (*%o0-%o7*). Internally they are mapped to up to 520 physical registers in up to 32 sets. As can be seen in figure 9.1, the `in` and `out` registers of side-by-side sets are overlapping to allow parameter passing.
The register mapping is controlled by the Current Window Pointer (CWP). When executing a *save* instruction or during a TRAP the window pointer is decremented and incremented by *restore* and *rett* (return from trap) instructions. A CWP over- or underflow results in a TRAP. The TRAP then has to read or write the current register set from or to the stack.
Even though all registers are general-purpose registers from the hardware point of view some registers contain special values by convention. Examples are the stack pointer which is always stored in the `out` register 6 (*%o6*), the frame pointer in the `in` register 6 (*%i6*) and the return address in register *%o7*. This design automatically converts the current stack pointer to a frame pointer when the register window is changed on subroutine calls.
The SPARC architecture provides a user mode and a supervisor mode. The supervisor mode is entered either by software when executing a *trap* instruction or by a hardware TRAP.
The SPARC architecture provides 256 TRAP's. On each TRAP a branch into the TRAP table is performed and the program counters of the interrupted instructions are saved in local registers. Other operations to save the current state of the interrupted process have to be performed by the operating system. At the end of the TRAP function, the OS returns to the user program by executing a *rett* instruction. Due to register window design, the TRAP handler can access its own local registers. Therefore, saving registers to the stack is not

[1] The first global register always contains the value zero.

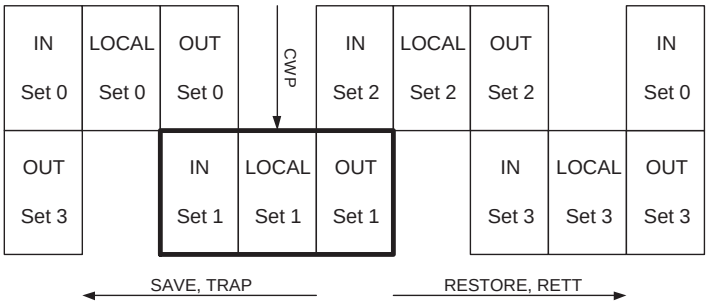



Figure 9.1: SPARC Register Windows

necessary in cases where the number of registers required by the TRAP handler does not exceed the number of available local registers.
The memory management unit (MMU) allows mapping between a program-dependent memory layout (context) and the physical memory available. The MMU distinguishes programs by a unique process ID (context ID) and can only be configured in supervisor mode.
Like on all RISC architectures the SPARC instruction set is very regular. There are only three different instruction formats, *Format 1*, *Format 2* and *Format 3* [85]. They are distinguished by a small two-bit opcode which allows the only *Format 1* instruction, the *call* instruction, to jump to arbitrary addresses in the 32 Bit virtual address space. *Format 2* instructions are mainly used for branches, and *Format 3* is used for all remaining instructions. The latter can address either two registers and a 13 bit immediate constant or three registers with an optional address space identifier (ASI). The ASI can be used by load and store instructions to select a different address space which can be used to directly access hardware like the MMU.
The current processor status is stored in the Processor Status Register (PSR). It contains among others the integer condition codes N, Z, V, and C[2], the `S` flag which indicates supervisor mode, the `PS` flag which stores the `S` value before a TRAP, and the CWP.

### 9.1.1 Register Protection

In addition to the *SAM* architecture presented in chapter 7, the SPARC implementation of *SAM* provides a register protection mechanism. It is not strictly necessary to provide a secure execution environment, since registers containing sensitive values can always be saved in protected memory and then cleared before executing unprotected instructions. But due to the large SPARC register set, clearing all registers containing sensitive data would degrade performance and, for this reason, register protection has been added to the *SAM* architecture.
The current register protection scheme of *SAM* has been modified compared to the protection scheme described in [72] in order to reduce the hardware overhead. Instead of storing all eight bits of the context number of the writing context along with each register, now only one additional register protection flag is required for each register. This modification saves up to seven bits for each physical register. The flag is set if the corresponding register has been written by a protected instruction and cleared otherwise, i.e., it is set according to the `PI` value (all processor flags are listed in table 7.1) of the writing instruction. On each read access the `PI` flag of the reading instruction is compared with the stored flag. If both are equal the read succeeds. If they are not equal, the processor has to distinguish the following two cases:

[2]N: negative, Z: zero, V: overflow, C: carry.

1. If PI is set, the tamper detection unit is informed to terminate the current protected process.

2. An unprotected read to a protected register succeeds, but instead of the current register value zero is read to prevent information leakage. An implementation which always treats different values in PI and the register protection flag as a tampering attempt is possible, too, but the current design has been chosen to stay compatible with the design described in [72]. This design had been chosen to stay compatible with memory protection, which allows reading of encrypted parts of memory contents by unprotected instructions as well (as long as the unencrypted parts are not visible).

Write access is always possible and sets the register protection flag accordingly. With only one flag per register different protected processes cannot be distinguished any longer. Hence, the processor has to clear all protected registers on context switches, as described below in section 9.1.3.
In some cases register values must be shared between protected and unprotected instructions, for example, to allow parameter passing between the unprotected part of the OS and a protected program. Therefore, register protection can be partly deactivated using a new instruction. This instruction allows to alter the register protection flag of all registers in the active register window and is described in the next section.
Register protection is not limited to general-purpose registers, because other special registers like the PSR can be used to pass small portions of information from the protected part to the unprotected part. Hence, *SAM* does not execute instructions accessing special registers speculatively; these instructions are stalled until PV equals one. Since the protected part of the operating system is small and these special registers can only be accessed in supervisor mode, the performance loss is minimal.

### 9.1.2 Instruction Set

The SPARC instruction set has been enhanced by only three additional instructions. Hence, the modifications to the instruction set are much smaller than for other security architectures like AEGIS. The existing SPARC instruction set has been designed with a pipeline implementation in mind. Therefore, pipeline-design related features are discussed when describing the new instructions as well.
All of the following new instructions are *Format 3* instructions:

1. *st imm* (secure TRAP). With this instruction the PT flag is set based on the least significant bit of the immediate field *imm*. The instruction can be executed only in supervisor mode, has to be located in protected memory, cannot be executed speculatively and has no operation for unprotected programs (PP unset), as can be seen in algorithm 9.1.

   The modification of the PT flag directly affects the next executed instruction if it is not located in encrypted memory, because due to the pipelined design the next instructions[3] are already loaded but not executed when the flag is altered. This design has been chosen, because in supervisor mode all instructions are unencrypted and therefore this behavior is no limitation and possible pipeline stalls would decrease execution performance. As described later in section 11.3 it is assumed here that all instructions

[3]The exact number is implementation-specific.

located in protected, but unencrypted memory are verified regardless of the value of `PI` to prevent memory tampering attacks.

**Algorithm 9.1**: Implementation of *st*

```
if PP==1 then
    if imm==0 then
        if ST==1 and PI==1 then
            while PV==0 do
                ;
            end
            ST=0;
        else
            security_violation();
        end
    else
        if ST==0 and PM==1 then
            while PV==0 do
                ;
            end
            ST=1;
        end
        security_violation();
    end
end
```

2. *rprot %rs1, %rs2, %rd* or *rprot imm, %rd* (register protection). This instruction controls the register protection flags for the current register window when the `PI` processor flag is set, as can be seen in algorithm 9.2. Since this instruction can be used to unprotect parts of the register contents with possibly sensitive data, *SAM* requires that the `PV` flag is set before further processing the instruction. The instruction can either be used with an immediate field or with three registers:

   - When used as immediate instruction the immediate bits have the meaning as described in table 9.1.
   - The variant with three registers uses register *%r1* as a mask to select the registers of the current window to be set with the corresponding values given in register *%r2*.

   In both cases the destination register *%rd* contains the flags for all registers of the current set. This instruction definition has been used to be able to alter register protection flags with one instruction. In almost all cases only the flags of the `out` registers of the current set have to be altered and this is possible using only one instruction – the *rprot* instruction – with the immediate field set accordingly. Therefore, the additional overhead is kept very low.

| Bit | Description |
|---|---|
| 12 | The new value for all non-masked registers. |
| 11 | When set, the flags of the seven `global` registers are set based on the mask to the value of bit 12. |
| 10 | Like bit 11, but now the eight `out` registers are set. |
| 9 | Like bit 11, but now the eight `local` registers are set. |
| 8 | Like bit 11, but now the eight `in` registers are set. |
| 7-0 | Mask to select the registers to update. Each bit in this mask corresponds with one register of the selected registers. |

Table 9.1: *st* immediate constant – description

**Algorithm 9.2**: Implementation of *rprot*

```
if PP==1 then
    if PI==1 then
        while PV==0 do
            ;
        end
        if instruction_format==immediate then
            rd=set_registerprotection_flags(imm);
        else
            rd=set_registerprotection_flags(%rs1, %rs2);
        end
    else
        security_violation();
    end
end
```

To further optimize execution of *rprot* it is implemented with a delay slot. That is, the next instruction following *rprot* does not change any register protection flag, even if it writes to a register. Instead, the flags set by *rprot* are used. Using this delay slot the protected part can be left with the following instructions in the given order:

```
// disable register protection
rprot (1 << 11 | 1 << 10 | 0xff), %g0;
// call subroutine and write return address to %o7
jmpl %o3, %o7;
// disable secure TRAP mode (called in branch delay slot)
ste;
```

Due to the delay slot the return address stored in register `%o7` is not protected and readable by unprotected instructions. Note that the *ste* instruction is executed in the branch delay slot of the *jmpl*[4], thus disabling the protected TRAP mode for the jump target.

When returning the following instructions should be executed in the given order:

```
// enable secure TRAP mode
sts;
// enable register protection
rprot (1 << 12 | 1 << 11 | 1 << 10 | 0xff), %g0;
// no operation in delay slot
nop;
```

The *nop* operation could be replaced by any operation not modifying the register values like *copybits*.

3. *copybits*. This instruction copies the current contents of the `PP`, `PT`, `PM`, and `PV` into the integer condition codes N, Z, V, and C for protected and unprotected processes as shown in algorithm 9.3. Therefore, this instruction provides an efficient way to branch based on the *SAM* flags, for example to skip code which should only be executed in case of protected programs. This instruction is not strictly necessary, because these values can be mapped to unused bits in the PSR as well. But then one additional instruction after reading the PSR to mask the desired bit would be required.

**Algorithm 9.3**: Implementation of *copybits*

```
N=PP;
Z=PT;
V=PM;
C=PV;
```

More instructions, for example to pass the encrypted base key $k_b$ to the processor or to set the process-specific base addresses, are not required due to SPARC's support of alternate address spaces. All these functions have been mapped into a previously unused address space. Using a memory-mapped interface often enlarges the overhead when compared with dedicated

[4] Jump to given address and adjust CWP.

instructions, but since this interface is only used on protected process creation and termination and in case of errors reported by the tamper detection unit, this overhead is acceptable. For example, the main delay before starting a protected process is caused by the decryption of $k_b$ and $r$. Furthermore, the number of unused opcodes for existing RISC architectures is rather low, which complicates the implementation of more functionality in instructions.
In addition to the new instructions, the two existing SPARC instructions *wrpsr*[5] and *rett* have been slightly modified. Both instructions can alter the processor's `S` flag. To prevent unintended transitions to user mode the `S` flag cannot be cleared when the following condition is met (see also table 7.2): $\texttt{S} \wedge \texttt{PP} \wedge \overline{\texttt{PI}}$. Each prevented clearing attempt results in a security violation reported to the tamper detection unit.
This modification ensures that for protected processes user mode can only be entered from protected instructions in supervisor mode. Other modifications to existing instructions are not required.

### 9.1.3 Context Switches

On SPARC processors the context is switched when writing a new context number to an MMU-specific register using a store instruction with an alternative address space. For *SAM* this context switch is reported to the processor by the MMU. The processor then stalls further execution until the cache sets the `PV` flag and clears all protected registers in all windows. Therefore, a context switch even to another protected process can be performed at any time in supervisor mode, a protected instruction is not required because all possibly remaining protected data is cleared.

### 9.1.4 TRAP Modifications

*SAM* adds the following two TRAP sources to the processor:

1. The tamper detection unit performs a TRAP for each security violation to inform the operating system that a tampering attempt has been detected. This is required because after each tampering attempt the corresponding program cannot be executed any longer and the operating system needs to be notified.

2. The RSA unit notifies the processor each time a key has been decrypted. This design has been chosen to prevent time-consuming polling when a new protected program is to be started.

In addition to the new TRAP sources, the main modification to the TRAP handling is the support of the secure TRAP mode. The processor now automatically sets the `ST` bit on each TRAP for protected processes. Each time the `ST` flag is already set on a TRAP or cleared during *rett* for protected processes a security violation is reported. Furthermore, the processor now stalls until the cache sets the `PV` flag on each TRAP.

### 9.1.5 Memory-Mapped Configuration Registers

Table 9.2 lists the most important data assigned to the alternative address space used for *SAM* configuration. The access column shows whether the corresponding value can be read

[5] Write to PSR register.

| Content | Access | Description |
|---|---|---|
| Control Register | R/W | Used to clear interrupt request, committing changes written to the context, base and secret key registers and other flags. |
| Error | R | Contains error information in case of failures. |
| Public Key | R | Contains the processor's public key information including the size of the key. |
| Context Number | R/W | The context number the following data is used for. |
| HASH_BASE | R/W | The address of the root hash (cf. figure 7.2). |
| ENC_BASE | R/W | Base address for encrypted memory. |
| PROT_BASE | R/W | Base address for protected memory. |
| UNSEC_BASE | R/W | Start of unprotected memory. |
| Secret Key | W | Used to store the encrypted secret key $k_b$ and $r$ before decryption. |

Table 9.2: *SAM* configuration address space

(R) or written (W). The configuration address space is mainly used by the operating system to create or terminate protected processes. Additionally, it provides information about detected security violations.
The configuration space is used by the operating system as follows: At program start-up, the operating system writes the context number, the base addresses and the secret key to the address space. Then, another write access to the control register is used to inform the hardware to take over all values to create the new protected process and to decrypt $k_b$ and $r$.

### 9.1.6 Cache

All changes described above are relatively small compared to the changes to the cache system. The cache design is described in chapter 11 because it is mostly processor-independent and applies to both SPARC and IA-32 processor implementations.

### 9.1.7 Further Changes

With its alternate address spaces, the SPARC instruction set provides different means to access memory. Since these instructions could be used to leak data in cases where an adversary is able to modify protected instructions before the cache detects them, *SAM* stalls execution of these instructions until all fetched cache lines have been verified. The same applies for all protected instructions executed in supervisor mode used to access IO-ports. Since the protected part of the kernel is small and the same code is executed frequently, the performance loss is small, because data is already present and verified in most cases.
To prevent any information disclosure the processor clears several processor flags like the C, Z, N, and O flags after detecting a tampering attempt. This prevents these bits from being used to leak data.

## 9.2 *SAM* for IA-32 Processors

The IA-32 *SAM* design is based on the SPARC design, since *SAM* has been first implemented exclusively for SPARC processors. This section gives an overview of the processor-related

changes required to adapt *SAM* to IA-32 processors. Due to the huge complexity of the IA-32 architecture, no attempts have been made to implement the proposed changes. Therefore, only the processor-related modifications are described here.
The IA-32 architecture has been developed by Intel and the current instruction format, and an architectural description can be found in Intel's Software Development Manuals [10, 11, 12, 13, 14]. In the following, the relevant parts for the *SAM* architecture are discussed.
Intel has developed the IA-32 architecture as an CISC architecture. The instruction format is complex and provides several operand-addressing modes, both to registers and directly to memory. The processor supports two memory-addressing modes, real mode and protected mode. Since only the protected mode provides virtual memory and a 32 bit virtual address space, it is the only mode considered for *SAM*.
The processor has eight general-purpose registers, but two of them are typically used for stack operations. The architecture compensates its small register set with the ability of most instructions to directly use memory contents as operands. The other registers are special registers such as memory management, interrupt registers and control registers.
The IA-32 architecture provides four privilege levels, called "rings", but most modern operating systems use only the lowest (ring 3) and highest (ring 0) level which in the following are referred to as user mode and supervisor mode. Privilege levels are changed either by instructions or by interrupts.
On each interrupt the processor locates the interrupt handler by indexing the Interrupt Descriptor Table (IDT) stored in memory at the address stored in the Interrupt Descriptor Table Register (IDTR) with the current interrupt number. On each interrupt the processor automatically stores the values of some registers like the stack pointer (ESP) and the address of the interrupted instruction (Instruction Pointer, EIP) on the stack assigned to the interrupt handler. The interrupt handler is left by the *iret* instruction, which restores the interrupted state with the previously stored values.
In the following, the required changes for an IA-32 *SAM* design are described. Note that the proposed design is not considered as a general design covering all IA-32 specifics, because IA-32 is very complex and provides partly redundant addressing modes, privilege levels, and task descriptors which are not widely used but would have to be supported for a full design. Instead, the parts of the IA-32 architecture which are used by Linux have been protected to allow a *SAM*-enabled Linux kernel as described in chapter 10.

### 9.2.1 Register Protection

Compared to the SPARC architecture no register protection is required for the IA-32 port of the *SAM* architecture due to the small number of general-purpose registers. Even explicit overwriting of these registers to hide their contents is not required in most cases, because the protected part can only be left in supervisor mode and most registers containing sensitive values have already been used and overwritten at this point.

### 9.2.2 Context Handling and Configuration

The IA-32 architecture does not contain a context identifier like SPARC does. Since some kind of identifier is required to distinguish unprotected from protected programs and to select the correct base key $k_b$, this information has to be supplied. For this implementation a previously unused control register (CR1) has been used. This has the advantage that no instruction set

| Bit | Access | Description |
|---|---|---|
| 31 | R | PT: Protected Interrupt |
| 30 | R/W | PP: Protected Process |
| 29 | R | PM: Protected Memory |
| 28 | R | PV: Protected Data Valid |
| 27 | R | NI: Nested Interrupt |
| 26 | R/W | PD: Protected Process Deletion |
| 25-20 | R | Error Code |
| 19-4 | R | Bitmap of already allocated protected processes |
| 3-0 | R/W | Context Number of the currently running protected process |

Table 9.3: Contents of CR1

changes are required to access this register, since it can already be addressed by the existing instruction set. This register now controls the basic *SAM* configuration, and its contents are assigned as described in table 9.3. All bits except the bitmap bits are only valid if the PP bit is set. Otherwise they are set to zero. The first four bits in this register are used to store the flags described in table 7.1 above. NI is described later in section 9.2.3 and PD can be used to delete a protected process entry including deletion of the corresponding cache contents and process-related keys. In the bitmap field each already allocated protected process is denoted with its bit set. The total number of protected processes (contexts) with this design is 16. Error codes and additional status information are provided in the Error Code. Here, for example, the processor passes information about the reasons for security violations to the operating system.
The additional information compared to SPARC stored in this control register is required, because IA-32 does not provide alternative address spaces which could be used for memory-mapped configuration registers. This affects key and memory region passing to the processor as well. Therefore, this data is now stored in RAM and directly accessed by the processor. These changes require new instructions not needed for the SPARC architecture. All new instructions are described in section 9.2.4.

### 9.2.3 Privilege Level Transitions

As described above, the IA-32 architecture defines four privilege levels. Since the rings 1 and 2 are typically not used and their usage would complicate the *SAM* design, any usage while executing protected programs is reported as a security violation to the tamper detection unit. On each transition between the two remaining privilege levels, ring 0 and ring 3, the processor has to check and set the PT flag, as described for the SPARC architecture. Furthermore, the NI flag is set in case of a nested interrupt to simplify protected interrupt handling. Each interrupt which interrupts an instruction executed in supervisor mode sets this flag to be able to distinguish between user mode to supervisor mode transitions, and interrupts within supervisor mode.
Unfortunately, due to the limited number of registers, the vital data of the interrupted program like the instruction pointer cannot be stored in registers, therefore it is written to the kernel stack. Since the normal kernel stack is located in kernel memory and therefore unprotected, an adversary could try to read or modify these values.
Thus, the kernel stack for protected programs is moved to protected memory. The stack can

be located anywhere in memory, but it has to be ensured that an attacker is unable to alter the pointer to this region.
For simplicity's sake, it is assumed that the base address $PKS_{base}$ of the protected kernel stack of each protected process is located at the same virtual address. This address can be set by the operating system before starting any protected process, and it is then incorporated into the hash-value computation to prevent tampering attempts. The same applies for the base address of the IDT ($IDT_{base}$) which has to remain static as well and both addresses have to point to protected memory. To be more specific, $PKS_{base}$ and $IDT_{base}$ are concatenated to form $T_{base}$ (see equation 8.1):

$$T_{base} = PKS_{base}||IDT_{base}.$$

A protected kernel stack has the following drawbacks compared to an unprotected stack: At first, the unprotected parts of the operating system are unable to access the protected stack; therefore the operating system has to switch to an unprotected stack located in unprotected memory before clearing the `PT` flag. Second, parameters passed to the kernel have to be copied by the operating system to the unprotected stack as well. The same applies for transitions back to user mode where the stack contents have to be moved back to protected memory and possible return values have to be copied, too.

### 9.2.4 Instruction Set

The new instructions for the IA-32 architecture differ from the ones needed for SPARC. Due to the larger number of addressable but currently unused opcodes, the *st* instruction has been split into an *ste* instruction and an *sts* instruction with their own opcodes.
As described above, the processor reads its memory base addresses and the encrypted base key directly from memory. In contrast to SPARC, reading processor configuration data from memory is a typical operation for IA-32-based processors, since many descriptor tables are stored in memory as well. For *SAM*, a new instruction is used to pass the base address of the configuration values to the processor. The new instruction is called *csee* (Create Secure Execution Environment) can only be executed in supervisor mode and requires one register as operand with the base address. The processor does not provide a public register to store this address (like the IDTR), because once the data has been read and processed by the processor the memory contents can be freed and reused.
More new instructions are not required, so that the instruction set modifications are small as well. Additionally, existing instructions have been slightly modified for *SAM*: all instructions accessing CR1 do not longer generate an "invalid opcode" exception, and instructions writing to unprotected regions or IO ports are stalled until `PV` is set.

## 9.3 Multiprocessor Support

Initially, *SAM* was designed for single processors, but multiprocessor support is not prohibited by the architecture, since all used keys are static for a program, and distribution to several processors is possible since protected memory contents can be accessed by more than one processor. But due to the internal caches of processors, accessing the same memory cache coherency has to be ensured. There are cache coherency protocols like MESI [33] to guarantee coherency, but all coherency protocols require that messages be exchanged between all processors to inform other processors about read or write access to memory regions. However, *SAM*

does not provide secure message passing between processors, and altering the consistency of processor caches could be used by an adversary to alter a protected program. Fortunately, this kind of attack is impossible when placing multiple processor cores on one die in one shared housing like used for SUN's new open source Niagara design (OpenSPARC T1[6]). Then, the whole multiprocessor core can be protected against external attacks and coherency can be guaranteed as long as only one multi-core processor is used.
Cache coherency is not the only requirement for multi processor support for *SAM*. As long as only one single-threaded process is involved, its execution can easily be switched between all cores. But with multiple processors it is desirable to execute a protected process in a multi-threaded fashion. The proposed processor implementations for SPARC and IA-32 processors support multi processing operations differently. For IA-32 the proposed design would have to be modified due to the static kernel stack. This does not apply to SPARC, which can support multi-threaded protected programs. However, the operating system has to be changed to ensure that multiple concurrently running threads do not allow more attacks, particularly in terms of locking and synchronization. The operating system design proposed in chapter 10 does not cover multi processing support and therefore a proper implementation is left as a future issue.

[6] `http://www.opensparc.net`

# Chapter 10

# *SAM* Operating System Design

This chapter describes a sample SPARC Linux 2.6.13 operating system design based on the design published in [71]. *SAM* does not require Linux as an operating system kernel, other kernels can be adjusted to support *SAM* as well. Linux has been chosen because the source code is freely available and widely used on server and desktop environments. Due to new instructions, cryptographic units, and memory layout, software modifications in the operating system and the executed program are required. The design presented in this thesis has been developed to protect the executed program but not the target system from malicious programs. Therefore, it is advisable to make use of additional security extensions like SELinux to protect the target host.

## 10.1 Threats

The *SAM* architecture provides hardware-based protections against external attacks, such as memory modifications and bus sniffing. Another class of attacks are software-based attacks originated by the operating system or by the administrator or the computer system. As described in section 2.2, the operating system is involved in many tasks and can harm the program in many ways. *SAM's* register and memory protection mechanisms can be used to protect the currently running process only. However, on a multitasking operating system several processes are executed in parallel and the operating system has to ensure secure ways to suspend execution of one protected process and resume execution of another protected or unprotected process. In the meantime, any modification to the suspended process has to be prevented.
Therefore, the following threats have to be considered by the secure operating system design:

- *Program counter manipulations*: The kernel stores the program counters (PC and NPC) in memory on each TRAP in order to be able to return to the interrupted instruction. A malicious kernel can modify these values to alter the program flow.
- *Signal Handling*: Other programs and the operating system can send signals to a program for notification purposes. These signals have to be passed in a secure way to prevent manipulations. However, since signals can be sent at any time, the programmer of the application has to handle them manually or ignore them.
- *Register manipulations*: Register values can be securely stored on the stack of the protected process to prevent tampering attempts. However, registers are only stored on the

stack on context switches or register window overflows. In cases where the protected part of the kernel is left without changing the context there is no need to store the register values and to restore them later. Due to the register window design, any tampering attempt to the current active register mapping has to be prevented as well. For example, the unprotected part of the operating system could be able to switch to another register window before entering the protected part. If this window has matching permissions, i.e. all registers are marked as protected, which is always the case for windows belonging to higher recursion levels of protected processes, can the attack remain undetected for a longer period of time, because the register protection flags are only able to distinguish between protected and unprotected register access but not between the correct and a wrong register set.

- *Forging of memory contents*: There are several places where the kernel is able to forge memory contents. For example:
  - when passing return values to the user program after a system call,
  - when accessing external devices,
  - when setting up the environment variables and command line parameters,
  - by manipulating the page mapping.
- *Manipulating system call return values*: Most system calls return values either in registers or in memory. These values can be tampered with by the operating system to manipulate the program.
- *Tracking the accessed memory area*: The operating system can record all accessed pages using a manipulated page table by forcing a page fault TRAP for each executed instruction. This information can then be used for statistical analysis as described in section 5.1.3.

## 10.2 Protected Kernel

Along with some changes in the hardware, *SAM* requires that a (small) part of the operating system be to be protected and checked automatically by hardware. This part resides in protected memory and, therefore, is protected by hash values, to be provided by the protected program. This design is fundamentally different from other protected operating system designs (presented in chapter 5), because here, the protected operating system is not protected in advance (for example by a protected boot loader), but only when executing protected processes and only for the duration of the protected process. This design has been chosen to limit the negative performance impact of protected program code to periods where protected processes are executed.
The protected part has a size of roughly a mere 64 kByte and is located directly over the user stack, at the beginning of the kernel virtual address space. Its main purpose is to ensure a secure transition between the protected user mode and the (mostly) unprotected kernel mode. Using this design, the major part of the operating system kernel, including drivers, can remain unprotected. As a side effect, kernel upgrades, such as bug fixes or additional drivers, are possible as long as they are not affecting the protected part.

Besides these hash values, each protected program has to provide one additional page located in virtual user space directly before the protected kernel. Initially, this page mostly consists of zeros and is used as a *protected compartment* for the corresponding program by the kernel. All data written to this page is protected by *SAM*, as it belongs to the protected and encrypted area, but it is used by the protected part of the kernel only. For this reason, the kernel must ensure that this page cannot be paged out and is not writable by the user process. The *SAM Linker* automatically creates this page before encrypting the program and initializes it with the entry point of the program. Then, this page is encrypted along with the rest of the program.

### 10.2.1 Limitations

Protected programs behave mostly like normal unprotected programs. However, any debugging support provided by the kernel has been prevented for protected programs. Therefore, the usage of the break-point handlers along with other debug functions result in an operating system generated security violation.

### 10.2.2 Loading Protected Programs

Each time a protected executable is to be started the kernel performs the following steps:

1. Identifying the executable by reading the target architecture. If the executable is not protected, then it is loaded in the same manner as by an unmodified kernel. The following steps are performed only if the executable is protected.

2. Assigning an unused secure context number to the new program. If there are no secure context numbers left, the kernel aborts loading the program.

3. Reading the base addresses for the different memory areas from the program and setting them for the selected secure context number using the memory-mapped configuration registers.

4. Reading the encrypted program-dependent secret key $k_b$ and random value $r$ from the executable and passing it to the RSA unit for decryption.

5. Loading the protected program like any other unprotected program by creating the page mappings. At this time, the required parts of the sparse hash tree and the protected *compartment* are mapped to memory. This is done by the ELF loader in the Linux kernel.

6. Waiting until the RSA unit has finished decryption. The operating system can schedule other processes during this time. The RSA unit raises an interrupt after successful decryption to notify the operating system, and the cache automatically loads the root hash.

7. Switching to the new secure context to execute a protected initialization routine, which is used to check and copy command line parameters and environment variables to the protected stack. This routine has to be located in protected memory, because these parameters are stored in encrypted memory. Copying data from unprotected memory to protected memory always contains the risk that tampered data is copied. Therefore,

the protected routine applies several consistency checks like ensuring valid pointers and zero terminated strings before actually copying the data. Furthermore, the program's entry point stored in the protected compartment is used to initialize the kernel structures located in protected memory used to store the program and next program counters. To ensure that this function is to be executed exactly one time for each protected program this function uses a flag in the protected compartment (as described in section 10.2.5).

8. Scheduling the new program. The first executed instruction is located at the program entry point stored at program linking time.

These steps ensure that the kernel can call the protected initialization routine only one time and the program is started at the intended position.

### 10.2.3 Hash Tree Handling

As described above, the hash tree is mapped to memory like all other program-related data. However, the operating system has to be adjusted to support an efficient hash tree handling. The first modification is required due to the sparse hash tree design. Typically, each time memory not mapped to physical memory is accessed a page fault interrupt is raised by the processor hardware. In case of the hash tree this mechanism cannot be used to map a zeroed page to the corresponding virtual address space in case of a page fault generated by the cache. The reason for this is *SAM's* interrupt handling, which requires that the cache has successfully verified all protected data until the interrupt handling continues. This requirement can obviously not be met when a page containing hash values is missing. Therefore, as described in section 7.8.2, the program is terminated in those cases. In order to prevent terminations of protected programs based on missing pages, the operating system has to load and map all pages containing hash values belonging to a newly mapped page containing program data into memory as well. Pages containing zeros only have to be mapped to physical memory as well if they belong to other pages already mapped to memory.

The second modification affects the paging routines as well. Since *SAM* uses virtual addresses for all hashes, their physical location can change. Hence, hash values and program-related data can easily be paged out by the operating system. However, the operating system has to ensure that pages are only paged out starting from the leafs of the hash tree. That is, a page containing hash value can at the earliest been paged out when all pages protected by these hashes have been paged out as well. Pages containing user data like the heap or the stack of the program do not need special treatment, they can be paged out at any time and are mapped to memory using the normal page fault mechanisms. The operating system implementation uses an additional counter in the page description structure of Linux. When loading a page the counters of all parent pages containing hash values for the current page are incremented. The counters are decremented each time a child page is removed where removing pages is possible only if the counter value is zero.

In order to prevent verification errors all page handling routines of the operating system have to be located in unprotected memory to prevent unintended memory verifications when loading these values from the hard disk. Otherwise, the cache would start to verify possibly not fully copied pages in cases where no DMA (Direct Memory Access) mechanisms can be used resulting in verification errors.

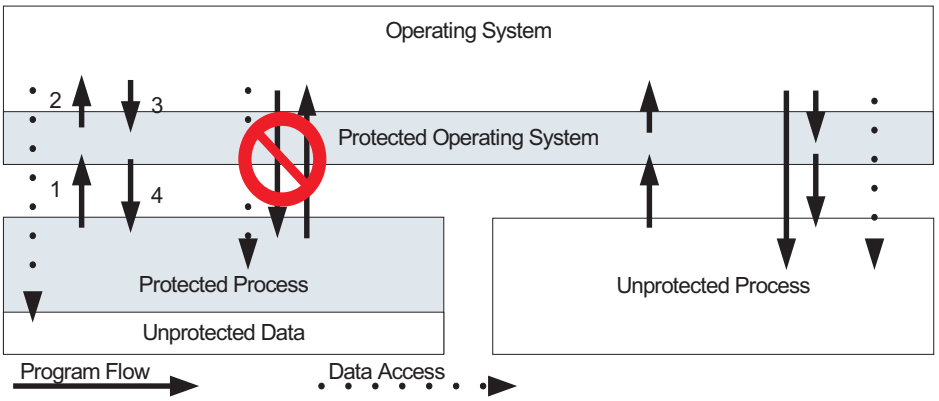



Figure 10.1: Transitions

### 10.2.4 User-Supervisor Mode Transitions

As described in the last subsections, the operating system is involved in TRAP handling. A TRAP for a protected program can be divided into the following transitions (as shown in figure 10.1):

1. *User space to kernel space*: This transition requires the TRAP table to be located in protected memory. As a result, the executed kernel code is protected and has full access to registers and memory contents.

2. *Protected kernel to unprotected kernel instructions*: This transition is preluded with the execution of the *ste* instruction to clear the `PT` flag. Hence, this instruction is to be called before leaving the protected area.

3. *Unprotected kernel to protected kernel*: With this transition the protected part can be reentered by executing the *sts* instruction again, thus setting the `PT` flag. Therefore, *sts* marks all valid entry points and prevents entering the protected kernel parts on other paths.

4. *Kernel space to user space*: This transition is performed by executing the *rett* instruction. Since user space can be entered only by protected instructions the unprotected part of the kernel is not able to enter user space.

These transitions can prevent external manipulations only if the state of the program cannot be modified between the second and the third transition and no other than the listed transitions are possible. To prevent these modifications, all sensitive data is stored in protected and encrypted memory. This data consists, at least, of the program counters (PC and NPC) of the next instructions to execute and the processor state registers.
All register values are stored on the stack located in protected and encrypted memory, too. In this way, *SAM* is able to prevent modifications of any of these values while the program is interrupted and unprotected data is executed.
As stated before, when the kernel has finished its tasks, the scheduler selects the next process to execute. If the selected candidate is other than the interrupted process, a context switch is performed. Before switching the context all remaining register values are stored on the

program stack because the context switch deletes all remaining protected registers and clears their protection bits. If the newly selected process is protected, the kernel performs the third transition in order to restore the program state and register values.
Another possible attack to alter the execution of protected programs would be a direct execution of code by jumping into the user program or somewhere into a protected kernel area. But this is prevented by the hardware, since protected instructions in supervisor mode without the `PT` flag set have no privileges to access encrypted memory or protected registers. Dropping privileges by clearing the supervisor flag in the PSR or by executing a *rett* instruction with a cleared `PT` flag is also prevented by the processor. Hence, a protected user program can only be resumed by following the protected kernel path.

### 10.2.5 Protected Compartment

The protected compartment mainly has the following purposes:

1. It provides a permanent protected storage used to store information about the state of the corresponding protected process. This status information is used for the following purposes:
   - Prevention of re-initialization of a protected process. As described above, the function to copy command line parameters has to reside in protected memory and a second call to this function could be used to attack the program by overwriting the beginning of the stack. Therefore, the `INIT` flag is used to indicate a call to these functions. Table 10.1 provides an overview of the most important flags used by the kernel.
   - Monitored transition handling. The kernel provides several exit points from where the protected part can be left and several entry points to re-enter it. To ensure that the kernel enters the protected part using a function that "matches" the exit point the kernel sets a flag for each allowed entry point before leaving the protected part.
2. The entry point is stored by the *SAM Linker* in this area.
3. The remaining memory could be used as a protected kernel stack if required. However, the current Linux implementation makes no use of a protected kernel stack.

### 10.2.6 System Calls

Since large parts of kernel for the current *SAM* architecture are unprotected, the possibility of forged system call return values remains. This cannot be easily prevented, since the return values of most system calls rely on external and therefore unprotected data, such as file contents or account-specific data. Hence, the protected program cannot trust system calls and must verify the results. This is particularly important for those system calls where the kernel returns memory addresses, like the `mmap` system call. Otherwise, the kernel could return addresses located inside the protected program area, which could result in overwriting program data. These checks can easily be implemented in the modified libc and require no modifications to the user program, thus dramatically simplifying the process of porting existing programs to the *SAM* architecture. They are not part of the kernel to leave the protected kernel space as

| Flag | Description |
|---|---|
| INIT | Used to mark successful initialization of the stack. Set, after setting up the parameters and environment variables on program start-up. This flag prevents a second or no invocation of the initialization routine. |
| PROCESSING | Set directly before the program is started (scheduled) for the first time. This flag prevents that protected kernel functions can be called which would expect a previously interrupted program. |
| TRAP | Set each time before the protected part is left to call a TRAP handler. |
| SYSCALL | This flag indicates that a system call is to be called. |
| ERROR | Set on each security violation detected by the operating system. The program cannot be started with this flag set. |

Table 10.1: Flags set and checked by the kernel

small as possible to lower the probability of erroneous implementations. Additionally, different user space programs may require different checks and therefore these checks belong to the user space.

### 10.2.7 Multi Threading and Signal Handling

The transition model described above works for single-threaded programs and has been successfully implemented. However, many programs are multi-threaded. In Linux, each thread of a multi-threaded program is mapped to its own process and the kernel mostly handles thread management. Since each thread has access to the whole program, the context id and the memory mapping of each thread is the same. Therefore, multi-threaded protected programs are possible, but the protected part of the operating system has to be adjusted to support multiple processes (threads) acting on the same program. This involves enhancing the protected compartment to store additional process-related information and flags to prevent any tampering attempts from the operating system. When running multi-threaded programs on a single processor only one program can be executed at the same time, thus simplifying the protection mechanisms. For multi processor systems many additional thread-synchronization related issues have to be addressed, but they are not part of the current operating system implementation.
The same applies for signal handling. The current implementation lacks support for signal handling, since this was not required to run the benchmarks, but signal support can easily be added by storing the program's signal handlers in conjunction with additional flags in the *protected compartment*.
However, both multi-threading support and signal handling complicate a secure operating system design and a secure application design. With the interaction between several threads the program design becomes more complex and it is much harder for the programmer to write secure programs. Hence, most protected programs do not need multiple execution threads and signal handling.

# Chapter 11

# Cache Architecture

In this section, implementation details of the L1 and L2 cache will be discussed. As encryption and verification takes place in L2 cache, most architectural changes are located there [73]. Data encryption and verification leads to additional latencies. *SAM* was designed to keep read latencies as small as possible. This, however leads to additional delays when protected data is written back to memory. *SAM* tries to hide these additional latencies where possible.

## 11.1 Comparison with other Caches

Both the L1 data and instruction caches are operating on virtual addresses (Virtually Indexed[1], Virtually Tagged[2]). This design has been chosen because it is used by the cache and processor model used as a base for the VHDL implementation (see section 14.1.1). The L2 cache operates on physical addresses but additionally stores the virtual address of each cache line. This is required because the virtual address is used for encryption, for decryption, and to compute the address of the hash value. Due to the cryptographic tasks of the L2 cache, the first-level caches have been implemented using the write-back algorithm to reduce the load on the L2 cache.
Because of the memory verification of the L2 cache parts of its contents are hash values. They are stored in cache lines (here called "hash lines") like other data requested by the processing unit and treated equally regarding the line exchange algorithm. The cache automatically computes the memory address of the corresponding hash value to the requested data and fetches it.
Due to the different memory views, the cache hit logic is more complex than in other caches. For example, a requested cache line already stored in the cache could be treated as miss, if it has been fetched for a different memory view. In cases of a permission-related miss the cache line is written back to main memory and fetched again with the correct permissions.

## 11.2 L1 Data Cache

The L1 cache has to keep track of the memory views of all stored cache lines. *SAM* uses one additional bit for each cache line, the *unprotected* bit, to reflect the memory state of the corresponding line. The cache sets this bit in the following two cases:

[1] The virtual address is used to select the set of a set-associative cache.
[2] The virtual address in conjunction with the context number is used as a tag.

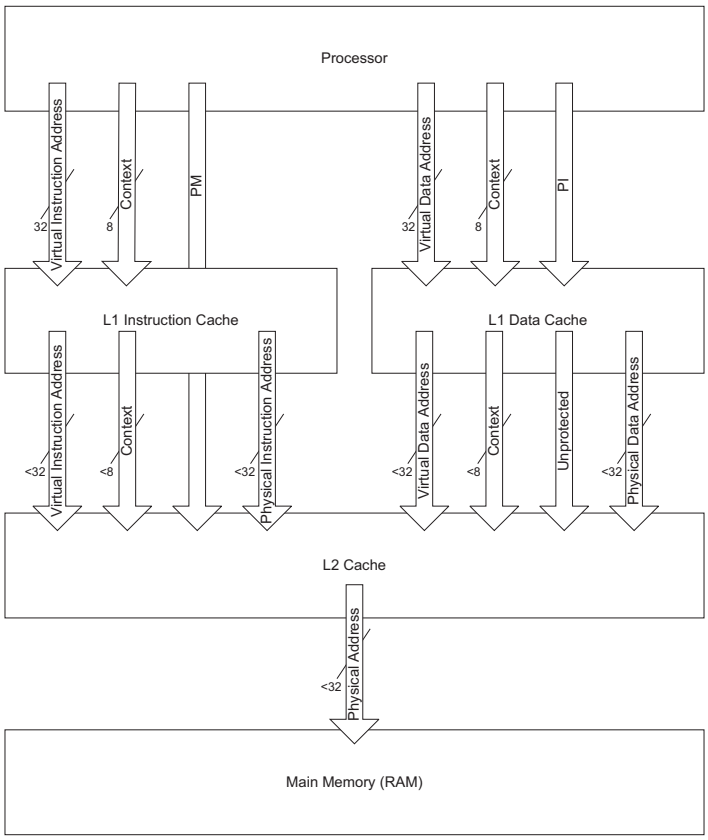



Figure 11.1: Addresses and flags passed to caches

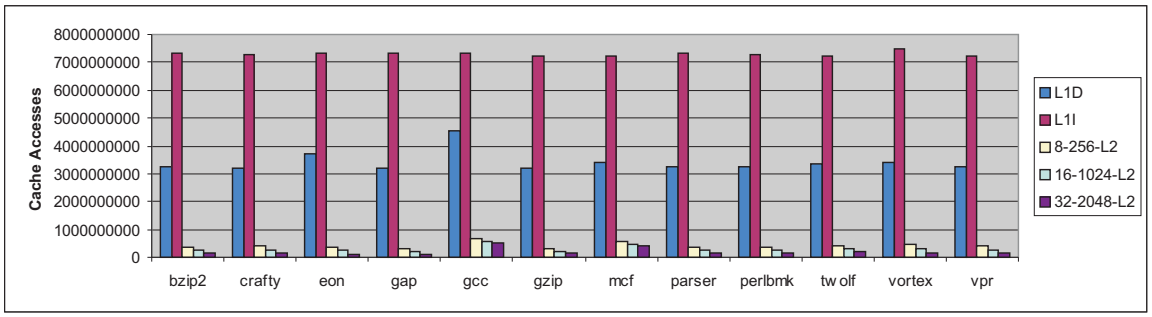
1. For cache lines fetched due to the memory access of an unprotected instruction.
2. For cache lines located in unprotected memory.

Internally, the cache sets the *unprotected* bit to the value of the `PI` flag of the fetching instruction, as can be seen in figure 11.1. This information is additionally passed to the L2 cache during write-back to suppress hash updates. Each time a protected instruction tries to access an unprotected line this access is treated as a miss and the line is written back to the L2 cache and re-fetched. This strategy may result in more misses in case of data being accessed by protected and unprotected instructions, but these cases are rather rare. Furthermore, additional bits to prevent these rare misses would make the L1 cache logic more complex, which would slow down the cache for each memory access. Small performance penalties in the L2 cache are more acceptable, because the L1-data (L1D) and L1-instruction (L1I) caches are much more often accessed than the L2 cache and their miss rate is very low, as can be seen in figure 11.2[3]. The results are computed using the cache simulator described in detail in section 14.2.
Table 11.1 gives an overview of the data stored for each cache line. The column "*SAM*?" states whether the corresponding entry is *SAM*-specific ($\surd$) or already used by a normal cache. It can be seen that the *unprotected* bit is the only addition compared to the normal cache.

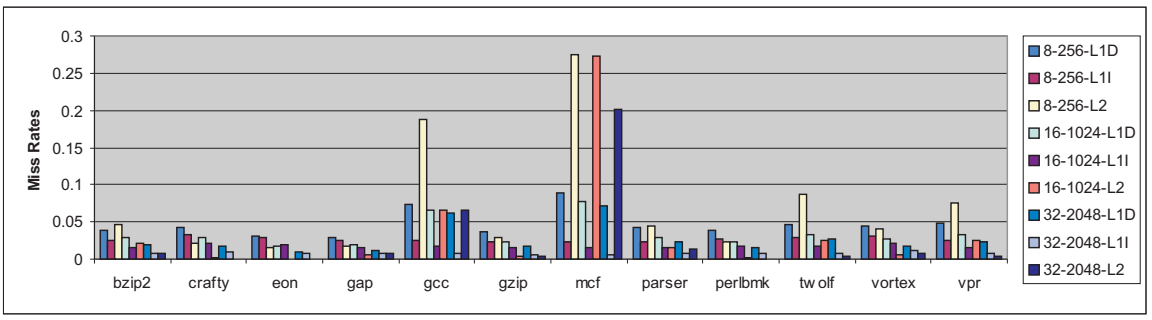
[3]The total number of L1 accesses is equal for all cache configurations and therefore given only once.

| Content | *SAM*? | Description |
|---|---|---|
| Dirty Bit | — | Set for modified cache lines. |
| Valid Bits | — | Set if parts of a cache line contain valid data. |
| Virtual TAG address | — | Most significant bits of the virtual address. The number of bits depends on the size of the cache line and the number of sets. |
| Context Number | — | Context triggering the load of the cache line. |
| Unprotected Bit | √ | Set if the corresponding cache line has been fetched by an unprotected process. |

Table 11.1: TAG-RAM entries used by the L1 data cache




(a) Total number of cache accesses




(b) Miss rates

Figure 11.2: Cache accesses and miss rates for selected cache configurations

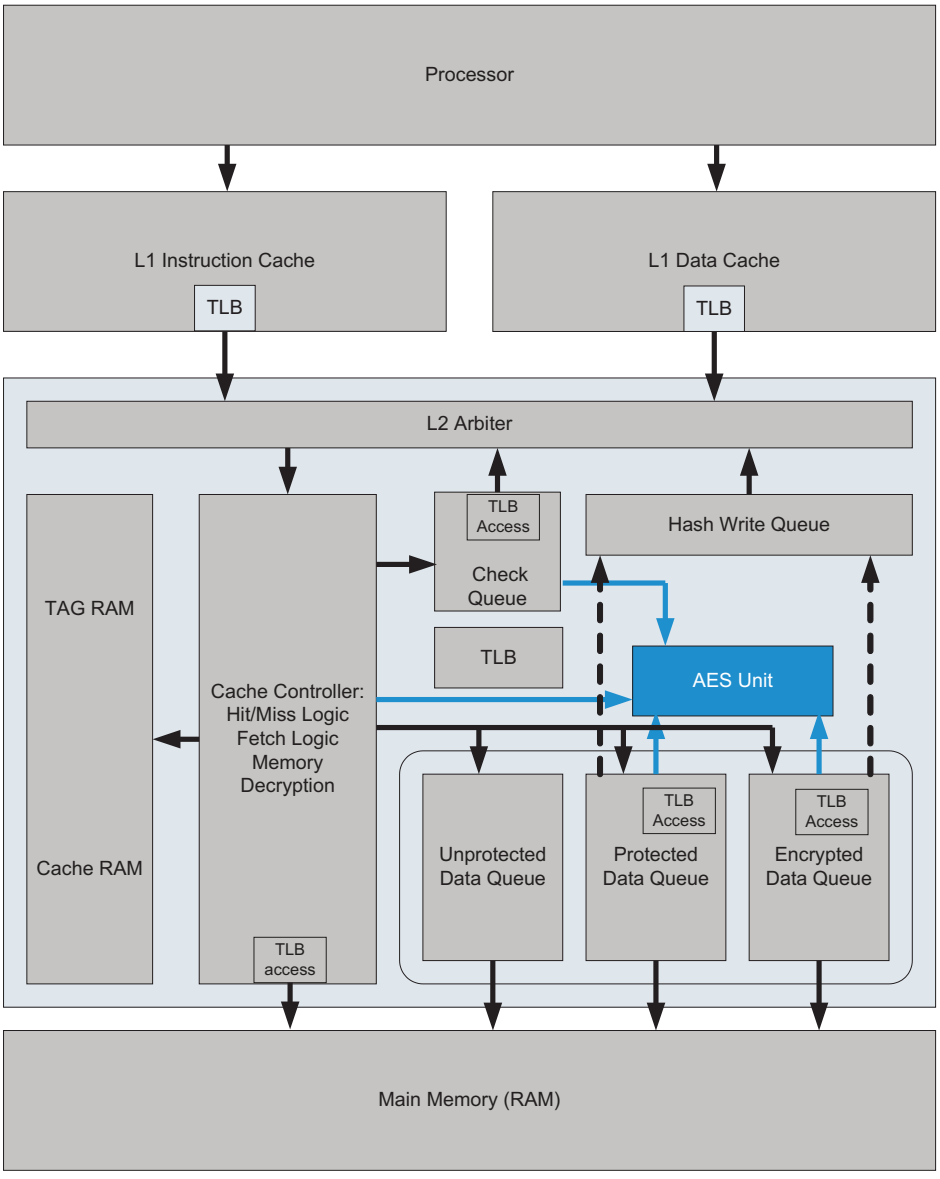



Figure 11.3: L1/L2 cache design

## 11.3 L1 Instruction Cache

For *SAM* architectures based on the Harvard design (separate data and instruction caches) the read-only instruction cache does not need to be modified. For the data cache it has to be possible to access encrypted parts without prior decryption, but this does obviously not apply to instructions, because encrypted instructions cannot be executed and would result in illegal instruction TRAPs most of the time. Therefore, the processor can directly terminate the currently running program in cases where encrypted instructions are to be executed without fetching them from the cache.
In contrast to the L1 data cache, which passes the value of the `PI` flag with its *unprotected* bit to the L2 cache, the L1 instruction cache does not use the `PI` flag for any operation. Instead, the value of the `PM` flag is directly passed to the L2 cache (see figure 11.1) in case of misses resulting in a verification of instructions regardless of the secure TRAP mode. This behavior has been selected to prevent undetectable memory manipulations in protected shared memory. This memory area is unencrypted and can therefore easily be modified by an attacker. For a succeeding attack, an adversary could place an *sts* instruction everywhere in protected memory as the last instruction in a cache line to skip the original entry point and all security checks by the operating system. Before executing *sts*, the `PI` flag is always zero and therefore no verification of the corresponding hash line would take place. This tampering attempt would thus remain undetected if the `PI` flag would be used for instructions as well. In contrast to `PI`, `PM` is always set in protected memory regardless of other flags for protected processes. Hence, this attack would be detected by the cache when verifying the corresponding cache line.

## 11.4 L2 Cache

The L2 cache has been newly designed, because most security-related functions are located here and therefore its design requires most changes compared to standard caches. Figure 11.3 gives an overview of the cache's internal units. The most noticeable change are five queues. They are used to process all functions related to protected memory and hash values in parallel to the normal cache logic and they are described in section 11.4.5. Note that the L2 cache contains only one TLB and all parts accessing the TLB have been marked with "TLB access".
The cache supports only a limited amount of encrypted processes. This helps to reduce the overhead caused by storing process-related data. The implementation is able to execute up to 16 protected processes at the same time. In the following, all parts shown in figure 11.3 are described in detail.

### 11.4.1 TAG RAM

Table 11.2 shows all contents stored in the TAG RAM of the L2 cache. It can be seen that the *SAM*-related overhead is much higher for the L2 cache than for the L1 data cache. All these aspects will be described in the following.
The L2 cache uses two bits to reflect the protection level of each cache line: The *unprotected* bit serves the same purpose as in the L1 cache. The cache refuses to update the hash tree during write-back if this bit is set. Additionally, for encrypted memory regions the *decrypted* bit indicates a decrypted cache line. This bit is used to prevent read or write access from

| Content | *SAM?* | Description |
|---|---|---|
| Dirty Bit | — | Set for modified cache lines. |
| Valid Bit | — | Set if the cache line contains valid data. |
| Physical TAG address | — | Most significant bits of the physical address. The number of bits depends on the size of the cache line and the number of sets. |
| Virtual TAG address | √ | Most significant bits of the virtual address. The number of bits depends on the size of the cache line and the number of sets. |
| Context Number | √ | Context triggering the load of the cache line. |
| Unprotected Bit | √ | Set if the corresponding cache line has been fetched by an unprotected process. |
| Decrypted Bit | √ | Set if the corresponding cache line has been decrypted. |
| Checked Context Bits | √ | Used to mark line checked for a specific context. |

Table 11.2: TAG-RAM entries used by the L2 cache

other contexts to decrypted memory parts and is used to optimize access to protected shared memory.
The L2 cache operates on physical addresses, but additionally stores the corresponding virtual TAG part of this address. This is required because it is impossible to compute the virtual address only from a given physical address. The virtual address is required for these reasons:

1. The virtual address is part of the hash-value computation. While the virtual address is available during fetch operations, it has additionally to be stored with the corresponding line to be able to compute the hash value when writing back the line.

2. It is required to determine the parent node in the hash tree, because the hash tree is based on virtual addresses.

3. The stored virtual address of a cache line and the requested virtual address must match in case of protected memory. If not, the cache initiates a recheck of the stored hash line regardless of the checked context bits described in the next section. This is required, since otherwise it would be possible to remap cache lines using the page table to different locations within the virtual address space, and of course, this tampering attempt has to be prevented.

On write-back, the cache has to select the matching secret key $k_b$ for hash-value computation and encryption. Thus, the context number of the fetching context (here called "owner context") is stored, too.

### 11.4.2 Hit Logic

The L2 cache contains a complex hit logic. The cache sets the following internal status bits if the requested cache line is already stored in cache RAM:

- Write Access: This bit is set on write access.

- Protected Memory: Set if the requested data is located in protected memory.

- Encrypted Memory: Set if the requested data is located in encrypted memory.
- Protected Instruction: Set if data is requested by a protected instruction.
- CX: Set if the owner context and the requesting context are different.
- DY: The accessed cache line is dirty.
- UN: The accessed cache line is unprotected.
- DE: The accessed cache line is decrypted.

Table 11.3 illustrates the action to be taken based on these internal status bits[4]. Each column represents another protection combination of the requesting instruction. For example, the first column represents an unprotected instruction accessing unprotected memory, whereas the last column represents a protected instruction accessing encrypted memory. The rows in this table show the different states of the cache line already stored in cache RAM. For example, the actions for shared protected memory are shown in column 5, rows 1, 3, 7 and 9. Possible actions are:

- WB: write-back current line and fetch the same line again with a different protection level
- F: fetch line and overwrite selected line in cache
- R: cache hit, process read-requests
- W: cache hit, process write-requests
- c: in cases where a *c* is appended to the action the cache line has to be verified as well even in case of a cache hit, if it has not been verified before for the accessing context number

After a line fetch or on write access the status bits of the corresponding cache line are updated according to the following Boolean functions:

$$\begin{aligned} DY &= \text{Write Access} \\ UN &= \overline{\text{Protected Memory}} \vee \overline{\text{Protected Instruction}} \\ DE &= \text{Protected Instruction} \wedge \text{Encrypted Memory} \end{aligned}$$

### 11.4.3 TLB

The L2 cache must be able to compute physical addresses of parent hash lines determined by their virtual address. Hence, as can be seen in figure 11.3, the L2 cache contains its own TLB. This design was chosen because this TLB exclusively contains addresses for the hash tree, which are typically not accessed directly by instructions. Consequently, there is little benefit in using the same TLB for completely different address regions. Furthermore, most TLB's in existing designs are directly coupled with the instruction fetch and memory access units and therefore another access by a third device would slow down normal processor operations.

[4]The following invalid status bit combinations are omitted, because they are prevented by the cache logic: "Unprotected Memory, Encrypted Memory" and "Unprotected, Decrypted".

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Protected Memory | | | | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| Encrypted Memory | | | | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| Protected Instruction | | | | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| Write Access | | | | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| CX | DE | UN | DY | | | | | | | | | | | | |
| 0 | 0 | 0 | 0 | R | R | R | R | R | F | W | W | W | W | W | F |
| 0 | 0 | 0 | 1 | R | R | R | R | R | WB | WB | WB | WB | WB | W | WB |
| 0 | 0 | 1 | 0 | R | R | R | R | Rc | F | W | W | W | W | Wc | F |
| 0 | 0 | 1 | 1 | R | R | R | R | WB | WB | W | W | W | W | WB | WB |
| 0 | 1 | 0 | 0 | F | F | F | F | F | R | F | F | F | F | F | W |
| 0 | 1 | 0 | 1 | WB | WB | WB | WB | WB | R | WB | WB | WB | WB | WB | W |
| 1 | 0 | 0 | 0 | R | R | R | R | R | F | W | W | W | W | Wc | F |
| 1 | 0 | 0 | 1 | R | R | R | R | WB | WB | WB | WB | WB | WB | WB | WB |
| 1 | 0 | 1 | 0 | R | R | R | R | Rc | F | W | W | W | W | Wc | F |
| 1 | 0 | 1 | 1 | R | R | R | R | WB | WB | W | W | W | W | WB | WB |
| 1 | 1 | 0 | 0 | F | F | F | F | F | F | F | F | F | F | F | F |
| 1 | 1 | 0 | 1 | WB | WB | WB | WB | WB | WB | WB | WB | WB | WB | WB | WB |

Table 11.3: L2 hit logic

### 11.4.4 AES Unit

The AES unit is a central part in this design and used by many other units (see figure 11.3). It is used to compute hash values and encrypted counter values. Due to the counter mode design and the hash-value computation algorithm, no decryption part is required. In our implementation the AES unit is controlled by a simple scheduler. This scheduler can pass a new cache line to be encrypted to the AES unit within each clock cycle. Since one cache line consists of four 128-bit blocks, the AES unit should consist of at least four AES encryption units to speed up memory decryption. It is possible to use a pipelined AES implementation as well, but this implementation consumes much more chip space (roughly speaking ten times more space) and – as can be seen later in section 14.6 – the number of parallel AES units has only a limited influence on the performance of the whole cache system.

### 11.4.5 Queues

The cache contains three different data queues to hide cache latencies. Their dependencies are shown in figure 11.4:

1. *Hash Write Queue*: On write-miss this queue stores the hash value to be written to cache RAM until the missing hash line is fetched. This queue is used to store computed hash values from the *Memory Write Queue* until the corresponding line is fetched to prevent deadlocks. Each cache access initiated by this queue is treated as a protected memory access.

2. *Memory Write Queue*: This queue contains cache lines to be written to main memory. Internally this queue is split into the following three queues:

   - *Unprotected Data Queue*: This queue is used for unprotected data to be written back to memory, and acts like a write buffer used in normal cache designs.
   - *Protected Data Queue*: This queue is used to compute hash values for protected data to be written back. The computed hash value is passed to the *Hash Write Queue* on write-miss.

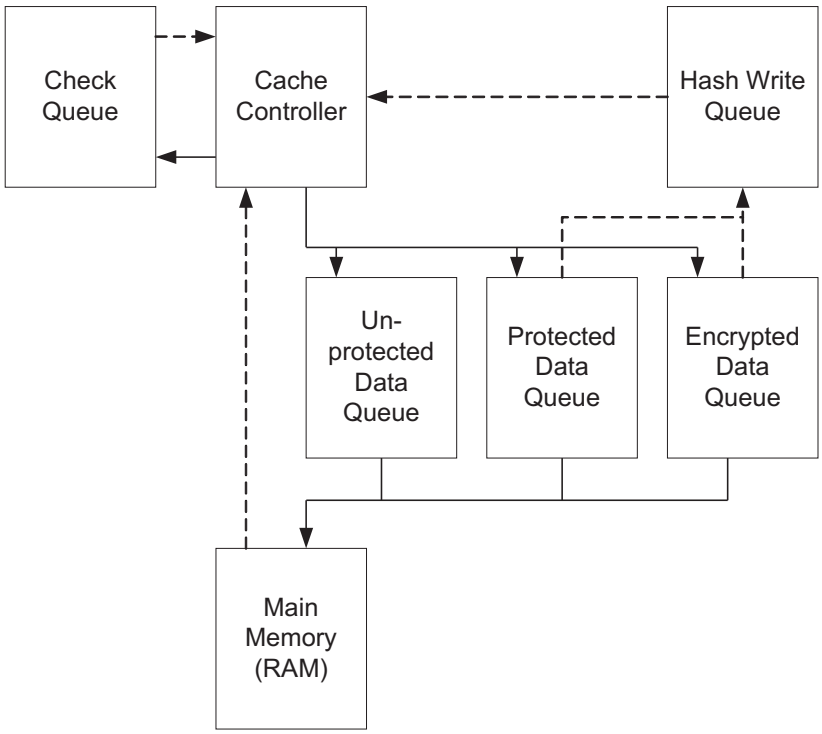



Figure 11.4: Queue data flow and dependencies

- *Encrypted Data Queue*: This queue serves the same purpose as the *Protected Data Queue* but additionally encrypts the given line before writing it back to memory.

3. *Check Queue*: The purpose of *Check Queue* is to verify cache or hash lines. A cache line is passed to the *Check Queue* if it is protected and unchecked for the accessing context. This queue then computes the hash value and compares this hash value with the parent hash values. The parent hash value is fetched from the cache. On a miss the queue waits until the requested data is fetched. For encrypted data the hash value required to decrypt the line is not directly passed to this queue. Instead, the cache waits until the *Check Queue* requests the line for verification. This design has been chosen to reduce the load of the *Check Queue*. Each cache access initiated by this queue is treated as a protected memory access.

Note that all queue elements can access the AES unit as needed. Three *Memory Write Queues* can result in data located at different memory regions written to memory possibly in a different order compared to models with only one *Memory Write Queue*. This is no problem for single processor systems but may cause problems in case of multiprocessor systems for example when locking memory using spinlocks. Hence, all data and the lock must not be stored in differently protected memory regions to ensure the original write order. In cases, where this is not possible the SPARC instruction set allows write synchronization by using the *stbar* instruction [85].
Each queue has a predefined size and only the first element can initiate a cache or memory access. The maximum sizes of all queues except for the *Hash Write Queue* can be chosen freely. The size of the *Hash Write Queue* $S_{HW}$ depends on the sizes of the *Protected Data*

*Queue* $S_{PD}$ and the *Encrypted Data Queue* $S_{ED}$:

$$S_{HW} = S_{PD} + S_{ED} + 1$$

Otherwise deadlocks might occur if the following conditions are met:

- The main cache logic waits until one of the *Memory Write Queues* has written back a specific cache line.
- The corresponding *Memory Write Queue* wants to access the *Hash Write Queue* to write the newly computed hash line.
- The *Hash Write Queue* is full and waits for the cache logic to become idle.

This deadlock can be solved if the *Hash Write Queue* is always able to store all computed hash values from all *Memory Write Queues*. Since the *Hash Write Queue's* priority is higher than the priority of other queues it is guaranteed that the queue contents can be written back to memory.

### 11.4.6 Cache Arbitration and Deadlock Prevention

The arbiter controls internal and external access to the cache controller. External access is initiated by the L1 I/D cache and internal access is requested by the queues. Since only the internal access can result in deadlocks, the L1 cache is omitted in figure 11.4. This figure marks all data flows back to the cache controller with dotted arrows.
The arbiter uses additional information to control bus assignment to prevent deadlocks and to ensure cache consistency. Due to the queue design, cache lines may be stored not only in the cache RAM but also in a queue. Therefore, the main cache logic cannot fetch a cache line if this line is currently processed by one queue. To prevent a costly termination of cache accesses, the arbiter is used to control L2 cache utilization based on its knowledge on queue contents. The arbiter considers the following information when granting access to the cache:

- If a requested cache line is in one of the *Memory Write Queues*, cache access is deferred until the line is written back to memory.
- A hash line cannot be accessed until all data for this hash line is written from the *Hash Write Queue* to cache RAM. The same applies in cases where the corresponding hash value is to be computed by the *Memory Write Queue*.
- The size of all queues is monitored. If one of the queues has less than one free element, all L1 requests are stalled until one element has left the queue. This prevents deadlocks on full queues caused by L1 requests.
- Queues with more active entries have a higher priority to access the cache than queues with less active entries.

If the cache logic detects that a cache line to be accessed is currently stored in the *Memory Write Queue* it simply stalls until the cache line is written back to memory to re-fetch it. This may happen, because the cache arbiter is only able to compare the actually fetched address with queue contents, but not their parent address within the hash tree required to decrypt

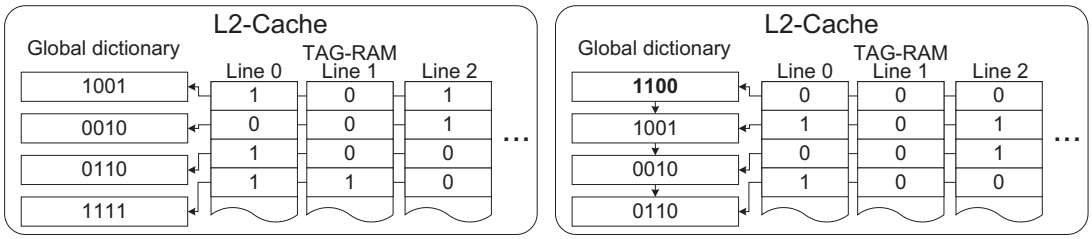
encrypted memory. In cases where waiting would result in a deadlock, the current operation is terminated to give other queues access to the cache.
This arbitration scheme can lower the probability of deadlocks, but the cannot be prevented due to the *Check Queue*. In the worst case writing one cache line to this queue may result in two additional cache lines to be verified. The following example illustrates this:

1. A cache line $L_1$ is passed to the *Check Queue* for verification.
2. The *Check Queue* computes the hash value $h_1 = H(L_1)$ and requests the corresponding hash line $L_{h_1}$ from the cache for comparison.
3. The request misses and the cache has to replace a dirty encrypted cache line $L_2$ to be able to load $L_{h_1}$.
4. $L_2$ is passed to the *Encrypted Data Queue* and the corresponding hash value $h_2$ is computed.
5. The cache passes the newly fetched cache line $L_{h_1}$ to the *Check Queue* for verification and proceeds the pending request initiated by the *Check Queue*.
6. The *Check Queue* deletes $L_1$ and proceeds with $L_{h_1}$.
7. When the *Encrypted Data Queue* has computed the hash $L_{h_2}$, it is passed to the *Hash Write Queue* and later to the cache. This may result in another miss and another new entry for the *Check Queue*.

Hence, the *Check Queue* likely is to overflow. To overcome this problem, the *Check Queue* requests hash values non cacheable when there is only one free entry left. The fetched cache line still has to be verified, but it cannot cause a cache line replacement resulting in one additional *Check Queue* entry only for each line to verify. It is therefore still sufficient to require one free queue entry.
Figure 11.4 shows another data flow which might cause deadlocks, because each protected data write-back to memory results in a hash value written back to the cache. The only case in which the *Protected Data Queue* and the *Encrypted Data Queue* could block is an overflow of the *Hash Write Queue*. But since this queue size depends on the sizes of the *Protected Data Queue* and the *Encrypted Data Queue*, this could happen only if the *Hash Write Queue* is not able to access the cache for a long time. This might happen in cases where the main cache logic waits for data to be written back to memory to fetch it again. The cache logic solves this possible deadlock by aborting the current operation to let the queues access the cache. This is a solution, because the blocking access could only be initiated by one of the L1 caches, since the cache waits for data in encrypted memory regions only. All other cases in which data is located in one of the *Memory Write Queues* are detected by the arbiter which prevents cache access until the data is written back to memory.

### 11.4.7 Speculative Execution

One of the countermeasures described in section 7.10.2 to prevent speculative execution-related attacks has to be implemented in the cache. The cache has to stall possible fetches until the *Check Queue* has finished verification of all previously fetched lines. This is implemented using a global flag, called the “tainted” flag, which is set on each protected line fetch issued




(a) Context Dictionary (b) Adding context "1100"

Figure 11.5: Context Dictionary

by one of the L1 caches. If this flag is already set when the cache wants to perform a fetch operation issued by one of the L1 caches, this operation is aborted to allow the *Check Queue* to access the cache. Until all data has been verified the tainted flag is cleared and the previously aborted operation is restarted.

## 11.5 Performance Optimizations

This section provides a brief overview of additional performance optimizations to the architecture. Most performance optimizations should take place inside the cache, since the cache is the bottleneck of this architecture. The cache contains most of the required cryptographic functionality, provides memory integrity verification in parallel to its normal operations and additionally stores hash values. All these properties cause a notable overhead to the normal cache operations required for unprotected programs.

### 11.5.1 Read-only Shared Memory

Since the operating system is partly protected and shared between all running processes – as described in chapter 10 –, the same protected memory area is accessed by many processes. This protected area has to be verified for each protected process, since each process provides its own hash values for this. To prevent expensive re-verifications for already verified data already stored in the cache, the cache provides a checked context dictionary. This dictionary is used to provide an efficient method to mark cache lines as "already checked" for several contexts.

The L2 cache verifies each cache line accessed by a protected instruction. Already checked cache lines do not have to be rechecked, as long as they remain inside the cache. Hence, a hash tree walk to verify a newly fetched cache line stops when the first already verified hash value is found in the cache RAM. Obviously, a cache line already stored in the cache and checked for a protected context A may not be trustworthy for a protected context B. Since frequently used parts of the OS remain in this kind of shared memory, the cache has to keep track of the contexts the current cache line has been checked for.

Special care must be taken to reduce the overhead caused by storing this information, because it has to be stored for each cache line in the cache. Therefore, *SAM* uses a dictionary-based approach shown in figure 11.5a to minimize the amount of RAM to store this information. The last four different context numbers a check was processed for are stored in a global dictionary. Each cache line (figure 11.5a shows three sample lines) now has to keep track of which of

the globally stored context numbers it has been checked for. Hence, this requires only four additional dictionary bits for each cache line.
Each time a new entry is added to the dictionary the oldest entry has to be removed first and the dictionary bits in each cache line must be updated. This is shown in figure 11.5b for the new protected context "1100". To speed up this task, the global dictionary and the dictionary bits for each cache line are stored in shift registers. Thus, they can easily be updated in one clock cycle. On write access, the context bits of others than the writing context are deleted.

### 11.5.2 Prefetching

Prefetching of memory contents to speed up cache operations has a long history. The idea is to predict future misses and fetch these data before it is actually requested by the processor by taking advantage of idle cycles on busses and caches. Obviously, the speedup achieved by prefetching depends on the quality of the prefetcher. If most predictions are incorrect, the additional overhead may indeed decrease performance. Of course, the prefetching accuracy is to a high degree dependent on the memory access pattern of the executed program. If this access is mostly random, prefetching cannot give any advantage. However, most programs generate at least partly regular access patterns, which can then be predicted.
For the *SAM* architecture, prefetching is more complicated compared with normal caches because of the following reasons:

- The prefetcher has to support memory views to prevent permission misses.
- The cache performs additional cryptographic tasks, such as hash tree walks. This results in fewer idle cycles to perform and hide prefetching tasks.
- Prefetching cache lines in encrypted memory may result in two cache misses, thus increasing the penalty in case of wrong predictions.

In this work two prefetching mechanisms have been evaluated. The first one, hash value prefetching, prefetches all hashes before they are actually requested by the *Check Queue*. The second prefetcher predicts future load accesses.

#### 11.5.2.1 Hash Value Prefetching

The prediction accuracy for hash values is 100%, since all required hash values can be computed directly. However, the prefetcher may try to prefetch values already stored in the L2 cache, thus degrading the possible performance gain.
The prefetcher is implemented using the same queue design as used by the *Check Queue*. It sniffs the bus to the *Check Queue*, reads all addresses addressed to the *Check Queue*, and stores them in its own queue. Then the parent of the stored addresses is computed, the corresponding physical address is requested from the TLB, and the computed address is fetched. This prefetcher helps to hide two latencies. First, the TLB access by the *Check Queue* is sped up, since this access will now always be a hit due to the previous access by the prefetcher. Second, the memory latency can be hidden as well, since *Check Queue* fetches are always hits in the L2 cache if the prefetcher is able to request the corresponding cache line before the *Check Queue* requests it.

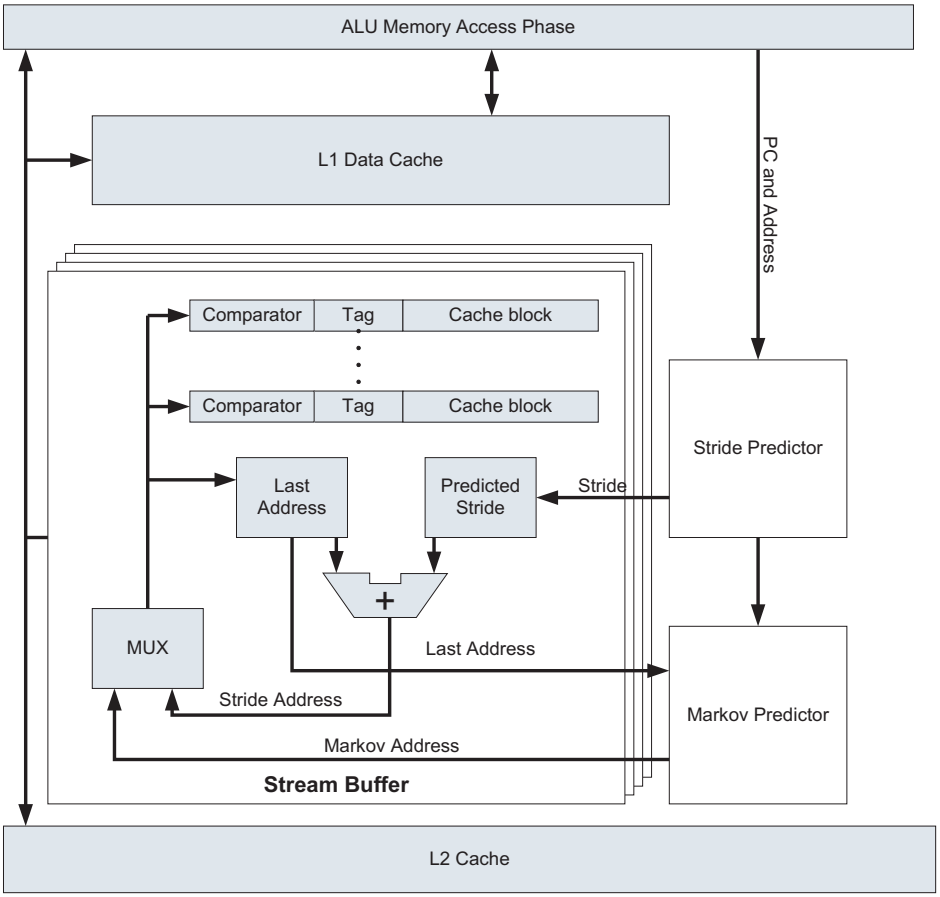



Figure 11.6: Stride-Filtered Markov-Predictor

#### 11.5.2.2 Data Prefetcher

The prefetcher is based on a design proposed in [83] using the address prediction confidence scheme for stream buffer allocations. As can be seen in figure 11.6, this design is a hybrid approach combining stream buffers with a Stride-Filtered Markov Predictor (SFM-Predictor). A stream buffer is a small fully associative cache used to extend the normal cache first proposed in [41]. This buffer prefetches predicted loads and moves them on access directly to the L1 cache. A possible predictor is a stride predictor. This predictor analyzes the missing addresses of load instructions to compute the stride between consecutive misses caused by the same instruction. Future misses are then predicted using this stride. This design has been analyzed in [23] and produces good results for programs accessing large arrays like data structures.
A Markov predictor [40] bases its prediction on the last values seen. An order $n$ context Markov predictor uses the past $n$ values to predict the next one. It can only provide a prediction if the given pattern has been seen and the transition is recorded in a prediction table.
The overall prediction accuracy can be improved if the input for the Markov predictor is filtered by means of a stride predictor. Only misses that cannot be predicted by the stride predictor are stored in the Markov predictor which results in a better prediction of stride- and non-stride-based memory access.
In the Markov predictor described above the cache line address is used as an index to select the entry with the predicted address. Hu *et al.* [35] has shown that an indexing scheme based on tag correlations (Tag Correlation Prefetcher, TCP) can outperform classical address indexing schemes. Hence, a Tag Correlation Prefetcher instead of the classical Markov predictor has been evaluated as well for the SFM predictor.
For the $SAM$-enabled cache, the streambuffer has been enhanced to support memory views by storing the context number and the protection level of the the requesting instruction. Additionally, the predictor can be configured to request verification of the prefetched cache lines.
The results of the different predictors are shown in section 14.6.

# Chapter 12

# Application Design

This chapter discusses the proposed application design and the modifications for *SAM*-enabled programs.

## 12.1 Compiler and Assembler

In contrast to AEGIS and XOM, *SAM* does not require any changes to the compiler. Hence, *SAM's* changes are high-level-language independent and existing compiler suites can be used. Though, the additional instructions require changes in the assembler. But since these instructions are not used in user mode, no *SAM*-aware assembler is needed for application design. Many operating system kernels are mostly written in high-level languages like C to support their portability. Since access to hardware like the Memory Management Unit or interrupt handling is not supported by C, low-level parts of each operating system are partly written in assembler. *SAM's* new instructions are used for interrupt handling and therefore they are used in parts already written in assembler resulting in an enhanced assembler as the only software requirement for *SAM*. The required changes for an existing assembler have been evaluated exemplary for the GNU binutils[1]. Only ten lines with the definitions of the new instructions have to be added to the file `opcodes/sparc-opc.c` containing the definitions for all sparc assembler instructions (2024 lines) to support the new instructions with different addressing schemes.

## 12.2 *SAM Linker*

### 12.2.1 Tasks

The *SAM Linker* converts a previously statically linked program into a protected program by encrypting it and computing the sparse hash tree. The *SAM Linker* is not able to move a previously linked program to higher addresses to get free virtual address space at lower addresses for the hash tree. Hence, the program has to be directly linked to higher addresses by the normal linker. For the GNU binutils this can easily be achieved by using a modified linker script. A linker script controls the behavior of the linker and contains the start address of the program. All other addresses are given relative to the start address resulting in only one changed line compared to the standard linker file. In this work all programs are linked

[1] `http://www.gnu.org/software/binutils`

to the base address `0x70000000`, thus leaving enough virtual address space for the hash tree. For smaller programs much higher addresses are possible as well to reduce the size of the hash tree.
The *SAM Linker* performs the following tasks when creating a protected program:

1. It reads and analyzes the normal program. This includes reading the entry point which points to the first instruction to be executed.
2. The base address of the hash tree is computed.
3. The random base key $k_b$ and the random pattern $r$ are generated.
4. A section with initialization parameters is written to a newly created executable. It is located directly below the address space of the operating system, its contents are described in 10.2.5. In the following this address space is referred to as *protected compartment*.
5. Now the hash tree with its encrypted root hash is computed over all protected memory. The protected operating system is read from a file to be able to compute matching hash values.
6. After creating the hash tree, the program can be encrypted. All parts including the `.text` and `.data` sections are encrypted.
7. The hash tree and the encrypted sections are written to a new file. This file is marked as a protected program by specifying a dedicated architecture.
8. A configuration section is added to the program. It contains $k_b$ and $r$ encrypted with the processor-specific public key $k^P_{pub}$:

   $$E_{k^P_{pub}}(k_b||r)$$

   and the four base addresses required for the memory layout as described in section 7.7.

The next subsection provides more information about the file format of protected programs.

### 12.2.2 *SAM* File Format

The current implementation of the *SAM* file format is based on the ELF layout. When reading ELF files the kernel directly maps chunks with a size of a page to memory. Hence, the position of data in the file and in memory relative to a page boundary has to be the same.
As described in section 2.2.1, the kernel is only able to map data located in segments to memory. A segment always covers a continuous memory region, and "holes" in a segment are not allowed. Hence, the sparse hash tree has to be stored in several segments to save space in the file. Segments can cover more memory than physically available in the file, because their header contains two entries, one for the overall size in memory and one for the possibly smaller amount of data stored in the file. For this kind of segments the operating system "fills" the remaining data with zeros when mapped to memory.
Each segment occupies an entry in the section header which is located at the beginning of each ELF file. This header grows and would result in different offsets, because segment data directly follows the ELF header. Since already linked data cannot be moved to other addresses, the *SAM Linker* inserts additional zeroed regions into the program.
*SAM* adds the following new segment types:

1. `PT_SAM_CONF`: This segment contains the memory base addresses and the encrypted keys. It is mapped to memory during program loading and the kernel unmaps it after successful program loading.

2. `PT_SAM_INIT`: The page containing the protected compartment is stored in this segment.

3. `PT_SAM_HASH`: The sparse hash tree is stored in segments of this type.

Adding new segments to an existing ELF file is not possible, because each new segment has to be referenced in the program header table located at the beginning of each ELF file. Hence, the *SAM Linker* has to create a new header before the original header while keeping the page alignment of the already existing segments. The old header remains unmodified and is encrypted when converting a program to a protected one since it is located in the encrypted memory area. It can therefore be accessed by the program initialization routines without the risk of tampering. The header used by the kernel to map the program to memory is not directly protected, but the processor would detect tampering attempts based on wrongly mapped segments using the provided hash values.

## 12.3 Library Support

Almost all programs require functions from additional libraries for their operation. For example, on UNIX-compatible systems the libc is an abstraction layer between the operating system and an application. The library provides functions like file or network access and memory management and most functions result in a system call. Almost all standard system calls are covered by the libc and, therefore, only the libc has to be adjusted for *SAM*. Changes are required at the following places:

- *SAM's* memory layout requires new functions to allocate unprotected memory, since the default for memory allocations is the encrypted heap.

- Parameter passing between the operating system and a program requires unprotected memory. Hence, all library functions passing pointers to the kernel have to be adjusted to allocate the memory in unprotected regions. Sometimes the pointer is given by the user program, for example when using read or write functions. Then the user program can be changed to explicitly allocate unprotected memory, or the library can copy this memory to unprotected areas. We propose a hybrid approach, that is the library checks all pointers to be passed to the kernel. In cases, where the pointer points to encrypted memory the data is copied to unprotected memory and then passed to the kernel. On completion data is copied back to encrypted memory.

- System calls may call unprotected kernel functions. Hence, the results passed from the kernel to the program may be forged. Therefore, the library has to perform basic validity checks. For example, when memory is requested from the kernel, the library has to verify the returned memory address to prevent memory based attacks by the kernel.

With a modified library existing programs can be used with small or no modifications. Indeed, existing programs are mostly not written to detect attacks based on their input data, especially if they are written as a console application. Therefore, changes are required to increase security. In the following some changes are presented.

## 12.4 Limited Execution

*SAM* programs can only be executed on the dedicated processors. Hence, unintended distribution is prohibited. But in some cases even the number of executions of a program should be limited as well. For example, in case of rented software a protected program should only be executable within one month.
Additional control of the execution of protected programs is only possible by the program itself. For example, a program could implement a license server approach, where it requests execution permission on start-up. Since a public key embedded in a protected program cannot be changed, a secure authentication of a remote license server is possible to prevent man-in-the-middle attacks.

## 12.5 Data Exchange

The AEGIS architecture provides additional hardware functions to sign results processed on a processor to prove their integrity. *SAM* does not provide these complex checks in hardware, but they can easily be implemented in software. For example, a protected program may embed a public/private key pair. This key can then be used to sign and possibly encrypt all computation results. For large results a hybrid approach using a random secret key for encryption (this key could be generated using the cryptographic functions described in section 7.5) could be used as well to improve encryption performance. Since the private key is known only by the program, all signed data can only be generated by the program.
Checking addresses or data structures cannot prevent forged data passed to the protected program. For example, when a program reads a file, this file could have been modified by the administrator of the system. Hence, the usage of cryptographic functions is a must each time external data is read or written. In the case of a file, the contents could be encrypted and hashed. The protected program reading the file can store the encryption key for the file directly in a constant variable, since the whole program is encrypted by *SAM*. Therefore, no user interaction, for example to type in a passphrase to decrypt a key, is required to read encrypted files. Network access can be secured using TLS ([18]) or similar protocols providing encryption and data integrity. Much the same as for the file access, keys can be stored inside the encrypted program. Secure access to external resources can further be simplified by using standardized libraries and wrapper functions.
However, even the best security architecture cannot prevent programming errors in applications. Though *SAM* prevents external modifications to a program, the smart use of forged data could exploit programming errors, which could further reveal internal data or even alter the program flow in some ways. Prevention of this kind of errors may be possible with high-level programming languages, which encapsulate direct memory or pointer access. Examples are the Java programming language or C++, when using the Standard Template Library (STL) or other high-level libraries.

## 12.6 Sample Application: Java Virtual Machine

Possible attacks for programs have been listed in chapter 3. Protected programs are still vulnerable to some of these attacks. For example, when a protected program is reading input data, buffer overflows are still possible. A possible solution for some popular attacks is the

usage of a high-level programming language which does not allow direct memory access and pointer handling. An object-oriented programming language with this properties is Java. Java programs are executed on a Java Virtual Machine (JVM, [52]), which prevents any direct access to the underlying hardware. However, most JVM's are implemented using C and, therefore, possible attacks are shifted from the user program to the underlying JVM, which has direct memory access. The following security implications have to be considered when using Java to execute programs:

- Many virtual machines provide a so-called "sandbox approach" to execute programs. The sandbox limits the actions which are allowed to be performed by a program. For example, disk and network access can be limited.
- In the past several JVM vulnerabilities have been discovered. This is mainly caused by the long history of Java and initially limited security capabilities. Newer virtual machine approaches like Microsoft's .NET provide more security [66].
- If a JVM implementation is verified by security experts, the risk of exploitable programming errors can be reduced.
- Each time a vulnerability in a virtual machine is found all programs running by that machine are affected. Hence, it is still possible to exploit programming errors of Java programs even if the program itself does not contain exploitable errors.
- Java programs depend on a huge library of Java classes. A class implementation may contain programming errors as well which may affect the security of Java programs.
- Many programmers have no special experience in writing secure program code and often input data is not verified. This gives attackers more weak points. When using a virtual machine some typical attacks are not possible.

Due to the given reasons, using Java in a JVM is a trade-off between improved security both for the executed program and for the computer executing a Java program and new security risks. We think that the advantages of using Java for software development for secure applications outweigh the risks. Furthermore, many Mobile Agents have been written in Java and they have to be protected. Therefore, a Java Virtual Machine has been adjusted for the *SAM* architecture to provide additional security. In the following the security design of the *SAM*-enabled Java Virtual Machine (SJVM) is presented.

### 12.6.1 Design Goals

The *SAM* architecture provides additional protection mechanisms which are useful for Java programs as well. But in contrast to native programs, Java programs have different properties resulting in different demands. Hence, the following design goals for a *SAM*-protected JVM have been considered to provide the same security-related goals as for the *SAM* architecture:

- Execution of encrypted class files.
- Class file verification to prevent manipulations.
- Dynamic loading and linking of encrypted class files. Unlike *SAM*, Java programs cannot be linked statically. Hence, class files have to be decrypted, verified and linked during runtime.

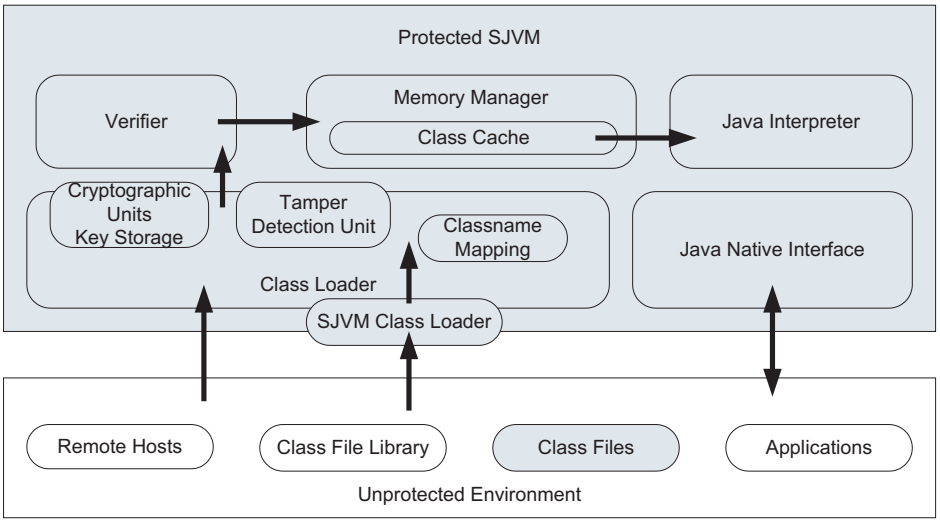



Figure 12.1: SJVM Overview

- Movable encrypted class files. Code migration is an important feature of Java code and has to be supported.
- Support of runtime class analysis with the reflection API.
- Class verification by the Java program. Each Java program should be able to decide during runtime if a possibly encrypted class file is trustworthy or not.

The following architecture description shows how these requirements have been put into practice.

### 12.6.2 Architecture

The basic architecture of the SJVM is similar to the *SAM* processor design, as can be seen in figure 12.1. The SJVM is considered to be trustworthy and can identify itself using cryptographic mechanisms. All verification and encryption steps regarding Java classes are done in software by the SJVM. The *SAM* architecture provides just one protected and encrypted container for all decrypted and verified class files. The presented architecture has been implemented based on the SableVM[2], a free JVM interpreter. In the following the architectural components are described in detail.

#### 12.6.2.1 Key Handling

Like a *SAM* processor each SJVM can be identified by a unique public/private key pair compiled into each machine. The keys are protected by the *SAM* architecture and can only be read by the SJVM. The public key is used to protect symmetric cryptographic keys. The symmetric keys are used for class file encryption. In contrast to *SAM*, the SJVM additionally supports digital signature handling to protect Java programs.

[2] `http://www.sablevm.org/`

Java programs are often distributed in JAR (Java Archive, based on the ZIP archive format) files. These files contain all application-specific class files and additional metadata. Hence, all additional cryptographic data like keys can easily be added into separate files within the archive.

#### 12.6.2.2 Encryption

Java class files are encrypted to hide their contents. They are decrypted transparently by the JVM and stored unencrypted in the protected container[3]. Class file encryption should not affect the file size compared to the original size. Hence, AES in Cipher Feedback Mode is used for encryption and a hash over the class name is used as an initialization vector.

#### 12.6.2.3 Class File Integrity Verification

For class file integrity verification the following two cases can be distinguished:

1. To ensure the integrity of encrypted classes a cryptographic hash value is directly incorporated into the class file and additionally encrypted.

2. Unencrypted classes stored on the file system of the execution host are protected by a cryptographic hash as well. This hash is stored with the corresponding class file name in a separate file. Since Java organizes class names hierarchically, a file stores the hashes of all classes on the same level. Additionally, upper level files contain hashes of the hash files of the next lower levels. This results in a tree structure of hashes and reduces the number of trusted hashes in the JAR file, because only the root hashes have to be given.

Hence, the SJVM provides efficient verification mechanisms both for encrypted and for unencrypted classes.

#### 12.6.2.4 Spoofing of Class Names

A class name may contain valuable information for an adversary. In Java, all files containing class descriptions are named with the class name. Typically, obfuscators are used to hide these information by replacing all class names (and method names) by randomly chosen names. This helps to hide internal class names but causes problems when using the reflection API, because the names known by the programmer and used by him in the program are changed. To overcome this problem the SJVM provides a more transparent solution by mapping all file names to randomly chosen ones, but additionally storing the mapping encrypted with the protected classes. This allows the SJVM to restore the original names when loading a class. The file with the mappings is stored within the JAR file.

#### 12.6.2.5 Program-Controlled Verification

Java allows linking against classes which are unknown during program creation time. For example, these classes can be automatically generated or downloaded from a web server.

[3] In the following the terms "encrypted" and "decrypted" are used from the JVM's point of view, because *SAM's* protection mechanisms are transparent to programs. Of course, even "unencrypted" data stored by the JVM in RAM is encrypted and hashed by *SAM*.

Therefore, these classes cannot be encrypted with the program's secret key and the corresponding hash of the file is unknown as well. Since loading of these classes is initiated by the Java program, verification of these classes has to be initiated by the program as well. Hence, the SJVM provides mechanisms for a user-defined class loader to verify this kind of classes by invoking a callback function of the user-defined class loader.

#### 12.6.2.6 Memory Management

For simplicity, it is assumed that all data like classes and variables are stored in memory and not written back to disk by the SJVM. Then verification of classes is required only at the first time a class is loaded from disk or over the network and the SJVM's memory acts like a huge cache for all data. The same applies for the hash values of the unencrypted classes stored on disk. If the SJVM does not store the files containing the hash values in memory, the full hash tree has to be verified each time a new class is loaded.

### 12.6.3 Usage Scenario

The proposed SJVM can be used to provide a secure execution environment for mobile code, like the one used for the Shaman system [82]. Assume that on each *SAM* processor in a network a SJVM is running a Java Execution Server (JES) application. This JES has the following properties:

- All JES classes are encrypted.
- It provides a unique private key and a signed public counterpart.
- It provides remote procedures to send, receive and execute (encrypted) Java programs. Program migration makes use of the Java object serialization to move a program in its current state.
- Additional remote procedures for key handling and encryption are provided.

With these properties a program migration from JES $A$ to JES $B$ may consist of the following steps:

1. $A$ determines the next JES where the program (mobile agent) $P$ is to be executed.
2. $A$ initiates an encrypted TLS connection to $B$. During TLS handshake the public keys $k_{pub}^{JES_A}$ and $k_{pub}^{JES_B}$ are exchanged.
3. Both keys are verified; if they are trustworthy, a secure connection is established and a possible man-in-the-middle attack is prevented.
4. $P$ is stopped and serialized and the serialized objects are transferred over the TLS connection.
5. $B$ receives the serialized objects and starts de-serialization. Since the SJVM does not execute unverified code, it invokes the callback function of the JES to verify the classes. The JES can then acknowledge loading these classes because they have been transferred over a secure channel.
6. The TLS connection is closed and $P$ can be executed on $B$.

This example shows that is is possible to implement mobile protected Java programs with the help of a SJVM. In cases where the protected programs need to write data to disk or start network connections the programs are responsible for data integrity and encryption. This is no problem, since the Java class library provides many functions for hashing and encryption.

### 12.6.4 Performance Analysis

The performance of the modified SJVM has been evaluated by measuring the time until a given program has been started on a Personal Computer without *SAM* extensions. A performance evaluation on the simulation environment later described in section 14.2 was not possible due to the low speed of the simulation model. The program has been started unprotected and protected several times to be able to measure the speedup. The measured speedup is approximately 0.83 and will be higher for most application scenarios, since the main latency results from the verification and decryption of class files on program start-up but once all class files have been loaded and cached in protected memory the performance degradation is much lower.

### 12.6.5 Security Analysis

The proposed design should provide the same protection to Java programs as the *SAM* architecture provides for native executables. This analysis only focuses on the design and not on a possibly erroneous implementation.
For each program executed as a protected program on the *SAM* architecture the goals given in section 6.2 are met. A security analysis of the *SAM* architecture is given in chapter 13. Therefore, only the SJVM-related parts are discussed here.
The SJVM is comparable to a *SAM* processor implementation. Due to its internal private key, Java programs can be bound to a specific set of SJVM's and are only executable by these SJVM's. The Java programs are fully encrypted and further protected by hash values. This prevents external modifications to these files and a program analysis is not possible.
Each Java program typically consists of several classes which can be hierarchically organized and stored in a JAR file. The class names and the hierarchical organization can give an attacker valuable information. Therefore, all class names are mapped to randomly chosen file names. But, in contrast to *SAM*, this does not hide all information about a program. Java loads all required classes on demand and therefore an adversary might still get timing information based on the access pattern of classes. But since encrypted classes do not contain any valuable information and external unencrypted classes provide standard functions this information is acceptable.
An attacker might try to replace unencrypted classes from the class path. This attack can be detected by the SJVM by checking the tree of hash values up to the encrypted root hashes provided by the encrypted program. Since the encrypted hash value cannot be modified, this attack is infeasible. However, the SJVM should not link external classes to a protected Java program without prior successful verification, i.e. non-speculative execution should be allowed.
Up to this point, only attacks to the protected program have been described. Attacks to the underlying execution host are possible, too, but they can be prevented by using the already existing Java sandbox mechanisms which can be used to restrict access to the host system.
As a result, the presented architecture provides mechanisms to protect Java programs both against manipulations and against program analysis. However, programming errors in the

SJVM or in the executed application cannot be prevented and they might be exploitable. But this is a general problem of large parts of protected program code.

# Chapter 13

# Security Analysis

This section analyzes the security provided by *SAM*. The analysis is partly based on the requirements given in section 6.2 and additionally presents the cryptographic parts as well as the logical design. It is assumed that direct physical attacks to the processor core are not possible.
Due to the complexity of the processor and the operating system, a formal proof of the security of *SAM* is virtually impossible. The secure processor communicates with other insecure components over several busses, IO and control signals. Additionally, the processor provides many internal states and, for example, behaves differently on state transitions like interrupts based on the previous state. Therefore, instead of a formal proof the security-relevant parts of *SAM* are analyzed and *SAM's* protection against commonly known attacks is discussed in the following.

## 13.1 Hashing

The memory hashing-scheme is used to detect any kind of memory tampering-attempts. *SAM's* security is based on the ability to detect any kind of modifications to protected memory contents and therefore the hashing has to be secure. Most hashing schemes used to detect tampering attempts on encrypted data calculate hash values over the ciphertext. This has the advantage that the hash value itself cannot reveal any information about the plaintext. *SAM* does not use this approach to be able to use the hash values as a counter for encryption. Therefore, other attempts to hide the plaintext have to be used. At first, we assume that the hashing function described in section 8.1 has all properties required for good hash functions as described in section 4.2, because the hashing scheme is based on a well-known application of block ciphers used to calculate hash values [74, §2.4.4]. Therefore, the hashing is not reversible and does not directly give any information about the hashed plaintext. However, an attacker could generate hash values of commonly used instructions and use them to identify the encrypted instructions. To prevent this, *SAM* XORs the plaintext with the random 512 bit $r$, thus making it impossible to get any useful information about the plaintext. Furthermore, even hash values computed over the same plaintext located at different memory addresses result in different hash valued due to the incorporation of $V$ into the hashing scheme.
The hashing scheme could only be broken if the attacker was able to get $r$. Since $r$ is encrypted using the processor's public key $r$ and this encryption is assumed to be secure, an attacker could try to attack *SAM's* hashing scheme to get $r$ or at least parts of it. According to

equations 8.1, 8.2, 8.4 and 8.5, $r$ is incorporated into the key and the plaintext when applying the AES encryption for the hashing algorithm.
Without loss of generality in the following we assume that the hash $H(p)$ for a 256 bit plaintext $p = p_1||p_2$ with a 256 bit secret random number $rnd = r_1||r_2$ ($p_1$, $p_2$, $r_1$ and $r_2$ have a length of 128-bit) is computed as follows:

$$H(p) = E_{p_1 \oplus r_1}(p_2 \oplus r_2) \oplus p_2 \oplus r_2. \tag{13.1}$$

We further assume that the attacker knows $p$ and the valid hash $H(p)$ as well as the hashing function. Since the attacker does not know $rnd$, he has to find a 256-bit value $x = x_1||x_2$ ($x_1$ and $x_2$ are 128-bit values), so that

$$H'(x) = E_{x_1}(x_2) \oplus x_2 = H(p). \tag{13.2}$$

If the attacker succeeds, $rnd$ can be computed as follows: $rnd = x_1 \oplus p_1||x_2 \oplus p_2$.
If we now assume that AES does not have any weak keys or other flaws that would allow an attacker to create a specific ciphertext while being able to modify the plaintext and the key, a brute force attack would be the only possible attack to compute $rnd$. Due to the length of $rnd$, a brute force attack is infeasible and therefore *SAM's* hashing scheme with a random value $r$ of twice of the size of $rnd$ provides a similar security as a hash computed over the ciphertext.
In the following more possible attacks related to *SAM's* hashing scheme are revisited.

### 13.1.1 Replay Attack

Here, the attacker replaces encrypted parts of the main memory with older encrypted data located at the same virtual addresses (typically without knowing the unencrypted contents). If the attacker replaces the corresponding hash as well, this attack might be undetected. Due to the hash tree and the permanent storage of the root hash inside the processor, this attack is not possible, since the current parent hashes would not match the replayed hash. Hence, the only possibility for attacks are cases in which the hash value matches by accident, as described for the birthday attack below.

### 13.1.2 Random Attack

The opponent selects a random value and expects that the change will remain undetected. If the hash function has the required random behavior, the probability of success is $\frac{1}{2^n}$, where $n$ stands for the number of bits in the hash. In *SAM's* case, this attack is not feasible, since we use 128-bit hashes.

### 13.1.3 Pre-Image Attack

Here the opponent searches another hash value that hashes to an existing hash value. This is not feasible, since it would require $2^{128}$ operations. Moreover, this attack requires that the attacker is able to compute the hash. The hashing algorithm is public, but without the knowledge of the secret value $r$ a valid hash cannot be computed.

### 13.1.4 Birthday Attack

Due to the birthday paradox described in section 4.2, two equal hash values can be found after $2^{\frac{n}{2}}$ hash-value computations with a probability of 0.5. In *SAM's* case $n$ has the value 128 resulting in $2^{64}$ computations. Due to $r$, the adversary is not able to perform this computations by himself, but he can try to monitor memory access until two hash values are equal.
*SAM's* hashing scheme incorporates the virtual address into the hashing algorithm. Thus, the adversary has to wait until two hashes for the same cache line match, since collisions for different cache lines are of no use. As the adversary is not able to compute the hash, he has to force the cache to compute as many hashes as possible as fast as possible. Hash values are computed only on cache line write-backs in case of dirty cache lines. Therefore, the cache lines storing the protected compartment are a good candidate, because they are written each time the program is interrupted. Therefore, a possible attack would be to interrupt a protected program using the timer interrupt virtually after each executed instruction followed by a cache flush.
If we assume that the memory bandwidth is approximately $2^{32}$ byte/s, then at most $2^{32}/2^6 = 2^{26}$ cache lines with a size of $2^6$ bytes can be written per second. This value is further halved, because for the attack to succeed a cache line has to be fetched and then written back, resulting in two memory operations. Hence, $2^{64}$ hash-value computations require at least $2^{64}/2^{25} = 2^{39}s$. This equals roughly 6 million years and, therefore, this attack can be considered infeasible. Additionally, this assumption does not take into account the time required for hash-value computation, memory encryption and memory verification. Furthermore, when using address bus randomization schemes much more data has to be transferred, which further slows down this attack.
Another limiting factor is the required memory for this attack to succeed, because the adversary has to store all $2^{39}$ hash values for comparison.

## 13.2 Cryptography

The encryption scheme is based on the use of the counter mode. As stated in [20], it is required that a unique counter be used for each plain text block that is ever encrypted under a given key. There is no particular indication on the counting values, as long as they satisfy the uniqueness requirement. This makes it possible for us to use the hash values as counters, since the design of the function $g$ used to generate an encryption key based on the virtual memory address $V$ and the encryption base key $k_b$ (see section 8.2) and the protocol guarantee that they are unique for a given derived key. In fact, as it is easily checked, they only depend on the contents of the particular block to be encrypted. In order to keep an overall good performance, $g$ should be evaluable in a short time frame (for example one clock cycle).
However, the ciphertext for the same data written at the same position in memory is always the same for a given protected program. Hence, *SAM* does not conceal the fact that written data is unchanged. This limitation could easily be solved in software by providing an additional counter value. However, this would result in an enlarged memory footprint and a lower speed and, therefore, this has not been considered for *SAM*. The applications that need to hide this information may store a small counter along with the data.
Another possible solution is stack and heap randomization. If the stack and the heap always start at different addresses for each program execution, the memory contents for each execution cannot be compared. If this technique is further used to randomize stack and heap

allocation during runtime, even function calls from the same recursion level do not result in the same encrypted memory pattern. Heap randomization can easily be implemented by the libc, but runtime stack randomization would require a modified compiler. Both techniques are already used to defeat several kinds of buffer overflow attacks and can easily be used to improve *SAM's* security. However, both randomization techniques can be implemented only if the instruction set contains instructions to get hardware-generated true random numbers. Otherwise, the tampered operating system could generate predefined values instead of random numbers to prevent randomization.

## 13.3 Speculative Execution

Without speculative execution of instructions an attacker is unable to modify protected data or instructions, because the cache will detect it using the hash values. *SAM's* speculative execution opens a small window in which protected instructions can be executed prior to their verification. For example, in counter mode each flipped bit of the ciphertext results in the same bit flipped in the plaintext after decryption. Therefore, in cases where the adversary knows the plaintext or at least parts of it, he is able to modify for example instructions intentionally. Much easier to accomplish are modifications of parts of the protected but unencrypted shared memory. Of course, these modifications will be detected, but due to speculative execution there might be enough time before detection to do something harmful.
Therefore, all possible paths to leak encrypted data within this small time frame have to be prevented by *SAM*. *SAM* introduces several restrictions to speculative execution to prevent any kind of leakage.
Protected data can be leaked by using the following methods:

- *Writing to unencrypted memory:* An attacker could try to read protected data to copy it into unencrypted memory. *SAM* prevents this attack by stalling memory writes by protected instructions to unprotected memory until `PV` is set. Hence, the tampering attempt would be detected before unencrypted data is written to memory.

- *Writing to unprotected registers:* Here the attacker loads encrypted memory contents into a general-purpose register and tries to unprotect it using *rprot*. This does not succeed, because *rprot* requires successful verification of all fetched cache lines before altering the register protection.

- *Writing to special registers and IO-ports: SAM* stalls writing to special registers and IO-ports until the cache has verified all instructions to prevent this attack. Processor flags like the C, Z, N or O flags are cleared after detecting a tampering attempt. Hence, they cannot be used to leak any information.

- *Writing to encrypted memory: SAM* does not delay writes to encrypted memory and therefore, tampered instructions can write arbitrary values to protected encrypted memory. However, since all cache memory contents belonging to the current context are deleted after detecting the modification, the attacker cannot take an advantage of this behavior.

- *Using the address bus:* The cache prevents any subsequent memory fetches each time a cache line has been loaded until the line has been verified successfully. Hence, the

address bus cannot be used as a side channel to leak information, and modifications in encrypted memory are of no use for an adversary.

- *Using side channels:* Most security architectures allowing speculative execution are vulnerable to side channel attacks. For example, an attacker could replace protected instructions by a sequence of instructions that read a value in memory, compare it with a constant and raise an interrupt based on the result of the comparison. Then, the attacker gets some information about the read value based on the occurrence of the interrupt. *SAM* prevents this attack by waiting until all data has been verified before processing the interrupt. Then, if any modification is detected, the interrupt generated by the attacker is overwritten by the interrupt raised by the tamper detection unit. Hence, the attacker is not able to distinguish if the interrupt is caused initially by his modified code or later by the tamper detection unit.

  In another scenario the attacker could execute a conditional branch instead of raising an interrupt and monitor the behavior of the processor. If, for example, the target instruction is already loaded, the program execution continues. In cases where the target instruction is not cached the cache would wait until all read cache lines have been verified before loading the cache line. Both cases cannot directly be distinguished, but an attacker could measure the power consumption of the processor to distinguish both cases. In this work no measures to prevent this kind of side channel attacks have been analyzed, but for a real implementation they have to be considered.

  The *SAM* design is a trade-off between security and performance. Typically, an architecture can be considered secure if the effort to break the security has to be much higher than the value of the protected data. For *SAM*, the value of the protected data may vary depending on the program. Hence, *SAM* could be enhanced to support both speculative and non-speculative execution based on the program's needs. This extension has not been considered for this work but can be added easily. Of course, non-speculative execution would result in a huge performance loss, as can be seen in section 14.6.1.

## 13.4 Processor Architecture

A security architecture, like *SAM*, is only useful if the execution of protected programs in simulated or tampered environments can be prevented. By providing a unique private key for each *SAM*-enabled processor protected by a tamper-resistant processor core, the distribution of protected programs can easily be limited to a predefined set of processors. Simulated processors are not possible, since they do not contain a signed public key[1].
The processor architecture provides mechanisms to ensure a secure execution environment through the use of encrypted and integrity-verified memory. In the simplest case there is only one program executed on a *SAM*-enabled processor. This program includes its own TRAP handlers and the whole accessed memory is encrypted and protected by hash values. Then it is obvious that an attacker cannot alter the execution of the program. The only thing an attacker might do is to interrupt the execution. External modifications to the intended program flow are not possible, since the attacker has no access to the processor's internal registers and units. In the following it is shown that the *SAM* architecture operating system design can be reduced to a secure single task execution.

[1] It is assumed that the manufacturer of the processor is trustworthy and keeps all sensitive keys secret.

In a multitasking environment a program is intentionally interrupted in order to execute other programs or handle other TRAP requests. These interruptions must be implemented in such a way that they cannot be used to attack the program flow or to reveal internal data of a protected program. The *SAM* architecture implements these requirements by providing a partly protected operating system. The architecture ensures that the instructions executed after interrupting a protected program are protected as well by preventing manipulations to the TRAP base register and enforcing a TRAP table located in protected memory. Since the hash values for this part of the operating system are provided by each protected program, the protected part can be considered as a part of the protected program.
The protected part of the operating system has full access to the program and its task is to leave the protected part ensuring that the program cannot be altered while interrupted, and that it is resumed exactly at the interrupted position. To this end, the current implementation stores all sensitive data in encrypted memory. Then, the protected part can drop its privileges by clearing the `PT` flag before executing unprotected parts of the operating system. The unprotected part of the operating system can only enter the protected part by calling a predefined set of entry points. Any of them sets `PT` and checks whether its own execution is allowed by verifying the previously stored flags. Other transitions to protected user mode are not possible, because they require `PT` to be set.
Other architectures like AEGIS or LaGrande require a secure boot process or a permanently protected operating system part. The drawback of both approaches is that the boot process involves more program code, which is unlikely to be implemented without security flaws, and the permanently protected operating system part in AEGIS is required to ensure valid memory management. This is in contrast to *SAM*, where the protected operating system performs mainly simple tasks like program state protection and register handling. Furthermore, the security of the protected part is not dependent on past actions performed before starting protected processes, which further reduces the possibility of security flaws. Sensitive data required for secure TRAP handling like the TRAP table base address cannot be changed by an attacker, because it is incorporated into the hashing algorithm, thus any modification would be detected because of non-matching hash values.
There is only the following small limitation in terms of providing a black-box-like execution of programs. The memory access pattern cannot be fully hidden. *SAM* protects the program counters so that an attacker is not able to read them. But, sniffing the memory bus, an attacker is able to record the accessed addresses at the cache line granularity. However, existing schemes to hide memory access patterns (see, for instance, [29, 100]) can of course be used for *SAM* as well.
Furthermore, when providing hardware implementations of cryptographic functions as proposed by *SAM*, memory access patterns cannot be used to disclose cryptographic keys. However, when making *SAM's* cryptographic units available to normal programs special care has to be taken to prevent any side channel attacks.

## 13.5 *SAM* Operating System Implementation

The following analysis is based on the threats listed in section 10.1 and assumes that the protected part of the operating system has been implemented based on the design presented in chapter 10.

- *Program counter manipulations*: The program counter is stored in the protected *com-*

*partment* and cannot be modified by unprotected instructions.

- *Register manipulations*: Registers are stored on the user stack which is protected by hash values.
- *Register value disclosure*: The hardware register protection prevents the reading of protected registers outside the protected kernel area. On context switches the register values are stored on the encrypted program stack by the protected part of the operating system.
- *Register mapping manipulations*: The register mapping is stored in the *protected compartment* as well.
- *Forging of memory contents*: *SAM* requires that a program verify all data read from unprotected sources. All data in protected regions is protected by hashes which incorporate the virtual address to prevent page table attacks, or written and verified by the protected part of the kernel (such as the environment variables and command line parameters). Therefore, the page table or the operating system functions handling page mapping do not need to be protected.
- *Manipulating system call return values*: This kind of attack cannot be prevented by the current design, because data from unprotected sources can always be manipulated. Fortunately, the number of different system calls used by most programs is fairly low and most direct attacks, such as forged pointers or sizes of these system calls, can be detected by the libc.
- *Tracking the accessed memory area*: When using *SAM* with additional address randomization schemes like HIDE, this attack can be prevented. In cases, where this is not possible due to performance issues, *SAM* prevents fine grained tracking of address information by clearing all page offsets on page misses. However, if an attacker has full hardware access, more address information can be recorded by sniffing the address bus.
- *Signal handling*: During runtime of a program any other process with matching permissions is able to send a signal to a program. Therefore, the protected program has to register signal handlers in cases where these signals should be processed. In all other cases the program is terminated. It is assumed that the operating system implementation securely passes signals to the protected program to return to the state of the interrupted function. Therefore, no further manipulations except denial of service attacks are possible.

## 13.6 Comparison with other Architectures

The architectures described in chapter 5 can protect program execution on different levels. The most secure execution environment are sealed secure co-processors like the IBM PCIXCC architecture [2]. However, as described above, secure co-processors are very expensive and limited in terms of processing power and memory. Other more lightweight security architectures like the XBOX and the LaGrande architecture use cryptography to protect memory contents, but decryption takes place in the processor and decrypted contents are stored in memory. This makes the architectures vulnerable to memory sniffing attacks.

To fill the gap between these lightweight security architectures and co-processor solutions, XOM, AEGIS and *SAM* have been developed. Since these architectures have been designed mainly as a replacement for a normal unprotected processor they do not offer enhanced protection against direct physical attacks to the processor core. This protection is required as otherwise an attacker might be able to read the secrets stored within a processor. AEGIS effectively protects the secret using Physical Unclonable Functions, but other attacks are still possible. For example, an attacker might try to modify the firmware stored within the processor's memory or disrupt control lines to change the behavior of the processor. These attacks are of course possible to XOM and *SAM*, too. Therefore, AEGIS does not provide more physical security than XOM or *SAM*. When comparing the memory protection of XOM, AEGIS and *SAM* it is obvious, that XOM provides the weakest protection. The usage of hash trees to prevent replay attacks is suggested, but not implemented. XOM does not provide secured system calls, hence only small parts of a program can be protected. Therefore, XOM does not protect the execution of larger programs, because the small protected parts can be executed in arbitrary order by an attacker (program flow manipulations).
The AEGIS architecture provides secure system calls and is not affected to this attack. However, AEGIS, like *SAM*, does not directly implement memory randomization schemes like HIDE, but suggests the additional usage of these schemes. For AEGIS, HIDE is required to reduce the amount of disclosed data using the memory decryption attack [84] without fully preventing it. *SAM* has been designed to prevent this attack directly, without the usage of memory randomization schemes. Hence, *SAM* provides the best memory protection when comparing it with XOM and AEGIS. However, *SAM* does not hide the information, that data stored at a given memory address is unchanged. This information could leak information to an attacker, but as described above (section 13.2) randomization schemes during runtime can prevent this information leakage.
The operating system security provided by *SAM* and AEGIS is comparable when implemented without errors. However, since the AEGIS kernel is responsible for the page table and other security-relevant functions, it is more likely that the resulting protected operating system part may contain exploitable programming errors.
As a result, the *SAM* processor architecture allows, in conjunction with a *SAM*-aware operating system, a secure execution environment for programs that meets the goals given in section 6.2.
However, since it is impossible for the operating system to analyze protected programs before executing them, it is possible that the contain viruses or other malicious program code. Therefore, it is advisable to enhance *SAM*-enabled operating systems with fine grained access control systems like SELinux to control and restrict access to resources. In another approach the operating system could be enhanced to start only digitally signed programs. This can easily be achieved by a signed encrypted base key $k_b$ and a verification of the signature before actually starting the program. A combination of current trusted computing concepts developed by the TCG and *SAM's* memory protection technique could further enhance the security of the whole system.
In the end, the security of a protected program is more endangered in most cases by programming errors —like buffer overflows— than by hiding the information-related to memory access. Therefore, protection schemes providing only a small gain in security but resulting in a big performance penalty have not been considered for *SAM*.

# Chapter 14

# Evaluation

In this chapter implementation issues and simulation results are discussed.

## 14.1 VHDL Implementation

Parts of the *SAM* architecture have been implemented as a VHDL model to prove their feasibility. The implementation covers the processor-related changes like the enhanced instruction set and a *SAM*-aware cache implementation. The cache implementation is based on the design presented in chapter 11, but implements only basic security functions like encryption, decryption and verification and lacks prefetching support.

### 14.1.1 Development Hardware

Since the cache with its cryptographic units and the enhanced processor is very large, a corresponding FPGA is required. Due to cost constraints, only two development boards, one with a XILINX Virtex 4 (XC4VLX60[1]) and one with a Virtex II (XC2VP50[2]), were available. Both of them with 59,904 and 53,136 LUT's[3], respectively, are too small for both the cache and the processor. Therefore, they have been connected using a high-speed LVDS[4] connection and the busses have been split using dedicated bus proxies (cf. figure 14.1). Note that due to the additional latencies caused by the interconnection, the resulting implementation could not be used for benchmarking purposes. Furthermore, the cache uses SDRAM provided on the development board with a cycle time of 90 ns to store all cache lines. Hence, main objectives of the implementation are overhead estimations and feasibility studies. Table 14.1 shows the required Lookup-Table Entries (LUT) required for the modified processor compared to the original LEON design. It can be observed that all *SAM*-specific changes result in approximately 62% more LUT's, mainly caused by the register protection. The required LUT entries have been used for comparison, since the internal architecture of both FPGA's differs, but for both FPGA's the LUT architecture is similar. Please note that the number of LUT's needed for each part of the architecture may vary slightly between different configurations, because any change in the configuration (for example when changing the number of cryptographic units) results in different optimization strategies. Therefore, the presented results can

[1] `http://www.xilinx.com/publications/matrix/matrix_V4.pdf`
[2] `http://www.xilinx.com/publications/matrix/virtexmatrix.pdf`
[3] Lookup Table.
[4] Low Voltage Differential Signaling, defined in the ANSI/TIA/EIA-644 standard.

| | *SAM*-LEON | LEON | *SAM*-Overhead |
|---|---|---|---|
| MMU | 4445 | 3584 | +24% |
| - D-Cache Controller | 1456 | 1365 | +6% |
| - I-Cache Controller | 648 | 677 | -4% |
| - AMBA Interface | 169 | 146 | +15% |
| Integer Unit | 2054 | 1983 | +3% |
| Integer Unit Extension | 405 | – | – |
| Register File | 25 | 14 | +78% |
| Register File Protection | 1528 | – | – |
| Tamper Detection | 60 | – | – |
| L2 Cache Configuration | 842 | – | – |
| Board Communication, Busses | 1270 | 180 | +600% |
| Total | 12902 | 7949 | +62% |

Table 14.1: Comparison of LEON and *SAM*-LEON

| Configuration Parameter | Value |
|---|---|
| Register Windows | 8 |
| TLB Entries per TLB | 8 |
| TLB Replacement Strategy | LRU |
| Sets per L1 Cache | 4 |
| Lines per L1 Set | 4 |
| Cache Line Size | 32 Bytes |
| L1 Cache Line Replacement Strategy | Random |

Table 14.2: LEON Configuration

be used only as a rough size estimation. In the following the whole model and the changes are described in detail. Figure 14.1 gives an overview of the placement of the different components on both FPGA's. Note that or simplicity's sake, in this figure only the interconnection between both boards is given, but no interconnection between internal components.

### 14.1.2 Processor

The processor model is based on the LEON 2 [26] SPARC V8 [85] processor model. The LEON design provides an execution pipeline with five stages[5], an Integer Unit (IU), virtual memory support, L1 data cache, L1 instruction cache and external peripheral devices like memory (RAM and ROM). The L1 caches are implemented using the virtually tagged/virtually indexed design [21]. The TLB is located between the L1 caches and memory. All external devices like memory are connected by AMBA [1] (Advanced Microcontroller Bus Architecture) and APB (AMBA Peripheral Bus) busses.
In the following the required changes for the *SAM* architecture are discussed. The configuration parameters used for synthesizing are listed in table 14.2.

[5]IF: Instruction Fetch, DR: Instruction Decode and Register Fetch, EX: Execute, ME: Memory Access, WB: Register Write-Back.

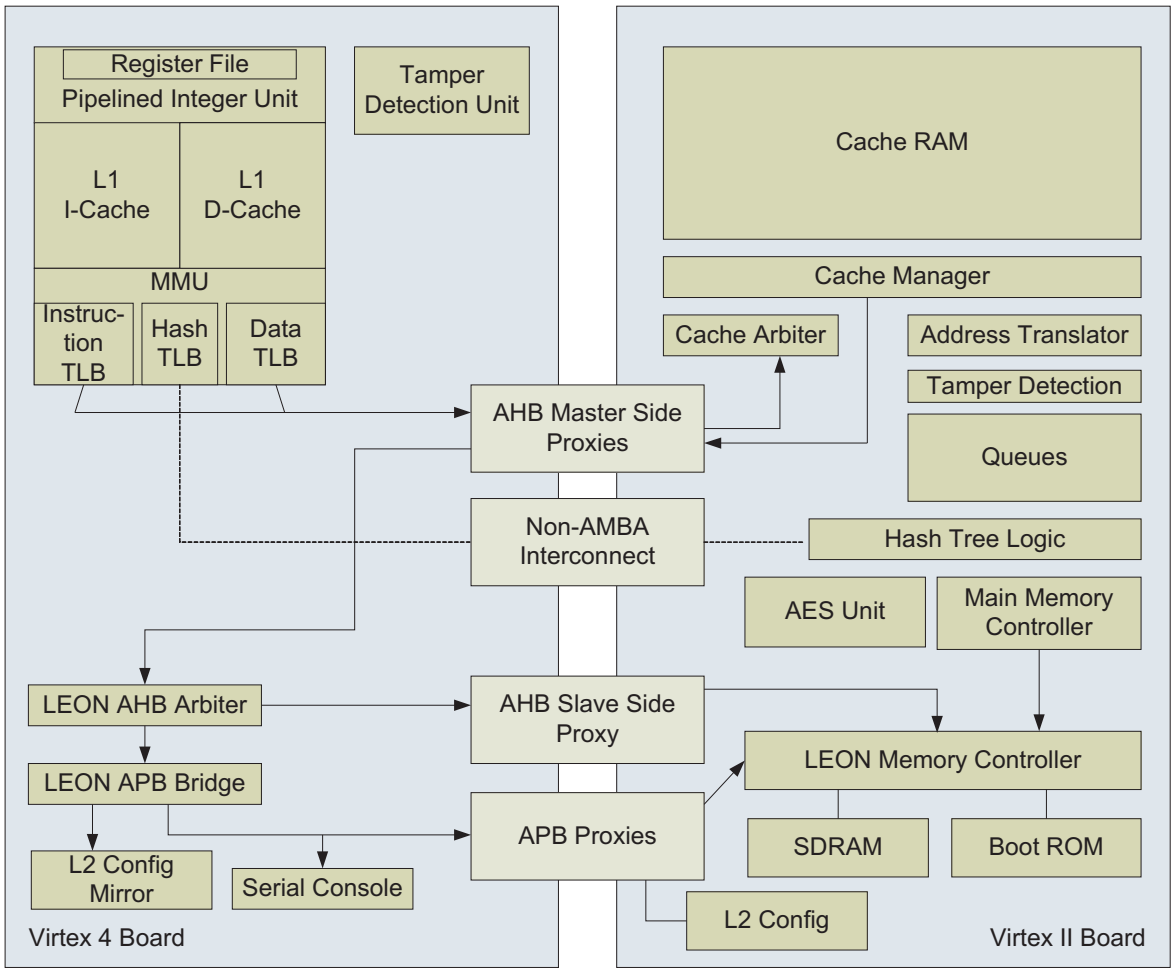



Figure 14.1: Placement of logical units on both FPGA's

#### 14.1.2.1 Integer Unit

The Integer Unit has been enhanced to support the additional three instructions *rprot*, *copybits*, and *st*, and the supervisor mode handling has been modified as described in section 9.1.

Since the changes on the protected TRAP mode should be visible for the instruction directly executed after *st*, it is required to stall the pipeline when *st* has been decoded, because any change on `PT` takes place while staying in the EX phase. Without a stall the old setting of `PT` would still be effective when the next instruction enters its DR phase and protected registers may not be accessible.

The instruction *rprot* has been implemented to suppress register protection flag updates for the directly following instruction resulting in the described delay slot.

The register protection mechanism has two properties: First, it has to store the value of `PI` for each register and second, all protected registers have to be cleared during context switches. The first property can easily be implemented with an additional bit associated with each physical register. The second one could be implemented by physically writing a zero to each register, but this would result in a huge hardware overhead. Therefore, the proposed solution assigns only two additional bits to each register: the *protected* bit holds the protection information and the *zero* bit can be used to mark a register as cleared. On each context switch the zero bit is set for each register marked as *protected* and left untouched otherwise and all *protected* bits are cleared. This implementation requires only two additional bits for each physical register and can clear all protected registers within one clock cycle.

Of course, the register file has been adjusted to honor the *protected* and *zero* bits. The protected bit is checked in parallel to register operation and the zero bit is connected to a multiplexer which returns the current register value or zero on each read access. Register protection is quite costly, as can be seen in table 14.1. The size of the enhanced register file with its protection mechanism causes the major overhead of the whole processor design. Therefore, further optimizations should be applied to the register handling. One possible optimization would be to not return zero on unprotected read access but to report this read to the tamper detection unit as discussed in section 9.1. Processor synthesizing has shown that the register file contains the critical path in the processor design halving the maximum processor speed from 98,9 MHz for the original LEON to 51,6 MHz for *SAM*-LEON.

However, it should be noted that this implementation – in contrast to the original LEON design – has not been further optimized for speed or size. Therefore, further optimizations could result in a smaller and faster implementation.

#### 14.1.2.2 L1 Caches

The main change to the L1 data cache is the additional *unprotected* bit indicating if the corresponding line has been read by a protected instruction or not. This additional bit increases the usage of RAM blocks slightly, as can be seen in table 14.3. The table lists the allocated block RAM for the unmodified LEON processor (LEON), for a *SAM*-aware L1-data cache (*SAM* L1-D), and for a *SAM*-aware L1-data and instruction cache (*SAM* L1-D+I). The L1 data cache configuration remains always the same, as listed in table 14.2. Only the L1 instruction cache has been adjusted in terms of lines. It can be seen that the additionally required block RAM for the *unprotected* bit is small compared to the original LEON design.

| Cache lines per L1-I set | LEON | *SAM* L1-D | *SAM* L1-D+I |
|---|---|---|---|
| 4 | 23 | 28 | 32 |
| 8 | 32 | 36 | 36 |
| 16 | 48 | 52 | 52 |

Table 14.3: FPGA-RAM usage

### 14.1.3 L2 Cache

The L2 cache implementation is based on the design described in chapter 11. The first major change compared to a standard L2 cache affects the implementation of the bus connecting both L1 caches with the L2 cache. This bus has been extended to pass the following additional information to the L2 cache:

- The virtual base address of the corresponding cache line.
- The context number of the context reading or writing the line.
- A flag representing the protection level of the accessing instruction.

This information is transferred over dedicated bus lines in parallel to the address and data. It is not required to pass information about the memory area (protected or encrypted memory) to the L2 cache, because the cache itself has access to the tables defining these areas and can get this information based on the virtual address. To speed up reading these addresses they have been stored on both FPGA boards where the configuration accessed by the processor (L2 Config Mirror, cf. figure 14.1) is implemented read-only. The tamper detection unit is directly connected with the L2 cache to be able to instantly act on detected tampering attempts by the L2 *Check Queue*.
Additionally, the cache has its own TLB, which is required to be able to transform the computed parent virtual hash tree addresses into their physical counterpart. This TLB directly accesses the page table in memory, that is the page table entries covering the hash tree cannot be cached. This design has been chosen to prevent possible deadlocks and to simplify cache design.
The L2 cache implementation for *SAM* is a completely new design. Therefore, the first implementation of the cache without *SAM* enhancements has been used as a reference to measure the overhead caused by *SAM*. The results based on the LUT usage are shown in table 14.4 for a cache with a size of 16k. Larger caches have not been synthesized due to the size constraints of the development boards. The results show that the AES unit in conjunction with the configuration unit occupies most of the additionally used lookup tables.

### 14.1.4 Cryptographic Units

The cryptographic units used in this work are based on standard cores. The RSA core is based on the Basic RSA Encryption Engine written by McQueen [55] and the AES unit has been written by the National Security Agency [64]. The sizes of both units are given in table 14.5. A comparison of these values with the one of the processor implementation in table 14.1 shows that the cryptographic units are much larger than the processor. Therefore, the *SAM* overhead without the cryptographic units is negligible compared to the overhead caused by the AES and RSA units, especially when taking into account that more than one AES unit

| | *SAM*-L2 cache | L2 cache |
|---|---|---|
| AHB[6] | 25107 | 15991 |
| Arbiter | 3 | 3 |
| Slave | 422 | 274 |
| Main Memory Controller | 54 | 57 |
| Cache RAM (with debugging unit) | 656 | 663 |
| Cache Manager | 5092 | 5379 |
| Queues | 9060 | 8115 |
| AES Unit with scheduler (encryption only) | 11031 | – |
| Hash Tree | 121 | – |
| Address Translator | 82 | – |
| Configuration unit | 5537 | 425 |
| Total | 42670 | 16480 |

Table 14.4: L2 cache LUT usage

| Unit | LUT's |
|---|---|
| RSA | 26533 |
| AES | 15987 |

Table 14.5: Size of cryptographic units

is typically required. However, the cryptographic units are not exclusively used by *SAM* and can be made available to user programs as general-purpose cryptographic units to reduce the *SAM*-only overhead.

## 14.2 Simulation Model

As described above, the VHDL model was not suitable for performance analysis and the cache too small to be useful. Therefore, a software simulation model has been developed, which has been used for all performance evaluations. The software simulation model can be split into two parts, a system emulator and a cache simulator. The system emulation is used to record a memory access pattern which is then passed to the cache simulator.

### 14.2.1 System Emulation Environment

At first the different components of the system emulator are described.

#### 14.2.1.1 Benchmark

For performance evaluation of computer systems benchmark programs are used. Several benchmarks with different purposes exist and they can be used for performance evaluation on different levels. Low level synthetic benchmarks like Dhrystone or Whetstone can be used to measure several processor-specific properties, but since they are highly specific, their expressiveness is limited. Therefore, today's most-used benchmarks are application benchmarks, because they allow to measure several parts of a computer system at the same time. A popular

application benchmark is the SPEC (Standard Performance Evaluation Corporation)[7] benchmark suite. Its performance evaluation is based on the runtime of several integer and floating point applications covering a wide range of applications. The SPEC benchmark mainly evaluates the performance of the processor and the memory hierarchy as well as the compiler. Other parts of a computer system like the disk or graphic performance have no or a negligible impact on the result.
Due to these properties, the SPEC 2001[8] benchmark suite has been used for the *SAM* performance evaluation described in the following. Since all benchmarks have to be executed by an operating system, a full system emulation including the modified operating system as described in chapter 10 is required.

#### 14.2.1.2 QEMU

All benchmarks are executed in a virtual machine emulating a SPARC-based computer with peripherals like hard disk, framebuffer and keyboard. This virtual machine is based on the free system emulator QEMU [4]. QEMU translates all instructions of the guest system into native assembler instructions of the host system. Hence, all timing and memory access information is lost. Therefore, QEMU has been extended to add special monitoring instructions during the translation step to record instruction execution and memory access when executing the translated code.
The current clock cycle has been restored by incrementing a counter before executing a simulated SPARC instruction. This results in a simulated instruction in each clock cycle (CPI[9] value of one) and hides any pipeline-related stalls. However, most modern processors with one pipeline can reach a CPI value close to one by using speculative execution and instruction reordering, for example. Furthermore, a "faster" processor reveals more additional latencies caused by the security-enabled cache.
QEMU is a system emulator and not a simulator. Hence, the behavior of the emulated system is emulated, but for example the emulated interrupts may occur at different times for concurrent runs of the same benchmark, because QEMU's timing is dependent on the host system's timing which is, for example, dependent on real hard drive accesses and general system load. While this is in general no problem, it may affect the generated benchmark results because the recorded memory access pattern and timing may change slightly an subsequent runs. Since the recorded data is used as input for the cache simulation, only one memory pattern for each benchmark run has been created and then used for all further cache simulations. The Linux kernel used to execute all benchmarks is described in section 14.3 and contains *SAM*-specific modifications. Hence, even for simulations where the *SAM* extensions in the cache have been disabled a *SAM*-enabled Linux kernel has been used.
Due to the long runtime (several hours up to one day just for recording the access pattern) and the resulting large trace files, not all data for each benchmark has been recorded. For each benchmark the first $2^{32}$ simulated instructions have been skipped before recording data to the trace file. This has the following reason: These instructions basically correspond to the instructions executed to load and initialize the benchmark. Hence, they are more or less equal for each executed benchmark and not of interest for further evaluations. Recording has been

[7] `http://www.spec.org`
[8] The 2001 suite has now been deprecated by the 2006 release, but the new release was not available by the time of writing the emulation environment.
[9] Cycles Per Instruction.

stopped after executing a given number of instructions in user mode resulting in a total of approximately $2^{32}$ instructions.
For all benchmark simulations, all program data is located between the virtual addresses `0x70000000` and `0xefffffff` and has been encrypted. The first 64 KByte of the operating system (`0xf0000000`–`0xf000ffff`) are protected but unencrypted. The hash tree starts at address `0x1aaaaab0`. Furthermore, the Linux kernel has been adjusted to allocate memory for the heap within the encrypted memory region starting at address `0x80000000` instead of the default address `0x50000000`.
In this paper, the single and multitasking behaviors of a *SAM*-enabled system have been analyzed. Therefore, many simulations have been evaluated using a single executed benchmark program as well as simulations with several benchmark programs running in parallel. Typically, the same SPEC benchmark is executed in parallel on multiprocessor machines. But some benchmarks have a huge memory footprint and paging of protected programs is currently not supported by the modified Linux kernel. Hence, each simulation consists of the execution of one instance of a primary benchmark (out of the SPEC suite) and a number of instances of a secondary benchmark. Due to its reasonable memory footprint, *eon* has been selected as a secondary benchmark.

#### 14.2.1.3 Trace File

The trace file stores the following data:

- The current clock cycle.
- The current context.
- The type of memory access: instruction, data, raw (by external devices) or I/O[10].
- The direction: read or write.
- The number of bytes accessed.
- The occurrence of a TRAP or context switch.
- The virtual (if available) and physical addresses.
- The program counter value (if available[11]).

To save space all this data has been compressed using a hybrid approach. The trace file consists of a concatenation of two possible data entries. The *full* data entry contains all information given above and the *partial* one is used for relative data, that is for example the difference between the last and the current program counter value. In cases where the partial entry cannot be used because it does not contain less frequently changed entries like the context, the full entry has to be used. The full data entry occupies 28 bytes, the partial one between two and 14 bytes. The resulting data is then compressed using the bzip2 algorithm to further remove redundant information.
Table 14.6 lists the sizes of the trace files for all single task benchmarks used in this work. It can be seen from the given values that the memory access pattern differs for all benchmarks. The benchmark *mcf* has the largest trace file and this is caused by a huge number of mostly random memory accesses, as can be seen later in this work.

[10] Input/Output.
[11] Raw access is caused by hardware and therefore no program counter is available.

| Benchmark | Size in bytes | Number of instructions |
|---|---|---|
| bzip2 | 2232067571 | 4117835299 |
| crafty | 2240613246 | 4177992027 |
| eon | 1743970187 | 4297527575 |
| gap | 1752094717 | 4260858498 |
| gzip | 1438749325 | 4089676301 |
| mcf | 6111031769 | 4178233635 |
| parser | 2416248972 | 4290269114 |
| perlbmk | 2661729920 | 4257584061 |
| twolf | 3729365651 | 4148162172 |
| vortex | 2873667896 | 4418790406 |
| vpr | 3296415059 | 4121323314 |

Table 14.6: Trace file information

| Cache property | Value |
|---|---|
| L1 placement | direct mapped |
| L1 line size | 32 bytes |
| L2 placement | LRU, 4-way-set |
| L2 line size | 64 bytes |
| TLB Entries | 64 |
| L1 Clock | 100% of main clock |
| L2 Clock | 33% of main clock |
| Queue Clock | 50% of main clock |
| Dictionary Entries | 4 |

| Bus | Width | Clocking |
|---|---|---|
| L1 ↔ L2 cache | 128 bit | 100% of main clock |
| to memory | 64 bit | 20% of main clock (+70 main clock cycles latency) |
| L2 cache ↔ Queues | 128 bit | 50% of main clock |
| to AES units | 128 bit | 50% of main clock |

Table 14.7: Cache properties

| Name | L1 size | L2 size | AES units | *Check Queue* and *Memory Write Queue* entries |
|---|---|---|---|---|
| 8-256 | 8k | 256k | 5 | 5 |
| 16-1024 | 16k | 1024k | 5 | 5 |
| 32-2048 | 32k | 2048k | 5 | 5 |

Table 14.8: Cache configurations

### 14.2.2 Cache Simulator

The *SAM* cache simulator simulates an L1 data and instruction cache as well as the L2 cache with all security-related and performance-related extensions described in this thesis to compute the number of simulated clock cycles for these operations. Instruction and data access are passed to the corresponding L1 cache and external device access is simulated by occupying the memory bus. The simulator is based on a simple discrete event simulation and uses a simulated clock signal for transitions between its internal states.
The cache simulator is fully configurable in terms of cache sizes, bus widths, number of queue entries and their thresholds, clock divisors to simulate different clock rates for buses and components like the caches, memory latencies or hashing algorithms. The clocking of the L1 cache is used as a reference and all other components are clocked with divisors based on this clock rate. For each generated trace file, a set of different cache configurations has been simulated to obtain the overall number of simulated cache clock cycles needed for all cache operations. This set includes a configuration without security extensions which is further used as a reference for the speedup computation.
The trace file does not contain any hash-related data due to the lack of hash tree handling within the operating system. Therefore, the cache simulator provides additional page-mappings to unused physical pages on the fly for hash values.

#### 14.2.2.1 Configurations

Table 14.7 lists the basic configuration used for all simulations. The bus width between the caches has been optimized in terms of the block size of both L1 caches given with the sizes of the L2 cache in table 14.8. The cache sizes have been chosen to represent a small, medium and a large cache. Much larger caches are common, too, but the performance of the simulation environment does not allow to simulate as many instructions as required to get reasonable results for very large caches in an acceptable time frame.

### 14.2.3 Limitations

Each simulation model simplifies the behavior of a real processor cache combination. The model used in this thesis has the following limitations:

- No feedback: On a real processor latencies caused by the cache result in pipeline stalls resulting in a different timing and a different memory access pattern. This cannot be simulated with the used model.

- QEMU emulates disks and other peripheral components but not their latencies. For example, the interrupt indicating that a hard disk read operation has been completed is directly triggered after sending the read request to the emulated disk. Therefore, peripheral devices are much faster than their real counterparts. This may additionally affect the timing. But a generally “faster” simulation environment puts more pressure to the cache system, since external latencies cannot be used to hide cache operations like verifications. Therefore, the performance on a real processor may be better than on a simulated one.

- The bus model used within the cache simulator requires at least one clock cycle for each transaction. This is in contrast to the LEON processor, whose busses are able to react

on control signals within one clock cycle. However, the busses between the simulated processor and the L1 caches are designed to be able to transfer data within each clock cycle.

- Access to all busses except the bus between the L1 caches and the L2 cache is granted based on a fixed arbitration scheme. That is, if two master devices are requesting the bus at the same time, always the same master device will get the bus.
- All communication between components of the cache is taking place over busses implemented with the same bus model. This is in contrast to the VHDL implementation, where point-to-point connections are implemented using a more simple and partly faster design.
- Paging of protected programs is not supported.

These simplifications are required to simulate a huge number of clock cycles. For example, with the much more accurate VHDL model the simulated clock speed would be lower than 10 Hz, which is much slower than the clock speed of the cache simulator with approximately 180000-250000 Hz based on the simulated workload. In general, most simplifications used for the simulation environment result in a "slower" simulated environment and therefore the computed results can be considered as worst-case data.

## 14.3 Operating System

Vital parts of the SPARC Linux operating system have been adjusted for the *SAM* architecture. The following changes have been implemented:

- *TRAP handling*: For *SAM*, the whole TRAP handling has been rewritten, because the Linux kernel contains a lot of architecture-specific code used for different SPARC processor models. Furthermore, all protected program code has been moved to the beginning of the virtual kernel address space. Due to alignment restrictions and the ELF header of the kernel, the TRAP table begins at address `0xf0004000` and all protected TRAP-handling code ends at address `0xf00062e0`. Hence, the TRAP-related changes occupy approximately 9000 bytes thus leaving enough free protected address space for additional protected kernel code. In total, 2045 lines including comments of kernel code have been modified or added.

  The performance of the new implementation has been measured using the simulation environment described above. To measure the kernel overhead 10 simulations using a standard kernel and 10 simulations using the *SAM*-kernel have been started and terminated after executing a fixed number of user space instructions to compute the speedup. The results have shown a speedup of 1. Hence, the *SAM*-related changes did not result in any measurable overhead, because with the rewrite some small performance optimizations could be applied to the TRAP handlers hiding the additional *SAM* instructions.

- *ELF loader*: For the ELF loader 675 lines including comments of kernel code have been modified or added. This includes the hash tree mapping routines required to load all pages containing hash values for all protected pages.

  The performance of the modified loader has not been further analyzed, because the main delay when loading a protected program is caused by the RSA unit.

| Parameter | Range | Description |
|---|---|---|
| Idle | 0–12 | Determines the number of idle cycles of the L2 cache to wait before trying to predict and fetch data. |
| NoCheck | yes/no | Pass fetched cache line to *Check Queue* for verification? |
| AllowStrideNull | yes/no | Whether a stride of 0 (accessing the same address) is predicted or not. |
| Buffers | 1–32 | Number of stride buffers. |
| AddressesPerBuffer | 1–16 | Maximum number of pre-fetched cache lines per stride buffer. |
| PriorityDecrementCycles | 1–100 | Number of cycles after which the confidence of an entry is decremented. |
| MaxConfidenceBits | 1–4 | Number of bits used for confidence information. |
| StridePrediction | yes/no | Enables or disables stride-prediction. |
| StrideBits | 1–16 | Number of bits used to index Stride table. |
| MarkovPrediction | yes/no | Enables or disables Markov-prediction. |
| MarkovAssociativity | 1–4 | Associativity of the Markov predictor. |
| MarkovBits | 1–16 | Number of bits used to index Markov table. |
| TCPEnable | yes/no | Use TCP instead of classical Markov predictor. |
| TCPTagsPerSet | 1–4 | Number of tag entries per set (associativity). |
| TCPTagbits | 1–8 | Number of bits used from the tag address as an index. |
| TCPSetbits | 1–8 | Number of bits used from the set address as an index. |

Table 14.9: Parameter ranges and descriptions used as a genome

The current protected code base in the kernel is much smaller than the proposed 64 kByte, but not all required parts have been currently implemented. However, even with a secure implementation of multi-threading and signal handling support a protected codebase of 64 kByte is likely to be sufficient. Page unloading and other currently not implemented functions do not require protected memory.

## 14.4 L2 Prefetcher

The usage of data prefetchers adds more parameters to a cache system like the number of entries of the stride predictor or the associativity of a Markov predictor. Hence, the optimal values are hard to find and highly workload-dependent. In order to find good parameters, in this work an evolutionary approach using the JavaEvA toolkit [87] has been used. The number of simulated clock cycles of a given workload has been used as a fitness function. Initial tests have shown that the *MutationMSRGlobal* mutation algorithm in conjunction with the values $\lambda = 100$ and $\mu = 20$ provides good results[12]. Table 14.9 lists the parameters

[12]Here $\mu$ denotes the number of selected best individuals from the current population of $\lambda$ individuals to generate the next generation.

(decision variables) which have been mutated. Note that the parameters MarkovPrediction, StridePrediction and TCPEnable have been set to a fixed value for each run of JavaEvA.

## 14.5 Cache-Memory Overhead

In this subsection the hardware overhead required for *SAM's* additional protection is presented. Table 14.10 sums up the cache-memory overhead for a sample cache system. The overhead for the L1 cache is negligible. For the L2 cache the amount of required additional TAG RAM is very high due to the TAG part of the virtual address, the context dictionary, and the context number. However, when considering the whole cache with all user data, an overhead of approximately 9% is reasonable.

## 14.6 Simulation Results

This section provides a performance analysis of the *SAM* architecture. Various cache and Linux configurations have been analyzed to investigate *SAM* from a performance-oriented point of view.
The results of all simulations are listed in appendix B. More information about the simulation environment and the abbreviations used in the appendix is provided in appendix A. In the following, references to the tables in appendix B with the results used to create the figures in this section are provided in footnotes.
Note that instead of the geometric mean of all twelve SPEC benchmarks the results for each single benchmark run are given. This illustrates that each benchmark has a different memory access pattern resulting in very much different results. However, a detailed analysis of the memory access pattern of each executed benchmark are going beyond the scope of this work. Therefore, no detailed analysis for each benchmark executed with a particular cache configuration can be provided.

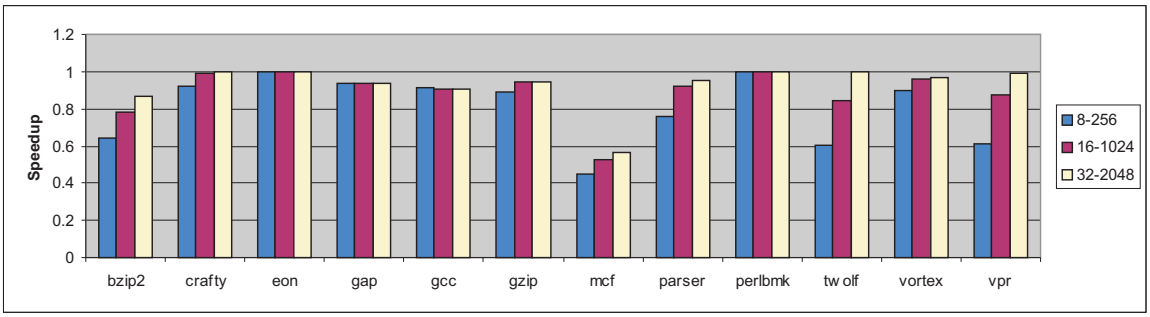
### 14.6.1 L2 Cache with speculative Execution

Figure 14.2a[13] shows the speedup[14] for all benchmarks executed in a single task environment and for the three standard cache sizes.
The performance of the larger caches for all benchmarks except for *mcf* is very good with a typical slowdown between 0.8 and 1. Some benchmarks, like *eon*, are executed with negligible slowdown. Of course, for smaller caches the slowdown increases, since a smaller cache is not able to hide the additional overhead as well as a larger cache. However, small L2 caches with only 256 kByte are no longer state of the art: most second-level caches of modern desktop processors provide between one and two megabyte RAM and, for these caches, *SAM* performs well.
Nonetheless, the slowdown can be worse if memory access patterns are comparable to those of the *mcf* benchmark. Figure 14.2b shows the percentage of cache misses with respect to the total number of cache accesses (shown in figure 14.2d for the second-level cache). It is obvious that the reason for the low performance is the huge number of cache misses for the

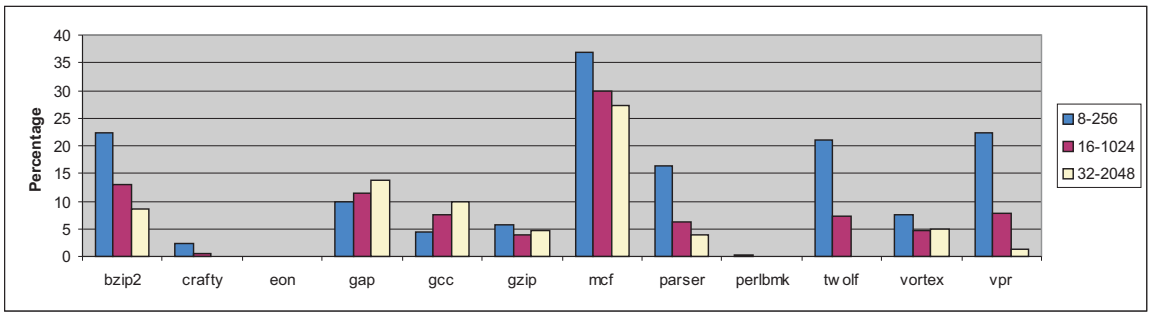
[13] Simulations shown in sections B.1.2, B.1.3, and B.1.4, column normal.
[14] The computed speedup is mostly smaller than one due to the *SAM* overhead. Therefore, it is sometimes denoted as slowdown.

| Component | Original | | *SAM* Overhead | |
|---|---|---|---|---|
| | total | per line | total | per line |
| *16K L1 Data Cache, direct mapped, write-back* | | | | |
| Cache Size | 16384 byte | 32 byte | – | – |
| Cache Lines | 512 | – | – | – |
| Dirty + Valid Bits | 1024 bit | 2 bit | – | – |
| Virtual Address | 9216 bit | 18 bit | – | – |
| Context Number | 4096 bit | 8 bit | – | – |
| Unprotected Bit | – | – | 512 bit | 1 bit |
| Total (administrative) | 14336 bit | 28 bit | (+4%) 14848 bit | 29 bit |
| Total | 145408 bit | – | (+0.3%) 145920 bit | – |
| *16k L1 Instruction Cache, direct mapped, write-back* | | | | |
| Cache Size | 16384 byte | 32 byte | – | – |
| Cache Lines | 512 | – | – | – |
| Valid Bit | 512 bit | 1 bit | – | – |
| Virtual Address | 9216 bit | 18 bit | – | – |
| Context Number | 4096 bit | 8 bit | – | – |
| Total | 145152 bit | – | – | – |
| *1 M L2 Cache, 4-way-set-associative, write-back, LRU strategy* | | | | |
| Cache Size | 1048576 byte | 64 byte | – | – |
| Cache Lines | 16384 | – | – | – |
| Sets | 4096 | – | | – |
| Dirty + Valid Bits | 32768 bit | 2 bit | – | – |
| LRU | 20480 bit | 5 bit | – | – |
| Physical Address | 229376 bit | 14 bit | – | – |
| Virtual Address | – | – | 229376 bit | 14 bit |
| Context Number | – | – | 131072 bit | 8 bit |
| Unprotected Bit | – | – | 16384 bit | 1 bit |
| Decrypted Bit | – | – | 16384 bit | 1 bit |
| Checked Context Bits | – | – | 65536 bit | 4 bit |
| Total (administrative) | 282624 bit | 21 bit | (+162%) 741376 bit | 28 bit |
| Total | 8671232 bit | – | (+9%) 9412608 bit | – |

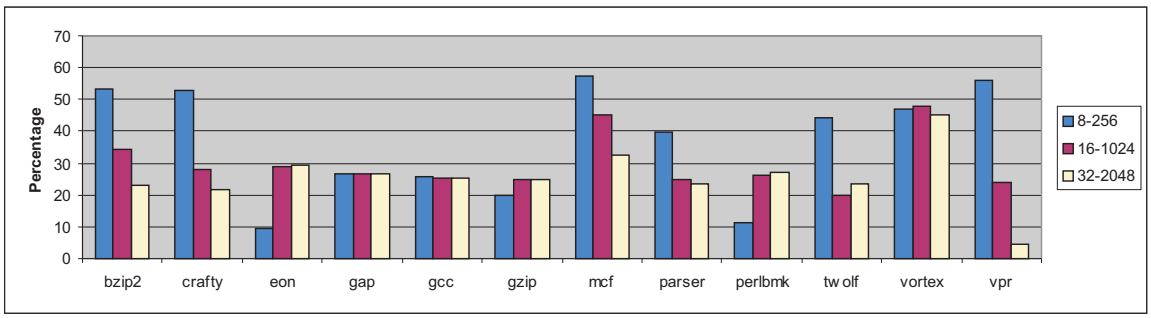
Table 14.10: Cache-memory overhead (without prefetching capabilities)



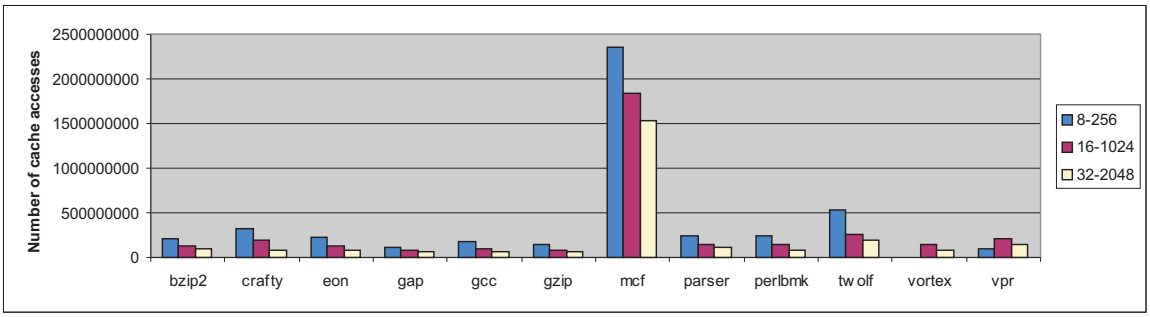

(a) *SAM* speedup




(b) L2 cache misses

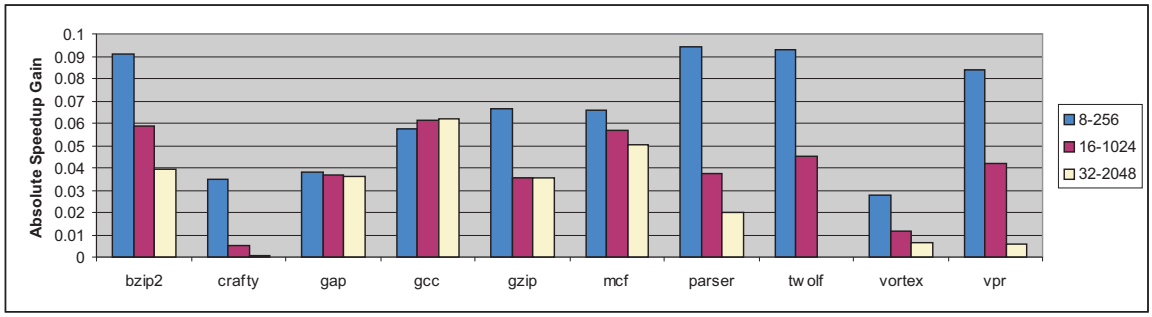



(c) L2 Hash misses

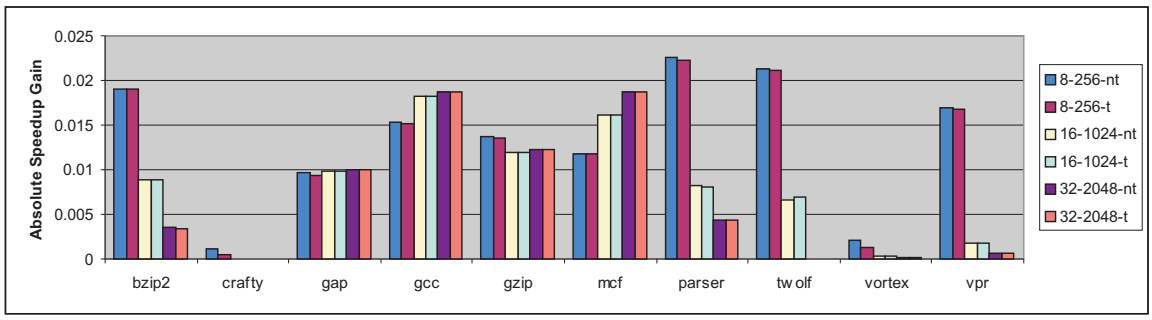



(d) L2 cache accesses

Figure 14.2: Cache simulation results




(a) Speculative execution of instructions




(b) Prevention of subsequent unverified fetches

Figure 14.3: Speculative execution

*mcf* workload. Workloads like *eon* fit into the second-level cache, resulting in a very good performance. Thus, the impact of changes to the cache configuration for workloads like *eon* or *perlbmk* is expected to be low and, therefore, the results for these benchmarks are omitted in some of the following figures, but full results are always available in the appendix.
For many misses of encrypted cache lines the overhead caused by decryption is low, because the corresponding hash line is already cached for up to 80% of all hash line accesses, as can be seen in figure 14.2c. The figure shows (in percent) how many of all hash value fetches result in a miss. For most benchmarks and cache configurations, this value is between 5% and 55%. This indicates that the usage of counter mode encryption with hash values used as a counter can hide the additional latency caused by reading the hash values in up to 95% of all encrypted cache line misses.

### 14.6.2 L2 Cache without speculative Execution

Figure 14.3a[15] shows the importance of speculative execution of instructions. It shows the absolute speedup gain of the normal, speculative execution of instructions compared to a nospeculative cache configuration that verifies all fetched instructions prior to their execution. For example, the 8-256 result has been calculated by subtracting the speedup of the normal 8-256 cache configuration from the 8-256 nospeculative configuration. In the following these differences are called “absolute speedup gain”.
Especially for smaller caches, it can be seen that speculative execution can speed up the

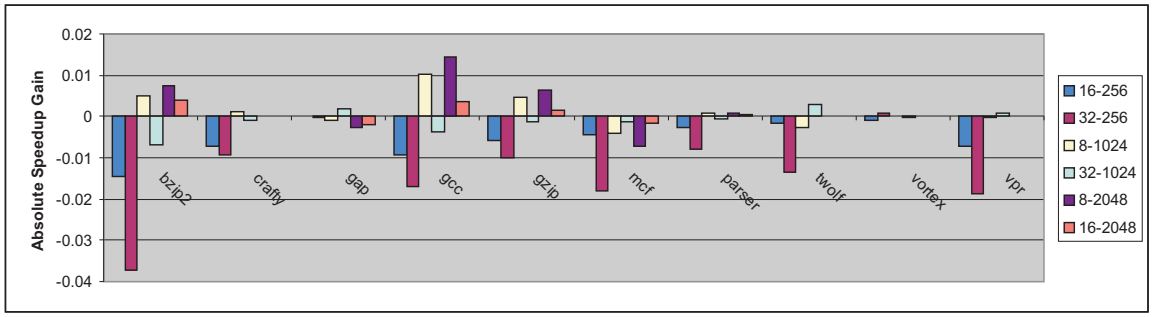
[15]Simulations shown in sections B.1.2, B.1.3, and B.1.4, columns normal and nospeculative.

executed benchmarks by approximately up to ten percent points, which shows the importance of speculative execution.
Figure 14.3b[16] shows the absolute speedup gain when deactivating the additional measures required to prevent the memory decryption attack described in section 7.10.1. The figure shows the difference between two configurations with lower security compared to the standard *SAM* cache configuration. The *-nt set of simulations allows subsequent cache line fetches without prior verification of already fetched lines, and the *-t configuration protects against the memory decryption attack by preventing subsequent instruction fetches without prior verification but allow them for data cache accesses. The first observation is that the results for both less secure configurations are close together. Hence, the protection against the memory decryption attack can be realized with nearly no additional performance penalties. However, a modified attack, which could be based on data fetches, cannot be prevented. For example, an attacker could try to modify data which results in control flow changes if this data is used for conditional branches to analyze the program behavior. Even if these attacks are less likely to succeed, they have to be prevented as well. Hence, *SAM* prevents subsequent cache line fetches for both data and instruction cache accesses in its default configuration. The performance impact is noticeable, but with a speedup decrement of approximately up to 0.02 still acceptable.

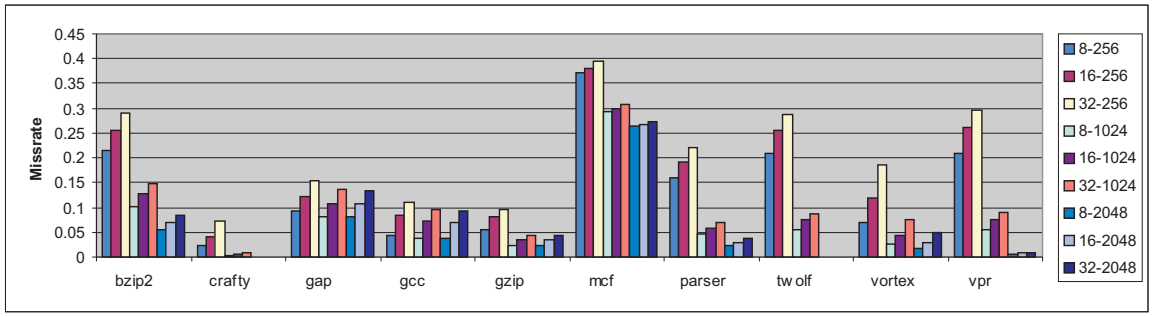
### 14.6.3 Cache Size Variations

In the standard configuration three cache sizes have been selected. To measure the impact of the size of the L1 caches several benchmarks with other cache size configurations have been evaluated. Figure 14.4a[17] shows the effect of different L1 cache sizes for a fixed L2 cache size. The results show the absolute speedup gain with the standard cache configuration with the same L2 cache size as a reference (for example, the 16-256 result has been calculated by subtracting the 8-256 speedup from the 16-256 speedup). As a matter of course the size of the L2 cache has a much larger impact on the slowdown than the size of the L1 cache. However, increasing the size of the L1 cache for a fixed L2 cache size does not always result in a better speedup. For some benchmarks like *bzip2* or *mcf* the speedup decreases when increasing the L1 cache size. This observation corresponds with the miss rate of the L2 cache as shown in figure 14.4b. This result shows that larger L1 caches do not always result in a better overall performance. According to [33], the miss rate of the L2 cache increases when the size of both L1 caches converge at the size of the L2 cache. In case of the simulated cache configurations the L2 cache is only four times larger than both L1 caches in the 32-256 configuration, and the miss rate of the L2 cache increases even if the number of L1 misses decreases for larger L1 caches, as can be seen in figure 14.4c. The reason for this is obvious: if data can stay longer in one of the L1 caches the probability of a cache line replacement of the same data in the L2 cache increases. This results in another miss in the L2 cache when the L1 cache writes back the dirty line resulting in two L2 misses for a L1 miss. A write through L1 cache can prevent many of these misses, but as stated above, the write-back strategy has been selected to relieve the L2 cache. Figure 14.5a[18] proves this by showing the absolute speedup gain for the smallest L2 cache configurations for a write through L1 cache configuration with the normal configuration as a reference. The speedup for the write through

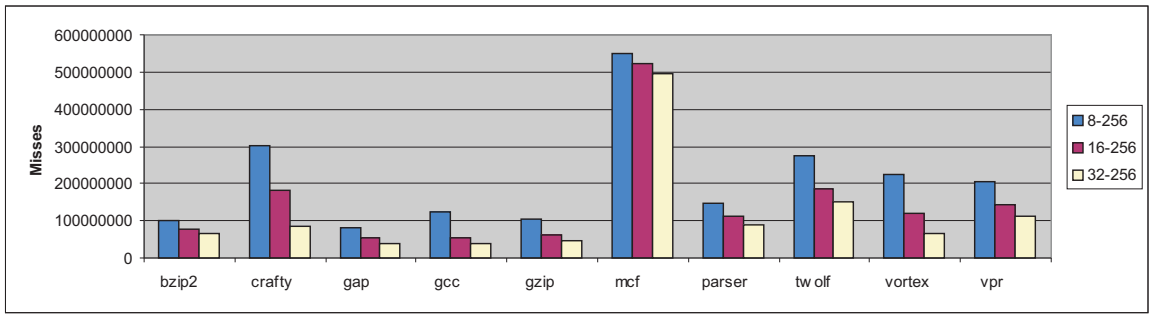
[16] Simulations shown in sections B.2.2, B.2.3, and B.2.4, columns notainted (*-nt) and tainted (*-t).
[17] Simulations shown in section B.3.1.
[18] Simulations shown in sections B.4.10, B.4.4, and B.4.7.



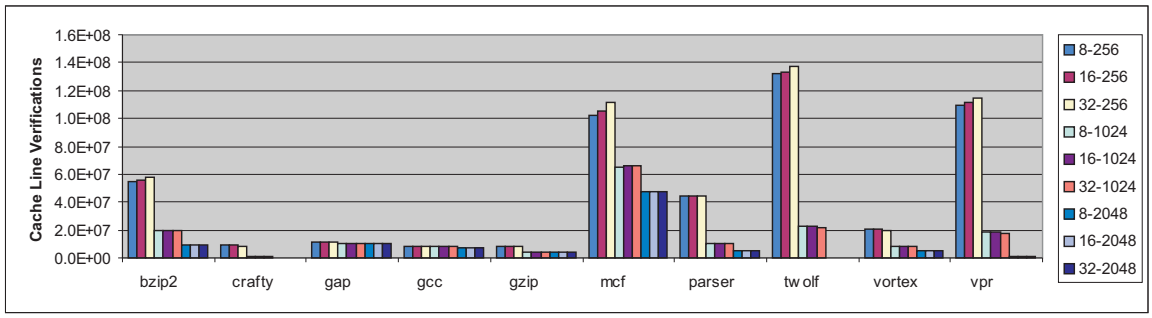

(a) Different cache sizes




(b) L2 miss rate

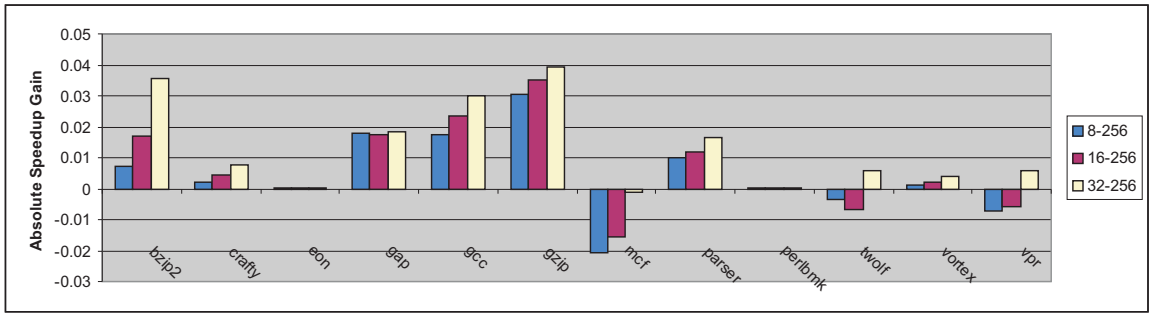



(c) Total number of misses of both L1 caches

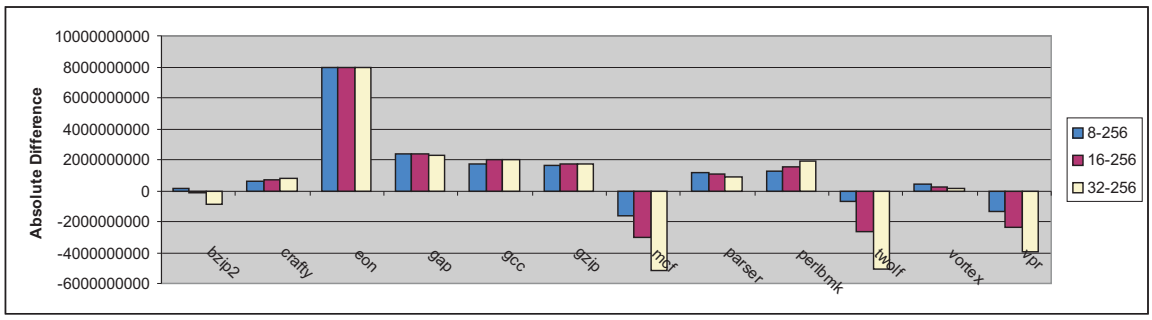



(d) Total number of cache line verifications

Figure 14.4: Different cache sizes




(a) Different L1 cache sizes




(b) Additional simulated clock cycles

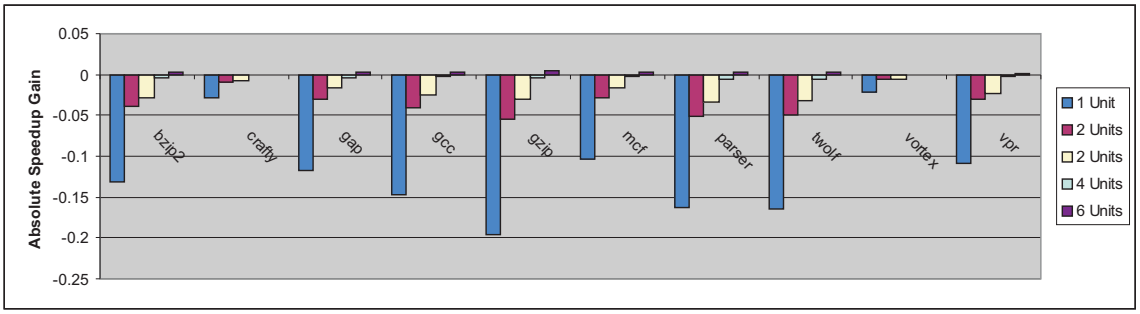
Figure 14.5: Write Through cache configuration

cache configuration is higher for all benchmarks except for *mcf*, *twolf* and *vpr*, but the overall number of simulated clock cycles is mostly larger for a write through cache, as can be seen in figure 14.5b[19]. This figure shows how many additional clock cycles have been executed for the write through cache configuration. Therefore, the *SAM* penalty is smaller for write through L1 caches, but in general for most executed benchmarks write through L1 caches perform worse than write-back L1 caches.
In case of a *SAM*-enabled cache the probability of cache line replacements further increases due to the additional hash values required for verification, and their cost is much higher due to their verification. Figure 14.4d[20] shows the number of cache line verifications and it can be seen that this number increases considerably with larger L1 caches at least for the *bzip2*, *mcf*, *twolf*, and *vpr* workload.

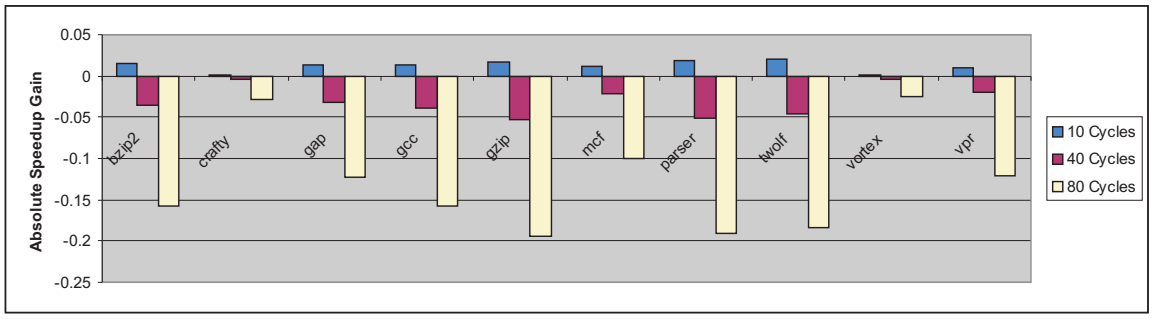
### 14.6.4 Cryptography

Here the impact of the cryptographic units used in the L2 cache are evaluated. This includes the number of AES units, their encryption performance, and the hashing performance. Due to the huge number of results, in the following only the speedup for the 8-256 cache configuration is given because this configuration stresses all cryptographic units more than both other configurations.

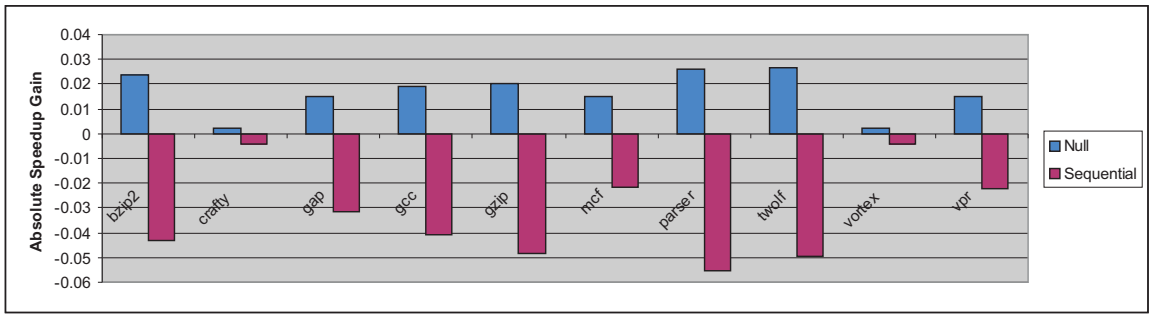
[19]In this figure the number of simulated clock cycles for the *mcf* workload has been divided by 10 to be able to show more details for the other workloads.
[20]In this figure the number of misses for the *mcf* workload has been divided by 10 to be able to show more details for the other workloads.




(a) Different numbers of AES units




(b) AES unit performance




(c) Impact of hashing algorithm

Figure 14.6: Performance of cryptographic-related parts of the L2 cache

#### 14.6.4.1 Number of AES Units

As can be seen in table 14.5, the AES unit has a huge impact on the overall size of the processor and cache. Hence, it is desirable to reduce the impact by reducing the number of AES units. Figure 14.6a[21] shows the absolute speedup gain for different numbers of AES units for a 8-256 cache configuration compared to the standard configuration with five units. With only one AES unit the performance degrades up to 20 percent points (*gzip* benchmark) compared to the configuration with six AES units. For all other benchmarks the degradation is smaller, but in most cases around 10 percent points. Only the small benchmarks like *eon* do not suffer from this limitation. However, the performance gain for more than two AES units is much smaller than the additional effort in terms of chip space. Hence, a small *SAM*-enabled processor could provide a good performance even with only two AES units.

#### 14.6.4.2 Performance Variations

The impact of a fast AES implementation has been analyzed in figure 14.6b[22] for the 8-256 cache configuration. In the default configuration the AES unit needs 20 cycles to perform an AES encryption and this configuration has been compared with faster (10 cycles) and slower (40 and 80 cycles) ones. Here only the slowest configuration has a notable impact on the slowdown. Hence, slightly slower AES units which are possibly smaller in terms of chip size can be used for smaller systems. However, when combining several means to reduce the chip size, several small performance degradations sum up and may result in a major performance loss.

#### 14.6.4.3 Hashing Algorithm

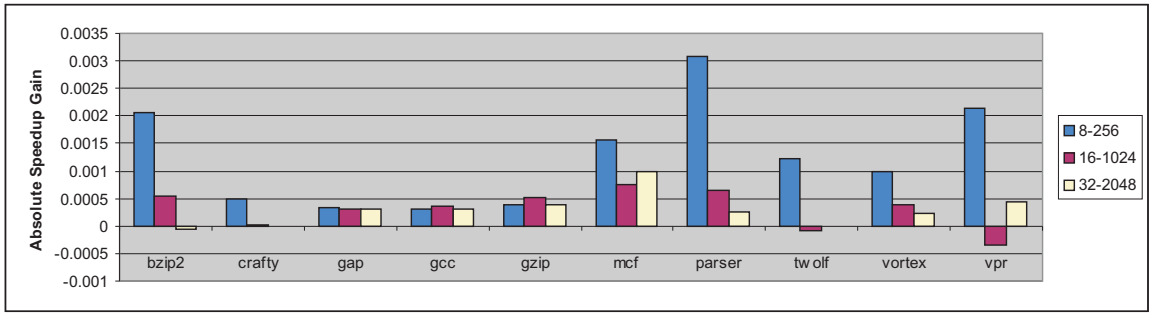
As described in section 8.1, the first hashing scheme has been improved and is now based on a tree algorithm instead of a sequential one to lower the hash computation time. Figure 14.6c[23] compares the sequential hash algorithm and a theoretical hash algorithm which is able to compute hashes within one clock cycle (denoted as *Null* in the figure) with the standard tree-based algorithm for a 8-256 cache configuration. The tree-based algorithm can improve the speedup up by more than five percent points compared to the sequential one. Its slowdown is now much closer to the optimal case determined by the Null algorithm. Furthermore, this figure shows that an even faster hashing algorithm possibly not based on a block cipher cannot improve the speedup considerably.

### 14.6.5 L2 Cache with Prefetching

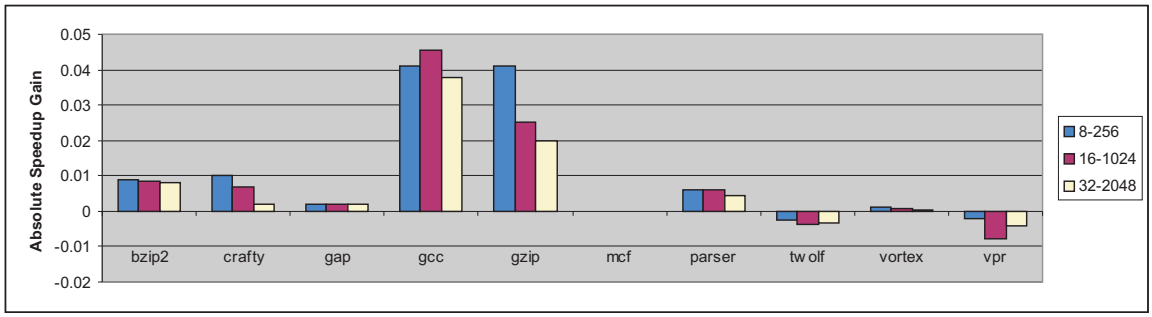
#### 14.6.5.1 Hash Value Prefetching

The results for a simple hash value prefetcher are shown in figure 14.7a[24]. The simple hash value prefetcher tries to prefetch a hash value before it is requested by the *Check Queue*. The prefetcher uses bus sniffing to be notified about new lines sent to the *Check Queue* and immediately performs a fetch to the parent hash. Unfortunately, this prefetching is not able to improve the speedup considerably. Further analysis showed that for large caches the number

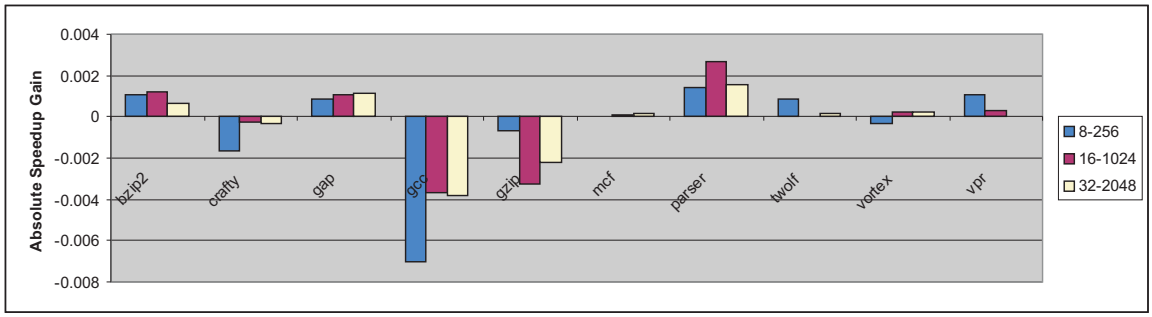
[21] Simulation shown in section B.5.4.
[22] Simulation shown in section B.6.4.
[23] Simulation shown in section B.7.4.
[24] Simulations shown in sections B.8.4, B.8.2, and B.8.3




(a) Hash value prefetching




(b) *SAM*-only prefetching




(c) Normal cache prefetching

Figure 14.7: Cache access prefetching

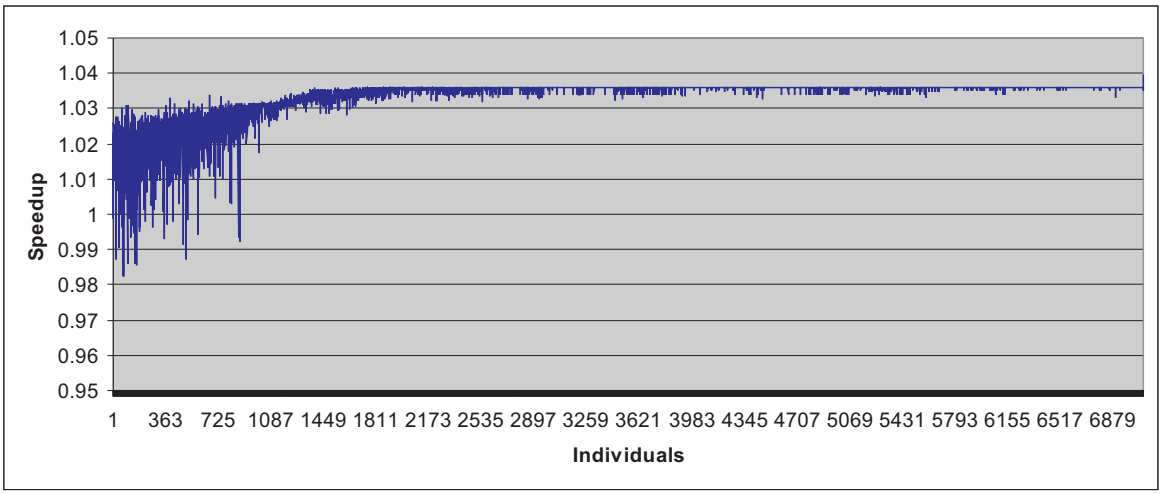



Figure 14.8: Geometric average of speedup for chosen benchmarks (8-256)

of times where hashes in upper levels of the hash tree are fetched was very low. Additionally, the amount of times where the prefetcher was not able to prefetch the required hash value before the request initiated by the *Check Queue* was high.
The results for a hash tree prefetcher that always performs a full hash walk up to the first hit are not shown here because they were similar to the ones of the simple hash prefetcher, as can be seen in section B.8.1.

#### 14.6.5.2 Data Prefetching

As described in section 14.4, JavaEvA has been used to find optimal values for the prefetcher. Figure 14.8 shows the geometric average of the speedup for the *gcc*, *gzip*, *crafty*, and *perlbmk* evaluated for a 8-256 cache configuration. The speedup has been used as a fitness function where each given value belongs to the speedup of an individual. It can be seen that the speedup converges to 1.036. For this evaluation only the four benchmarks mentioned above instead of all 12 benchmarks have been used to limit the simulation time. To further reduce the simulation time all simulations have been stopped after $2^{28}$ instructions.
The prefetcher configuration found during this evaluation is listed in the 8-256 column in table 14.11, and the absolute speedup gains of this configuration for all benchmarks are shown in figure 14.7b[25]. The reference cache configuration in this case is a standard cache configuration without prefetching capabilities. In this figure, the speedups for a cache without and with prefetching capabilities are compared. It can be observed that the implemented prefetching algorithm is not able to speed up all simulated workloads equally. For example, the *gcc* benchmark can be sped up by up to five percent points, but other benchmarks like *twolf* or *vpr* are even slightly slower.
One reason for this observation is that using a prefetcher always causes additional load on the cache system. For the *SAM* architecture this load is increased due to the hash values required to decrypt encrypted memory. To measure whether this additional load decreases

[25]Simulations shown in sections B.9.7, B.9.3, and B.9.5.

| Parameter | 8-256 | Markov | TCP | Stride-only | Markov-only |
|---|---|---|---|---|---|
| Best speedup | 1.036 | 1.0476 | 1.045 | 1.044 | 1.026 |
| Idle | 0 | 0 | 2 | 3 | 0 |
| NoCheck | no | no | no | no | no |
| AllowStrideNull | no | no | no | yes | yes |
| Buffers | 32 | 8 | 32 | 32 | 32 |
| AddressesPerBuffer | 4 | 4 | 4 | 4 | 4 |
| PriorityDecrementCycles | 100 | 80 | 90 | 15 | 45 |
| MaxConfidenceBits | 4 | 7 | 3 | 4 | 5 |
| StridePrediction | yes | yes | yes | yes | no |
| StrideBits | 5 | 5 | 5 | 9 | – |
| MarkovPrediction | yes | yes | yes | no | yes |
| MarkovAssociativity | 2 | 4 | – | – | 4 |
| MarkovBits | 8 | 7 | – | – | 12 |
| TCPEnable | no | no | yes | no | no |
| TCPTagsPerSet | – | – | 1 | – | – |
| TCPTagbits | – | – | 7 | – | – |
| TCPSetbits | – | – | 4 | – | – |

Table 14.11: JavaEvA results

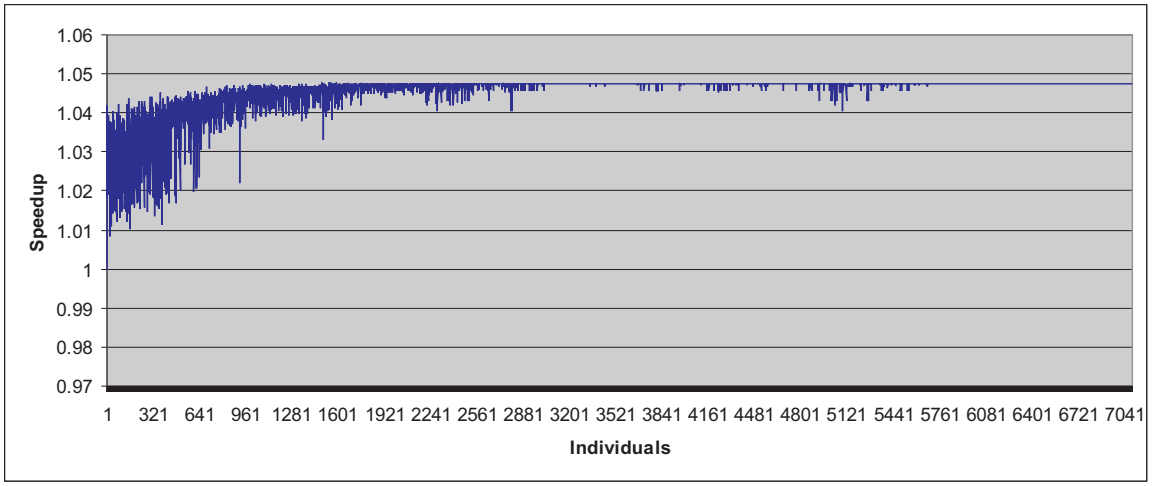
the performance of the prefetcher, a new set of simulations has been carried out. Figure 14.7c[26] shows the results of these simulations. Here the absolute speedup gain is computed with a normal cache *with* prefetching capabilities as a reference. The results show that the speedups are mostly equal. Therefore, the additional overhead of the *SAM* architecture does not worsen the performance of this prefetching algorithm. For some benchmarks like *gcc* the performance is slightly lower, others are performing slightly better but in any case the difference is less than one percent point. Hence, the prefetcher is not able to speed up protected programs more than unprotected ones.

In the following only the *gcc* workload for a 16-1024 cache configuration has been used to further analyze the prefetcher, because it has shown the best prefetching results. The influence of the predictor is negligible, as can be seen in figure 14.9. The first figure (14.9a) shows the speedup using the normal Markov predictor; the second figure (14.9b) shows the speedup for the same cache configuration, but this time using the TCP predictor. In both cases the resulting speedup is around 1.04 (see table 14.11, columns Markov and TCP). Therefore, the Markov predictor implementation has no significant effect on the results.

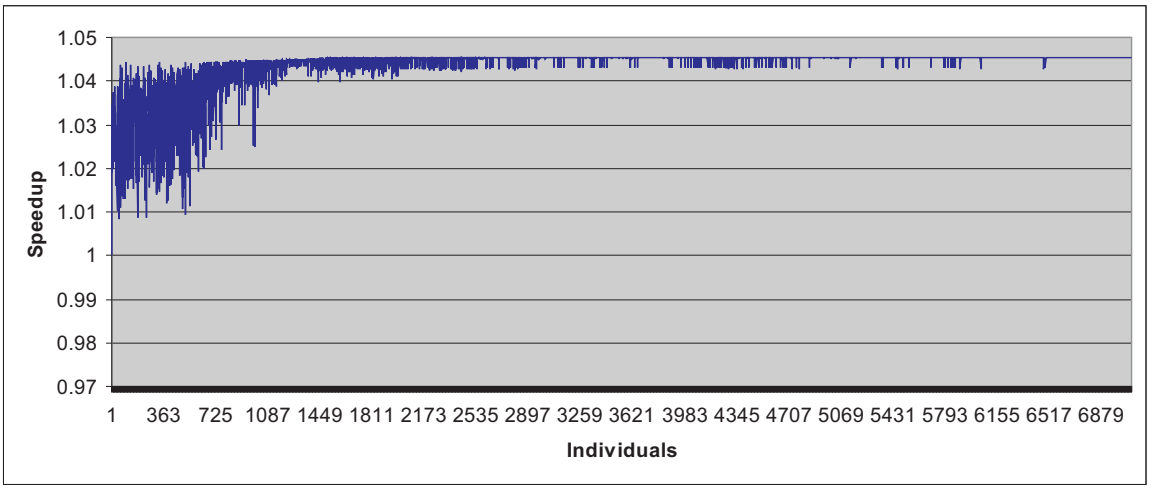
To measure the effect of the stride predictor two more configurations have been evaluated (see figure 14.10). The first one (figure 14.10a) is a pure Markov predictor without stride filter; the second one (figure 14.10b) is a pure stride predictor without additional Markov-prediction. It can be seen that the lack of the stride predictor results in a smaller speedup, whereas the lack of a Markov predictor has only no visible impact on the speedup. One reason for this behavior is that a stride predictor is able to predict compulsory[27] misses whereas a Markov predictor can only predict non-compulsory misses. The best configurations for both simulations are shown in table 14.11 in the columns Markov-only and Stride-only.

[26] Simulations shown in sections B.9.6, B.9.2, and B.9.4

[27] A miss is compulsory if the corresponding cache line is fetched the first time.




(a) Speedup for *gcc* benchmark (16-1024) with Markov-prediction




(b) Speedup for *gcc* benchmark (16-1024) with TCP prediction

Figure 14.9: Different predictors

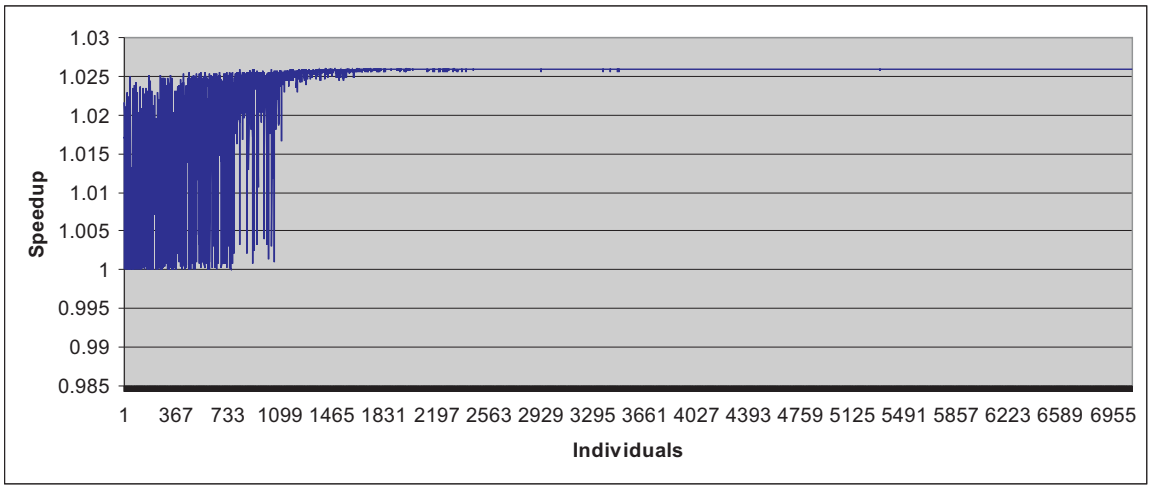
### 14.6.6 Queues

The impact of various queue-related changes is shown in the following sections. This includes queue size and queue performance variations. For the following benchmarks the cache configuration 8-256 has been used. The results for the other two configurations were similar.

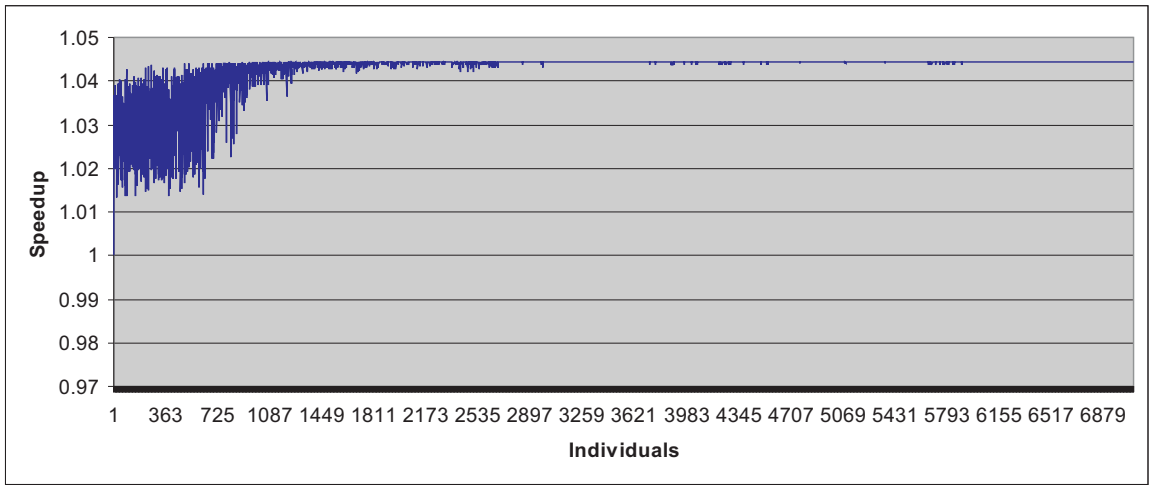
#### 14.6.6.1 Queue Sizes

Increasing the number of queue elements has no impact on the speedup, as can be seen in figure 14.11a[28]. The absolute speedup gain is close to zero for configurations using 3, 4, 10, and 20 elements when using the standard queue size or five elements as a reference. One reason for this is the more secure handling of speculative execution. The cache simply defers

[28]Simulation shown in section B.10.4.




(a) Speedup for *gcc* benchmark (16-1024) without stride filter



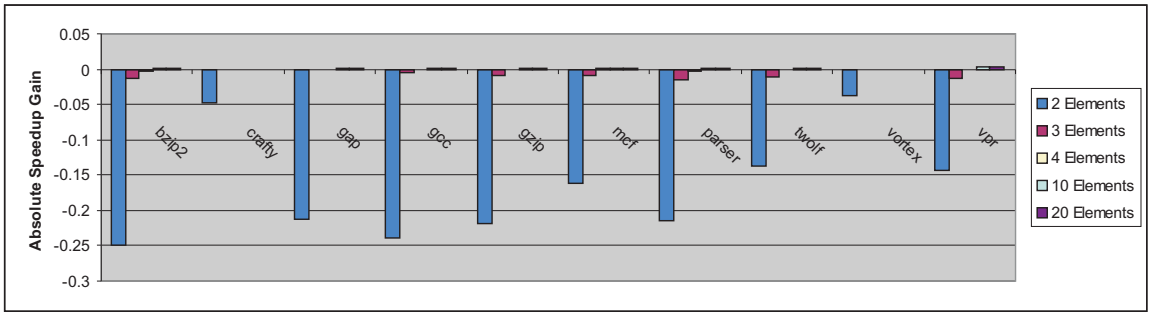

(b) Speedup for *gcc* benchmark (16-1024) without Markov predictor

Figure 14.10: Influence of stride predictors and Markov predictors

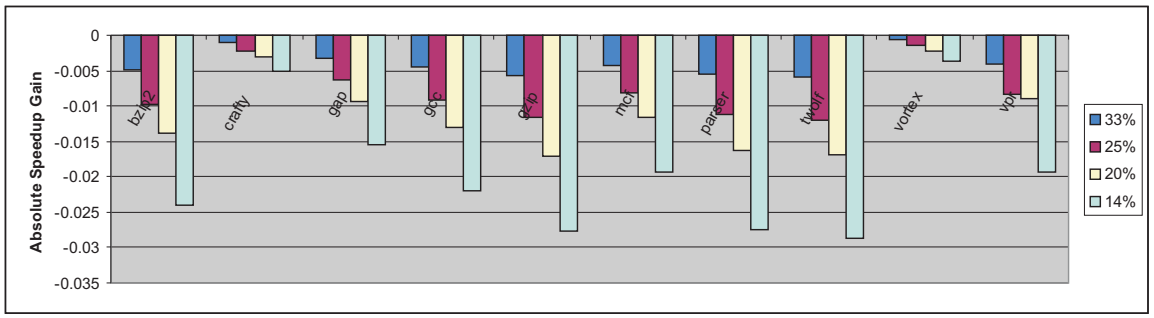
fetches which, for example, could result in additional check queue elements until the previous cache line has been verified. Hence, the only time a number of cache lines are to be verified in parallel are cases where several misses are caused by the cache itself, for example when cache accesses by the *Hash Write Queue* result in misses. However, decreasing the number of queue elements has a huge impact and can lower the speedup by more than 45 percent points, as can be seen for the results with only two queue entries. These results show that five queue elements seem to be reasonable.

#### 14.6.6.2 Queue Performance

In the default configuration the queues are clocked with half of the main processor clock. However, the main speed of the queues does not have a notable impact on the speedup




(a) Variation of the queue sizes




(b) Variation of the queue clocking

Figure 14.11: Queue-related configurations

of most benchmarks, as can be seen in figure 14.11b[29] which shows the absolute speedup gain compared to the default cache configuration. The main reason for this is that the main performance loss of the architecture is caused by the additional delay when fetching encrypted memory contents. This is not handled by queues, but by the main cache logic. The queues are involved only in data verification and write-backs, and both tasks do not have a huge impact on the whole performance. Verification is done in parallel with other cache operations and typically the processor does not have to wait until a write-back of data has finished.

### 14.6.7 Multitasking Benchmarks

For these benchmarks, shown in figures 14.12 and 14.13, four protected benchmarks have been started in parallel. The additional benchmarks are started to cause additional context switches and they have the protected parts of the operating system accessed by more than one protected context.
Figure 14.12[30] directly compares the results of single task benchmarks with the multitasking ones. It can be seen that the slowdowns are unbalanced. Some benchmarks, like *bzip2* or *twolf*, perform better in the multitasking environment, whereas others, like *gcc*, perform worse. Of course, the overall runtime of all four benchmarks is increased, but the speedup is not as high as for the execution of a single benchmark. Therefore, *SAM* performs well even when

[29] Simulation shown in section B.11.4.
[30] Simulations shown in sections B.9.9, B.9.11, and B.9.13 compared with simulations shown in sections B.9.3, B.9.5, and B.9.7.

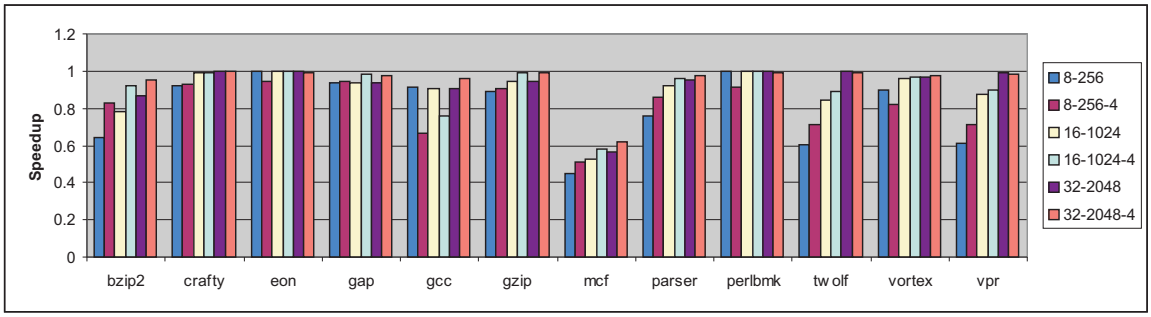



Figure 14.12: Comparison between single-task and multitasking benchmarks

executing more than one protected program at the same time.

#### 14.6.7.1 Prediction

The absolute speedup gains for the four protected processes mentioned above with the standard cache configuration without prefetching as a reference are shown in figure 14.13a[31]. The absolute speedup gains for most multitasking benchmarks are slightly inferior to those shown in figure 14.7b for single tasks benchmarks. Like for the single task case *gcc* performs best, but the speedup gain for *gzip* is much lower in the multitasking environment. However, the speedup in the multitasking case is higher than in the single task case, as can be seen in figure 14.12. Hence, *SAM's* multitasking performance for a prefetching cache is comparable to its single task performance.

#### 14.6.7.2 Cache Line Verification

For static data, like instructions, a full hash walk for verification is not required, as stated in section 8.1. Figure 14.13b[32] shows the performance improvements of this property. Here, the absolute speedup gain of configuration that performs a hash tree walk even for static data is shown with the default configuration as a reference. This optimization mainly speeds up small cache configurations, but not more than three percent points. Hence, it can be used for small systems in which no self-modifying code is executed.

#### 14.6.7.3 Cache Dictionary

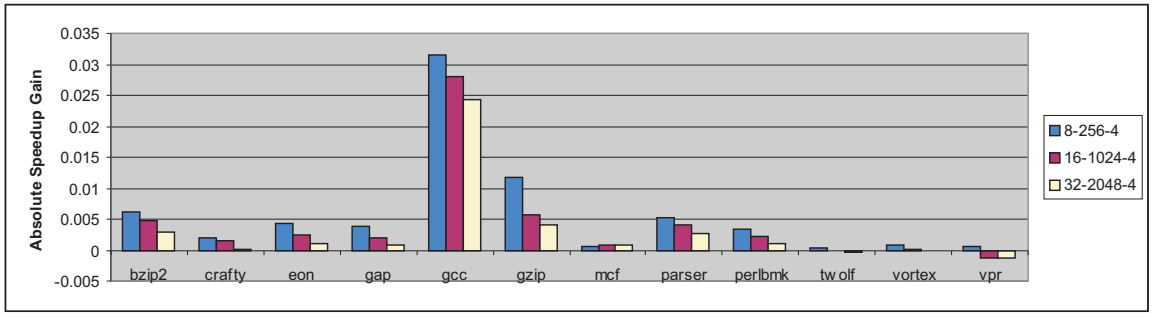
The impact of the cache dictionary can be seen in figure 14.13c[33]. Here, the absolute speedup gain for cache configurations with one and two dictionary entries (*-dic1 and *-dic2) compared with the standard configuration with four entries is given. The results show that the dictionary can improve the overall performance up to 12 percent points for the *vortex* benchmark and significantly for all other simulated workloads. This shows that shared protected memory can be handled well by the current cache implementation as long as the number of dictionary entries at least matches the number of parallel executed protected processes.

[31] Simulations shown in sections B.9.9, B.9.11, and B.9.13.
[32] Simulations shown in sections B.12.2, B.12.3, and B.12.4.
[33] Simulations shown in sections B.13.2, B.13.3, and B.13.4.

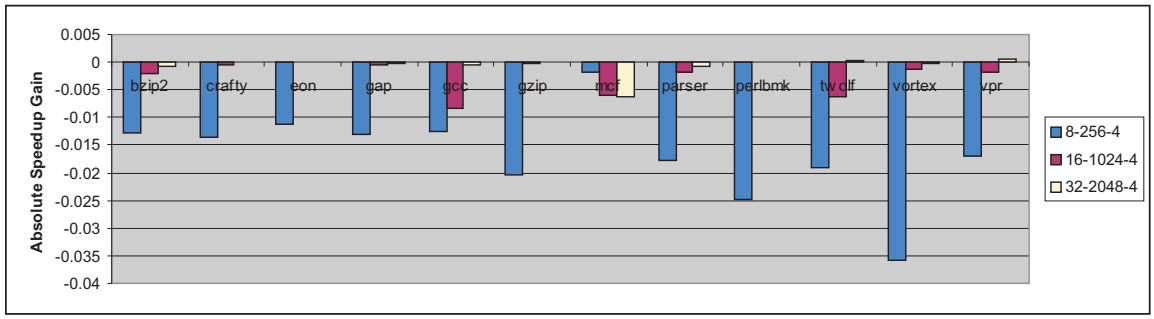
#### 14.6.7.4 Size of Protected Kernel Area

Limiting the size of the protected kernel area does not only simplify maintaining and verifying this code but can additionally improve the overall performance. Figure 14.13d[34] shows the absolute speedup gains for a fully protected kernel (*-all) and a fully unprotected kernel (*-user) compared to the default configuration with 64k protected kernel memory. It can be seen that protecting the whole kernel has a huge performance impact whereas the current small protected kernel performs as well as a configuration without any protected kernel code at all.

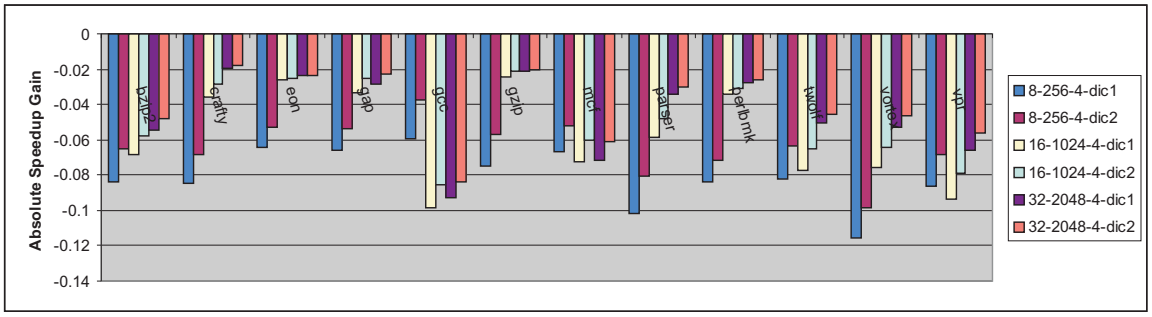
[34]Simulations shown in sections B.14.2, B.14.3, and B.14.4.



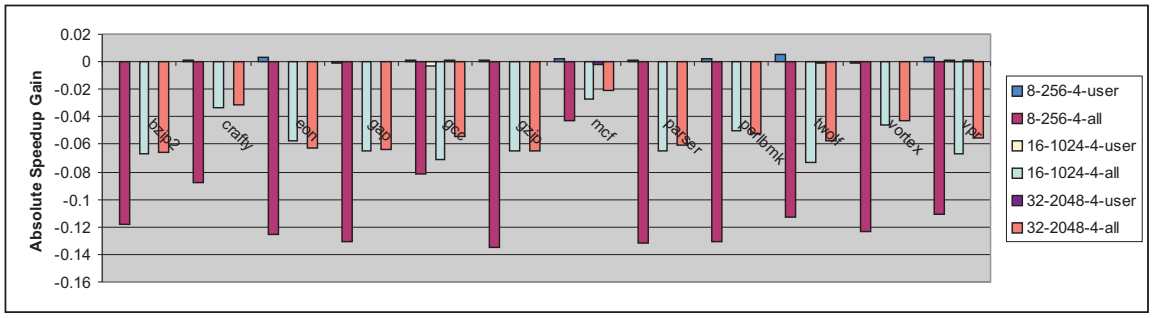

(a) *SAM*-only prefetching




(b) Hash walk for instructions




(c) Cache Dictionary




(d) Size of protected kernel

Figure 14.13: Multitasking benchmarks

# Chapter 15

# Conclusion and Future Work

In this thesis the *SAM* architecture has been presented. The architecture consists of an extension to a standard microprocessor and provides execution of protected and unprotected programs in a multitasking environment. *SAM* assumes that the processor core, including the cache and all data stored there, are protected against physical attacks. Hence, data has to be protected when leaving the core, i.e., when sensitive data is written back to memory. Inside this core, physical attacks are assumed to be impossible: protection has to be assured at a logical level, i.e., by restricting memory access.

External data is transparently protected by strong AES encryption in counter mode and hash values are used to prevent both external data disclosure and modifications. The processor supports two different memory views for encrypted memory: Encrypted data is decrypted only when accessed by instructions that are protected by hash values, and the hash values are updated for write accesses. Other instructions are only allowed to access encrypted memory, and hash values are not updated. Internally, the processor supports additional protection schemes, like register protection, and an enhanced trap handling to provide support for a protected operating system implementation. A sample operating system implementation based on Linux with a small protected kernel part has also been introduced.

The presented simulation results show that the performance penalty caused by *SAM* is quite low. Several factors contribute to this result, such as the deployment of counter mode cryptography, which hides the cryptographic latency for most loads, the reuse of hash values as a counter, which reduces the memory footprint, and the sophisticated design of the cache system.

The hardware implementation proves that the main parts of *SAM* can be implemented efficiently in hardware with a small overhead. The major overhead is caused by the cryptographic units, but they can be made available to programs to speed up normal cryptographic operations as well.

As a result, the *SAM* architecture provides a secure execution environment for programs using a standard and only partly enhanced processor and operating system without affecting the execution performance much. This reduces the effort required to adjust existing programs for *SAM*. Compared with other architectures, such as those based on a secure co-processor or AEGIS architecture, the required modifications are minimal. No dedicated compiler or development environment is needed.

This protection scheme can be used for a wide range of software, from multimedia players featuring embedded cryptographic keys, which allow a controlled decryption of multimedia

contents, to effective copy-protection schemes, which cannot be broken by software modifications, and permit, for example, the public deployment of secret algorithms. However, protection of digital multimedia content requires protected data paths to external devices to prevent sniffing when decrypted data is passed to their output device like the graphic or sound card. Therefore, the *SAM* architecture could be combined with parts of the LaGrande architecture for more security.

Future work includes hardware and software improvements. On the hardware side the architecture has to be ported to a multi-core processor design. Even a multiprocessor design where the processor cores are not located in the same housing is possible. However, then the busses used for communication between the processors have to be protected and encrypted as well. The operating system has to be enhanced as well. The current implementation lacks support for paging out protected pages and for secure signal handling and multi-threading.

# Appendix A

# System Emulation Parameters

In the appendix the abbreviations listed in table A.1 are used:

| Abbreviation | Description |
|---|---|
| CS | Cache Size. The first number represents the size of both L1 caches in kByte, and the second number represents the size of the L2 cache in kByte. |
| SID | Simulation Identifier. This is the internal identifier of this simulation. |
| BE | Benchmark Environment. Determines if a single benchmark or multiple benchmarks have been started. |
| REF | Reference Simulation. This is the reference simulation used for speedup calculations. Typically this is a standard cache without any security functions and without prefetching capabilities. |

Table A.1: Abbreviations

For all cache configurations only a small set of different trace files has been used. Table A.2 lists these configurations. For all simulations kernel 2.6.13 with *SAM* modifications has been executed on a simulated computer with 512 Mbyte memory, and writing to the trace file has been started after $2^{32}$ instructions. The column *Benchmarks* lists the number of benchmark programs started in parallel. In case of parallel execution of several benchmarks the first started benchmark is the one listed in the tables shown in chapter B, and for the remaining ones *eon* has been started. In the column *Usertermclock* the number of instructions recorded in user space for each executed benchmark before terminating the simulation is given in hexadecimal notation. Therefore, the number has to be smaller in case of parallel execution of several benchmarks to record approximately $2^{32}$ instructions.

| SID | Benchmarks | Usertermclock |
|---|---|---|
| P1 | 1 | 0x0dfffffff |
| P4 | 4 | 0x05fffffff |

Table A.2: QEMU configurations

# Appendix B

# Speedups for selected Configurations

The basic cache configuration used for all simulations without prefetching has already been listed in table 14.7.

In the following, the speedups for all evaluated cache configurations are listed.

## B.1 Speculative Execution

### B.1.1 SID: speculative

| Name | Speculative Execution |
|---|---|
| normal | yes |
| nospeculative | no |

### B.1.2 CS: 16-1024; SID: speculative (section B.1.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | |
|---|---|---|
| | normal | nospeculative |
| bzip2 | 0.782843 | 0.724325 |
| crafty | 0.987439 | 0.981999 |
| eon | 0.999977 | 0.999962 |
| gap | 0.939804 | 0.903325 |
| gcc | 0.908137 | 0.846896 |
| gzip | 0.945378 | 0.909786 |
| mcf | 0.524216 | 0.467516 |
| parser | 0.921794 | 0.884098 |
| perlbmk | 0.999925 | 0.999793 |
| twolf | 0.845337 | 0.800338 |
| vortex | 0.957067 | 0.945669 |
| vpr | 0.871296 | 0.829082 |

### B.1.3 CS: 32-2048; SID: speculative (section B.1.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | |
|---|---|---|
| | normal | nospeculative |
| bzip2 | 0.865854 | 0.826482 |
| crafty | 0.999456 | 0.999111 |
| eon | 0.999977 | 0.999962 |
| gap | 0.939902 | 0.903777 |
| gcc | 0.902791 | 0.841011 |
| gzip | 0.943702 | 0.908160 |
| mcf | 0.562288 | 0.511907 |
| parser | 0.955722 | 0.935417 |
| perlbmk | 0.999928 | 0.999797 |
| twolf | 0.999869 | 0.999754 |
| vortex | 0.969878 | 0.963177 |
| vpr | 0.994175 | 0.988151 |

### B.1.4 CS: 8-256; SID: speculative (section B.1.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | |
|---|---|---|
| | normal | nospeculative |
| bzip2 | 0.641587 | 0.550424 |
| crafty | 0.920315 | 0.885222 |
| eon | 0.999557 | 0.998845 |
| gap | 0.934038 | 0.895850 |
| gcc | 0.913673 | 0.856098 |
| gzip | 0.887723 | 0.821065 |
| mcf | 0.448160 | 0.382592 |
| parser | 0.757447 | 0.663347 |
| perlbmk | 0.998348 | 0.997045 |
| twolf | 0.606752 | 0.513884 |
| vortex | 0.899877 | 0.872151 |
| vpr | 0.609607 | 0.525767 |

## B.2 Subsequent Unverified Fetches

### B.2.1 SID: unverified fetches

| Name | Unverified subsequent data fetches | Unverified subsequent data and instruction fetches |
|---|---|---|
| notainted | no | yes |
| tainted | yes | no |
| tainteddata | no | no |

### B.2.2 CS: 16-1024; SID: unverified fetches (section B.2.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | notainted | tainted | tainteddata |
| bzip2 | 0.791727 | 0.791669 | 0.782843 |
| crafty | 0.987471 | 0.987464 | 0.987439 |
| eon | 0.999979 | 0.999977 | 0.999977 |
| gap | 0.949628 | 0.949621 | 0.939804 |
| gcc | 0.926392 | 0.926367 | 0.908137 |
| gzip | 0.957289 | 0.957242 | 0.945378 |
| mcf | 0.540353 | 0.540339 | 0.524216 |
| parser | 0.930019 | 0.929826 | 0.921794 |
| perlbmk | 0.999927 | 0.999927 | 0.999925 |
| twolf | 0.852002 | 0.852247 | 0.845337 |
| vortex | 0.957367 | 0.957374 | 0.957067 |
| vpr | 0.873071 | 0.873051 | 0.871296 |

### B.2.3 CS: 32-2048; SID: unverified fetches (section B.2.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | notainted | tainted | tainteddata |
| bzip2 | 0.869454 | 0.869247 | 0.865854 |
| crafty | 0.999458 | 0.999464 | 0.999456 |
| eon | 0.999979 | 0.999978 | 0.999977 |
| gap | 0.949843 | 0.949849 | 0.939902 |
| gcc | 0.921482 | 0.921480 | 0.902791 |
| gzip | 0.955961 | 0.955947 | 0.943702 |
| mcf | 0.580970 | 0.580962 | 0.562288 |
| parser | 0.960118 | 0.960062 | 0.955722 |
| perlbmk | 0.999931 | 0.999930 | 0.999928 |
| twolf | 0.999895 | 0.999894 | 0.999869 |
| vortex | 0.970112 | 0.970113 | 0.969878 |
| vpr | 0.994808 | 0.994807 | 0.994175 |

### B.2.4 CS: 8-256; SID: unverified fetches (section B.2.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | notainted | tainted | tainteddata |
| bzip2 | 0.660628 | 0.660548 | 0.641587 |
| crafty | 0.921523 | 0.920831 | 0.920315 |
| eon | 0.999645 | 0.999642 | 0.999557 |
| gap | 0.943686 | 0.943442 | 0.934038 |
| gcc | 0.928957 | 0.928789 | 0.913673 |
| gzip | 0.901475 | 0.901192 | 0.887723 |
| mcf | 0.459969 | 0.459934 | 0.448160 |
| parser | 0.779995 | 0.779641 | 0.757447 |
| perlbmk | 0.998289 | 0.998443 | 0.998348 |
| twolf | 0.628109 | 0.627820 | 0.606752 |
| vortex | 0.901946 | 0.901192 | 0.899877 |
| vpr | 0.626620 | 0.626368 | 0.609607 |

## B.3 Cache Size Variations

### B.3.1 SID: size

| Name | Configuration |
|---|---|
| normal | default configuration, only size variations |

### B.3.2 CS: 16-1024; SID: size (section B.3.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.782843 |
| crafty | 0.987439 |
| eon | 0.999977 |
| gap | 0.939804 |
| gcc | 0.908137 |
| gzip | 0.945378 |
| mcf | 0.524216 |
| parser | 0.921794 |
| perlbmk | 0.999925 |
| twolf | 0.845337 |
| vortex | 0.957067 |
| vpr | 0.871296 |

### B.3.3 CS: 16-2048; SID: size (section B.3.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.869762 |
| crafty | 0.999492 |
| eon | 0.999976 |
| gap | 0.938124 |
| gcc | 0.906489 |
| gzip | 0.945488 |
| mcf | 0.561093 |
| parser | 0.956404 |
| perlbmk | 0.999930 |
| twolf | 0.999865 |
| vortex | 0.969942 |
| vpr | 0.994235 |

### B.3.4 CS: 16-256; SID: size (section B.3.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.627729 |
| crafty | 0.912011 |
| eon | 0.999496 |
| gap | 0.934987 |
| gcc | 0.904055 |
| gzip | 0.881240 |
| mcf | 0.443016 |
| parser | 0.753741 |
| perlbmk | 0.998447 |
| twolf | 0.605026 |
| vortex | 0.898801 |
| vpr | 0.608234 |

### B.3.5 CS: 32-1024; SID: size (section B.3.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.776188 |
| crafty | 0.986347 |
| eon | 0.999977 |
| gap | 0.941555 |
| gcc | 0.904470 |
| gzip | 0.943803 |
| mcf | 0.522637 |
| parser | 0.920625 |
| perlbmk | 0.999923 |
| twolf | 0.848798 |
| vortex | 0.956827 |
| vpr | 0.872110 |

### B.3.6 CS: 32-2048; SID: size (section B.3.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.865854 |
| crafty | 0.999456 |
| eon | 0.999977 |
| gap | 0.939902 |
| gcc | 0.902791 |
| gzip | 0.943702 |
| mcf | 0.562288 |
| parser | 0.955722 |
| perlbmk | 0.999928 |
| twolf | 0.999869 |
| vortex | 0.969878 |
| vpr | 0.994175 |

### B.3.7 CS: 32-256; SID: size (section B.3.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.605793 |
| crafty | 0.908910 |
| eon | 0.999585 |
| gap | 0.934355 |
| gcc | 0.896305 |
| gzip | 0.877647 |
| mcf | 0.429041 |
| parser | 0.747400 |
| perlbmk | 0.998930 |
| twolf | 0.592640 |
| vortex | 0.901147 |
| vpr | 0.597564 |

### B.3.8 CS: 8-1024; SID: size (section B.3.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.787551 |
| crafty | 0.988708 |
| eon | 0.999976 |
| gap | 0.938062 |
| gcc | 0.918449 |
| gzip | 0.950181 |
| mcf | 0.519523 |
| parser | 0.923022 |
| perlbmk | 0.999920 |
| twolf | 0.841955 |
| vortex | 0.957270 |
| vpr | 0.870231 |

### B.3.9 CS: 8-2048; SID: size (section B.3.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.873104 |
| crafty | 0.999540 |
| eon | 0.999975 |
| gap | 0.936384 |
| gcc | 0.917517 |
| gzip | 0.950349 |
| mcf | 0.555186 |
| parser | 0.957190 |
| perlbmk | 0.999926 |
| twolf | 0.999852 |
| vortex | 0.970005 |
| vpr | 0.994668 |

### B.3.10 CS: 8-256; SID: size (section B.3.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.641587 |
| crafty | 0.920315 |
| eon | 0.999557 |
| gap | 0.934038 |
| gcc | 0.913673 |
| gzip | 0.887723 |
| mcf | 0.448160 |
| parser | 0.757447 |
| perlbmk | 0.998348 |
| twolf | 0.606752 |
| vortex | 0.899877 |
| vpr | 0.609607 |

## B.4 Write-through Configuration

### B.4.1 SID: write-through

| Name | Configuration |
|---|---|
| normal | default configuration, but L1 caches configured with write through strategy |

### B.4.2 CS: 16-1024; SID: write-through (section B.4.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.785817 |
| crafty | 0.987865 |
| eon | 0.999980 |
| gap | 0.955846 |
| gcc | 0.931967 |
| gzip | 0.973099 |
| mcf | 0.492431 |
| parser | 0.925775 |
| perlbmk | 0.999921 |
| twolf | 0.834828 |
| vortex | 0.957352 |
| vpr | 0.864970 |

### B.4.3 CS: 16-2048; SID: write-through (section B.4.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.872317 |
| crafty | 0.999505 |
| eon | 0.999979 |
| gap | 0.954573 |
| gcc | 0.931300 |
| gzip | 0.973597 |
| mcf | 0.524061 |
| parser | 0.958988 |
| perlbmk | 0.999926 |
| twolf | 0.999846 |
| vortex | 0.970259 |
| vpr | 0.994916 |

### B.4.4 CS: 16-256; SID: write-through (section B.4.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.644704 |
| crafty | 0.916595 |
| eon | 0.999632 |
| gap | 0.952207 |
| gcc | 0.927460 |
| gzip | 0.916184 |
| mcf | 0.427370 |
| parser | 0.765483 |
| perlbmk | 0.998535 |
| twolf | 0.598150 |
| vortex | 0.900741 |
| vpr | 0.602358 |

### B.4.5 CS: 32-1024; SID: write-through (section B.4.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.782388 |
| crafty | 0.986948 |
| eon | 0.999980 |
| gap | 0.955733 |
| gcc | 0.930618 |
| gzip | 0.972855 |
| mcf | 0.490860 |
| parser | 0.924651 |
| perlbmk | 0.999919 |
| twolf | 0.835062 |
| vortex | 0.957221 |
| vpr | 0.865887 |

### B.4.6 CS: 32-2048; SID: write-through (section B.4.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.869865 |
| crafty | 0.999472 |
| eon | 0.999979 |
| gap | 0.954351 |
| gcc | 0.929829 |
| gzip | 0.973130 |
| mcf | 0.522014 |
| parser | 0.958169 |
| perlbmk | 0.999924 |
| twolf | 0.999844 |
| vortex | 0.970178 |
| vpr | 0.994889 |

### B.4.7 CS: 32-256; SID: write-through (section B.4.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.641421 |
| crafty | 0.916596 |
| eon | 0.999717 |
| gap | 0.952546 |
| gcc | 0.926321 |
| gzip | 0.916784 |
| mcf | 0.427880 |
| parser | 0.764067 |
| perlbmk | 0.998968 |
| twolf | 0.598466 |
| vortex | 0.905057 |
| vpr | 0.603591 |

### B.4.8 CS: 8-1024; SID: write-through (section B.4.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.789633 |
| crafty | 0.988892 |
| eon | 0.999981 |
| gap | 0.956099 |
| gcc | 0.937188 |
| gzip | 0.974692 |
| mcf | 0.493423 |
| parser | 0.927161 |
| perlbmk | 0.999924 |
| twolf | 0.838968 |
| vortex | 0.957726 |
| vpr | 0.865782 |

### B.4.9 CS: 8-2048; SID: write-through (section B.4.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.875024 |
| crafty | 0.999546 |
| eon | 0.999979 |
| gap | 0.954904 |
| gcc | 0.936888 |
| gzip | 0.975243 |
| mcf | 0.525200 |
| parser | 0.959961 |
| perlbmk | 0.999930 |
| twolf | 0.999844 |
| vortex | 0.970480 |
| vpr | 0.994880 |

### B.4.10 CS: 8-256; SID: write-through (section B.4.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups normal |
|---|---|
| bzip2 | 0.648695 |
| crafty | 0.922486 |
| eon | 0.999668 |
| gap | 0.952169 |
| gcc | 0.931284 |
| gzip | 0.918231 |
| mcf | 0.427404 |
| parser | 0.767307 |
| perlbmk | 0.998401 |
| twolf | 0.603380 |
| vortex | 0.901078 |
| vpr | 0.602422 |

## B.5 AES Units

### B.5.1 SID: AES

| Name | Number of simulated AES units |
|---|---|
| aes-1 | 1 |
| aes-2 | 2 |
| aes-3 | 3 |
| aes-4 | 4 |
| aes-5 | 5 |
| aes-6 | 6 |

### B.5.2 CS: 16-1024; SID: AES (section B.5.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | | | | |
|---|---|---|---|---|---|---|
| | aes-1 | aes-2 | aes-3 | aes-4 | aes-5 | aes-6 |
| bzip2 | 0.682901 | 0.754218 | 0.761478 | 0.781285 | 0.782843 | 0.784093 |
| crafty | 0.981310 | 0.985635 | 0.985963 | 0.987396 | 0.987439 | 0.987502 |
| eon | 0.999953 | 0.999969 | 0.999971 | 0.999976 | 0.999977 | 0.999977 |
| gap | 0.823029 | 0.910182 | 0.924144 | 0.936653 | 0.939804 | 0.943058 |
| gcc | 0.738831 | 0.863263 | 0.880312 | 0.905001 | 0.908137 | 0.910607 |
| gzip | 0.790237 | 0.904710 | 0.926894 | 0.941470 | 0.945378 | 0.950298 |
| mcf | 0.406315 | 0.489880 | 0.502579 | 0.521346 | 0.524216 | 0.526577 |
| parser | 0.823984 | 0.896231 | 0.905883 | 0.919503 | 0.921794 | 0.923500 |
| perlbmk | 0.999790 | 0.999880 | 0.999886 | 0.999925 | 0.999925 | 0.999926 |
| twolf | 0.756151 | 0.819504 | 0.826603 | 0.843842 | 0.845337 | 0.847089 |
| vortex | 0.948904 | 0.954764 | 0.955104 | 0.957031 | 0.957067 | 0.957093 |
| vpr | 0.810637 | 0.851389 | 0.855766 | 0.870551 | 0.871296 | 0.872305 |

### B.5.3 CS: 32-2048; SID: AES (section B.5.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | | | | |
|---|---|---|---|---|---|---|
| | aes-1 | aes-2 | aes-3 | aes-4 | aes-5 | aes-6 |
| bzip2 | 0.796317 | 0.845839 | 0.850365 | 0.864972 | 0.865854 | 0.866670 |
| crafty | 0.999034 | 0.999324 | 0.999341 | 0.999454 | 0.999456 | 0.999458 |
| eon | 0.999953 | 0.999970 | 0.999971 | 0.999977 | 0.999977 | 0.999977 |
| gap | 0.824435 | 0.910895 | 0.924457 | 0.936825 | 0.939902 | 0.943154 |
| gcc | 0.732965 | 0.857782 | 0.874512 | 0.899491 | 0.902791 | 0.905424 |
| gzip | 0.787709 | 0.903097 | 0.925009 | 0.939705 | 0.943702 | 0.948646 |
| mcf | 0.445139 | 0.526385 | 0.537075 | 0.560137 | 0.562288 | 0.564599 |
| parser | 0.899393 | 0.941687 | 0.946971 | 0.954522 | 0.955722 | 0.956629 |
| perlbmk | 0.999792 | 0.999882 | 0.999888 | 0.999928 | 0.999928 | 0.999929 |
| twolf | 0.999681 | 0.999812 | 0.999822 | 0.999866 | 0.999869 | 0.999871 |
| vortex | 0.964544 | 0.968327 | 0.968546 | 0.969851 | 0.969878 | 0.969889 |
| vpr | 0.983891 | 0.991552 | 0.991548 | 0.994042 | 0.994175 | 0.994458 |

### B.5.4 CS: 8-256; SID: AES (section B.5.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | | | | |
|---|---|---|---|---|---|---|
| | aes-1 | aes-2 | aes-3 | aes-4 | aes-5 | aes-6 |
| bzip2 | 0.509937 | 0.601963 | 0.613473 | 0.638319 | 0.641587 | 0.643816 |
| crafty | 0.891124 | 0.911378 | 0.912854 | 0.920119 | 0.920315 | 0.920496 |
| eon | 0.998207 | 0.999160 | 0.999240 | 0.999547 | 0.999557 | 0.999564 |
| gap | 0.816992 | 0.903910 | 0.917971 | 0.930819 | 0.934038 | 0.937138 |
| gcc | 0.765732 | 0.873695 | 0.888266 | 0.910854 | 0.913673 | 0.916025 |
| gzip | 0.691972 | 0.832368 | 0.857343 | 0.882977 | 0.887723 | 0.893110 |
| mcf | 0.344866 | 0.419264 | 0.431534 | 0.445772 | 0.448160 | 0.450741 |
| parser | 0.594994 | 0.706290 | 0.723758 | 0.751956 | 0.757447 | 0.760916 |
| perlbmk | 0.996395 | 0.997778 | 0.997878 | 0.998338 | 0.998348 | 0.998357 |
| twolf | 0.442762 | 0.557166 | 0.574925 | 0.601770 | 0.606752 | 0.610462 |
| vortex | 0.878224 | 0.893466 | 0.894503 | 0.899680 | 0.899877 | 0.899979 |
| vpr | 0.501568 | 0.578942 | 0.586612 | 0.607324 | 0.609607 | 0.611590 |

## B.6 Performance of AES Units

### B.6.1 SID: AES cycles

| Name | Required Cycles for AES encryption |
|---|---|
| aes-10 | 10 |
| aes-20 | 20 |
| aes-40 | 40 |
| aes-80 | 80 |

### B.6.2 CS: 16-1024; SID: AES cycles (section B.6.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | | |
|---|---|---|---|---|
| | aes-10 | aes-20 | aes-40 | aes-80 |
| bzip2 | 0.790517 | 0.782843 | 0.763395 | 0.674811 |
| crafty | 0.987701 | 0.987439 | 0.987007 | 0.982979 |
| eon | 0.999979 | 0.999977 | 0.999970 | 0.999945 |
| gap | 0.952718 | 0.939804 | 0.908832 | 0.819358 |
| gcc | 0.923757 | 0.908137 | 0.861878 | 0.719105 |
| gzip | 0.962740 | 0.945378 | 0.898820 | 0.787597 |
| mcf | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| parser | 0.930206 | 0.921794 | 0.897563 | 0.816684 |
| perlbmk | 0.999927 | 0.999925 | 0.999916 | 0.999823 |
| twolf | 0.852930 | 0.845337 | 0.828374 | 0.757434 |
| vortex | 0.957381 | 0.957067 | 0.956392 | 0.949878 |
| vpr | 0.875750 | 0.871296 | 0.863533 | 0.818983 |

### B.6.3 CS: 32-2048; SID: AES cycles (section B.6.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | | |
|---|---|---|---|---|
| | aes-10 | aes-20 | aes-40 | aes-80 |
| bzip2 | 0.869768 | 0.865854 | 0.855407 | 0.799491 |
| crafty | 0.999475 | 0.999456 | 0.999423 | 0.999143 |
| eon | 0.999979 | 0.999977 | 0.999970 | 0.999946 |
| gap | 0.952718 | 0.939902 | 0.909345 | 0.820552 |
| gcc | 0.919155 | 0.902791 | 0.855977 | 0.713262 |
| gzip | 0.961219 | 0.943702 | 0.896478 | 0.784251 |
| mcf | 0.577695 | 0.562288 | 0.532457 | 0.438140 |
| parser | 0.960300 | 0.955722 | 0.942658 | 0.895915 |
| perlbmk | 0.999930 | 0.999928 | 0.999918 | 0.999825 |
| twolf | 0.999888 | 0.999869 | 0.999824 | 0.999626 |
| vortex | 0.970039 | 0.969878 | 0.969447 | 0.965368 |
| vpr | 0.995310 | 0.994175 | 0.994076 | 0.987925 |

### B.6.4 CS: 8-256; SID: AES cycles (section B.6.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | | |
|---|---|---|---|---|
| | aes-10 | aes-20 | aes-40 | aes-80 |
| bzip2 | 0.657284 | 0.641587 | 0.605339 | 0.483649 |
| crafty | 0.921849 | 0.920315 | 0.916564 | 0.891913 |
| eon | 0.999609 | 0.999557 | 0.999353 | 0.998426 |
| gap | 0.946848 | 0.934038 | 0.902516 | 0.812167 |
| gcc | 0.926774 | 0.913673 | 0.874598 | 0.756565 |
| gzip | 0.905336 | 0.887723 | 0.835550 | 0.693591 |
| mcf | 0.459587 | 0.448160 | 0.425859 | 0.348739 |
| parser | 0.776090 | 0.757447 | 0.706618 | 0.566630 |
| perlbmk | 0.998467 | 0.998348 | 0.998095 | 0.996828 |
| twolf | 0.626887 | 0.606752 | 0.560532 | 0.423687 |
| vortex | 0.901308 | 0.899877 | 0.896624 | 0.875431 |
| vpr | 0.619462 | 0.609607 | 0.589398 | 0.489479 |

## B.7 Different Hashing Algorithms

### B.7.1 SID: hashing algorithm

| Name | Hashing Algorithm |
|---|---|
| hashnone | Hash calculation requires one clock cycle |
| hashseq | Sequential hashing algorithm |
| hashtree | Tree-based hashing algorithm |

### B.7.2 CS: 16-1024; SID: hashing algorithm (section B.7.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | hashnone | hashseq | hashtree |
| bzip2 | 0.792587 | 0.761459 | 0.782843 |
| crafty | 0.987693 | 0.987093 | 0.987439 |
| eon | 0.999980 | 0.999968 | 0.999977 |
| gap | 0.955013 | 0.908625 | 0.939804 |
| gcc | 0.930638 | 0.858582 | 0.908137 |
| gzip | 0.964447 | 0.900812 | 0.945378 |
| mcf | 0.543215 | 0.495783 | 0.524216 |
| parser | 0.931858 | 0.897417 | 0.921794 |
| perlbmk | 0.999928 | 0.999919 | 0.999925 |
| twolf | 0.853905 | 0.829273 | 0.845337 |
| vortex | 0.957531 | 0.956271 | 0.957067 |
| vpr | 0.874891 | 0.863986 | 0.871296 |

### B.7.3 CS: 32-2048; SID: hashing algorithm (section B.7.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | hashnone | hashseq | hashtree |
| bzip2 | 0.870014 | 0.855530 | 0.865854 |
| crafty | 0.999479 | 0.999427 | 0.999456 |
| eon | 0.999980 | 0.999969 | 0.999977 |
| gap | 0.955023 | 0.908909 | 0.939902 |
| gcc | 0.925959 | 0.852488 | 0.902791 |
| gzip | 0.963119 | 0.898111 | 0.943702 |
| mcf | 0.582937 | 0.531363 | 0.562288 |
| parser | 0.961056 | 0.942610 | 0.955722 |
| perlbmk | 0.999931 | 0.999922 | 0.999928 |
| twolf | 0.999897 | 0.999816 | 0.999869 |
| vortex | 0.970135 | 0.969404 | 0.969878 |
| vpr | 0.995516 | 0.994189 | 0.994175 |

### B.7.4 CS: 8-256; SID: hashing algorithm (section B.7.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | hashnone | hashseq | hashtree |
| bzip2 | 0.665034 | 0.598368 | 0.641587 |
| crafty | 0.922291 | 0.916111 | 0.920315 |
| eon | 0.999621 | 0.999363 | 0.999557 |
| gap | 0.949051 | 0.902254 | 0.934038 |
| gcc | 0.932456 | 0.872691 | 0.913673 |
| gzip | 0.907941 | 0.839065 | 0.887723 |
| mcf | 0.463252 | 0.426375 | 0.448160 |
| parser | 0.783640 | 0.702180 | 0.757447 |
| perlbmk | 0.998485 | 0.998058 | 0.998348 |
| twolf | 0.633528 | 0.557436 | 0.606752 |
| vortex | 0.901990 | 0.895757 | 0.899877 |
| vpr | 0.624497 | 0.587289 | 0.609607 |

## B.8 Hash Value Prefetching

### B.8.1 SID: hash prefetching

| Name | Prefetching |
|---|---|
| hashpf | Hash value prefetching |
| hashsniff | Sniffing-based hash prediction |
| hashtree | Tree-based hash prediction |
| nopred | No hash value prediction |

### B.8.2 CS: 16-1024; SID: hash prefetching (section B.8.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | | |
|---|---|---|---|---|
| | hashpf | hashsniff | hashtree | nopred |
| bzip2 | 0.783323 | 0.783382 | 0.783118 | 0.782843 |
| crafty | 0.987470 | 0.987459 | 0.987465 | 0.987439 |
| eon | 0.999977 | 0.999977 | 0.999977 | 0.999977 |
| gap | 0.940121 | 0.940106 | 0.940442 | 0.939804 |
| gcc | 0.908389 | 0.908505 | 0.908259 | 0.908137 |
| gzip | 0.945802 | 0.945886 | 0.946213 | 0.945378 |
| mcf | 0.524851 | 0.524979 | 0.523705 | 0.524216 |
| parser | 0.922424 | 0.922455 | 0.922375 | 0.921794 |
| perlbmk | 0.999926 | 0.999926 | 0.999925 | 0.999925 |
| twolf | 0.845236 | 0.845248 | 0.845263 | 0.845337 |
| vortex | 0.957004 | 0.957463 | 0.957107 | 0.957067 |
| vpr | 0.871022 | 0.870945 | 0.871530 | 0.871296 |

### B.8.3 CS: 32-2048; SID: hash prefetching (section B.8.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | | |
|---|---|---|---|---|
| | hashpf | hashsniff | hashtree | nopred |
| bzip2 | 0.865759 | 0.865793 | 0.865939 | 0.865854 |
| crafty | 0.999458 | 0.999461 | 0.999456 | 0.999456 |
| eon | 0.999977 | 0.999977 | 0.999977 | 0.999977 |
| gap | 0.940204 | 0.940219 | 0.940524 | 0.939902 |
| gcc | 0.902981 | 0.903105 | 0.902839 | 0.902791 |
| gzip | 0.943968 | 0.944086 | 0.944379 | 0.943702 |
| mcf | 0.563195 | 0.563271 | 0.562461 | 0.562288 |
| parser | 0.955910 | 0.955984 | 0.955954 | 0.955722 |
| perlbmk | 0.999929 | 0.999929 | 0.999929 | 0.999928 |
| twolf | 0.999872 | 0.999872 | 0.999871 | 0.999869 |
| vortex | 0.969971 | 0.970109 | 0.969915 | 0.969878 |
| vpr | 0.994623 | 0.994604 | 0.994168 | 0.994175 |

### B.8.4 CS: 8-256; SID: hash prefetching (section B.8.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | | |
|---|---|---|---|---|
| | hashpf | hashsniff | hashtree | nopred |
| bzip2 | 0.643250 | 0.643658 | 0.642684 | 0.641587 |
| crafty | 0.920785 | 0.920798 | 0.920787 | 0.920315 |
| eon | 0.999558 | 0.999558 | 0.999559 | 0.999557 |
| gap | 0.934370 | 0.934362 | 0.934754 | 0.934038 |
| gcc | 0.913854 | 0.913986 | 0.913733 | 0.913673 |
| gzip | 0.888052 | 0.888104 | 0.888339 | 0.887723 |
| mcf | 0.447913 | 0.449724 | 0.447217 | 0.448160 |
| parser | 0.760631 | 0.760541 | 0.759516 | 0.757447 |
| perlbmk | 0.998394 | 0.998352 | 0.998351 | 0.998348 |
| twolf | 0.607952 | 0.607971 | 0.607318 | 0.606752 |
| vortex | 0.899038 | 0.900874 | 0.899650 | 0.899877 |
| vpr | 0.611284 | 0.611754 | 0.610739 | 0.609607 |

## B.9 Cache Line Prefetching

### B.9.1 SID: prefetching

| Name | Prefetching |
|---|---|
| nopred | Cache without prefetching but with *SAM* enabled |
| pred | Cache with prefetching and *SAM* enabled |
| reference | Normal cache without prefetching and without *SAM* |
| prednosam | Normal cache with prefetching and without *SAM* |

### B.9.2 CS: 16-1024; SID: prefetching (section B.9.1); BE: P1, REF: cache with prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | nopred | pred | reference |
| bzip2 | 0.775585 | 0.784064 | 0.990729 |
| crafty | 0.980249 | 0.987218 | 0.992719 |
| eon | 0.997375 | 0.999944 | 0.997399 |
| gap | 0.938823 | 0.940871 | 0.998957 |
| gcc | 0.861094 | 0.904434 | 0.948198 |
| gzip | 0.917626 | 0.942139 | 0.970645 |
| mcf | 0.524368 | 0.524285 | 1.000290 |
| parser | 0.918424 | 0.924468 | 0.996344 |
| perlbmk | 0.997002 | 0.999925 | 0.997076 |
| twolf | 0.849273 | 0.845351 | 1.004657 |
| vortex | 0.956455 | 0.957310 | 0.999360 |
| vpr | 0.879455 | 0.871576 | 1.009364 |

### B.9.3 CS: 16-1024; SID: prefetching (section B.9.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | nopred | pred | prednosam |
| bzip2 | 0.782843 | 0.791402 | 1.009358 |
| crafty | 0.987439 | 0.994459 | 1.007334 |
| eon | 0.999977 | 1.002552 | 1.002608 |
| gap | 0.939804 | 0.941853 | 1.001044 |
| gcc | 0.908137 | 0.953845 | 1.054632 |
| gzip | 0.945378 | 0.970632 | 1.030243 |
| mcf | 0.524216 | 0.524133 | 0.999710 |
| parser | 0.921794 | 0.927860 | 1.003669 |
| perlbmk | 0.999925 | 1.002857 | 1.002933 |
| twolf | 0.845337 | 0.841432 | 0.995365 |
| vortex | 0.957067 | 0.957923 | 1.000640 |
| vpr | 0.871296 | 0.863490 | 0.990723 |

### B.9.4 CS: 32-2048; SID: prefetching (section B.9.1); BE: P1, REF: cache with prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | nopred | pred | reference |
| bzip2 | 0.858693 | 0.866515 | 0.991730 |
| crafty | 0.997280 | 0.999154 | 0.997823 |
| eon | 0.998265 | 0.999951 | 0.998288 |
| gap | 0.939008 | 0.941056 | 0.999049 |
| gcc | 0.863014 | 0.898950 | 0.955940 |
| gzip | 0.922031 | 0.941468 | 0.977036 |
| mcf | 0.562517 | 0.562457 | 1.000408 |
| parser | 0.952975 | 0.957248 | 0.997126 |
| perlbmk | 0.999174 | 1.000005 | 0.999246 |
| twolf | 1.003482 | 1.000031 | 1.003613 |
| vortex | 0.969577 | 0.970088 | 0.999690 |
| vpr | 0.998583 | 0.994232 | 1.004433 |

### B.9.5 CS: 32-2048; SID: prefetching (section B.9.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | nopred | pred | prednosam |
| bzip2 | 0.865854 | 0.873741 | 1.008339 |
| crafty | 0.999456 | 1.001334 | 1.002182 |
| eon | 0.999977 | 1.001665 | 1.001715 |
| gap | 0.939902 | 0.941952 | 1.000952 |
| gcc | 0.902791 | 0.940384 | 1.046091 |
| gzip | 0.943702 | 0.963596 | 1.023503 |
| mcf | 0.562288 | 0.562228 | 0.999592 |
| parser | 0.955722 | 0.960008 | 1.002882 |
| perlbmk | 0.999928 | 1.000760 | 1.000755 |
| twolf | 0.999869 | 0.996431 | 0.996400 |
| vortex | 0.969878 | 0.970389 | 1.000310 |
| vpr | 0.994175 | 0.989844 | 0.995586 |

### B.9.6 CS: 8-256; SID: prefetching (section B.9.1); BE: P1, REF: cache with prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | nopred | pred | reference |
| bzip2 | 0.634067 | 0.642678 | 0.988279 |
| crafty | 0.908652 | 0.918655 | 0.987327 |
| eon | 0.994365 | 0.999367 | 0.994805 |
| gap | 0.932827 | 0.934928 | 0.998703 |
| gcc | 0.867820 | 0.906625 | 0.949814 |
| gzip | 0.847725 | 0.887033 | 0.954943 |
| mcf | 0.448264 | 0.448205 | 1.000232 |
| parser | 0.752884 | 0.758889 | 0.993976 |
| perlbmk | 0.994616 | 0.998410 | 0.996262 |
| twolf | 0.610043 | 0.607596 | 1.005423 |
| vortex | 0.898278 | 0.899564 | 0.998223 |
| vpr | 0.612810 | 0.610667 | 1.005254 |

### B.9.7 CS: 8-256; SID: prefetching (section B.9.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | nopred | pred | prednosam |
| bzip2 | 0.641587 | 0.650300 | 1.011860 |
| crafty | 0.920315 | 0.930447 | 1.012836 |
| eon | 0.999557 | 1.004586 | 1.005222 |
| gap | 0.934038 | 0.936142 | 1.001299 |
| gcc | 0.913673 | 0.954529 | 1.052838 |
| gzip | 0.887723 | 0.928886 | 1.047183 |
| mcf | 0.448160 | 0.448100 | 0.999768 |
| parser | 0.757447 | 0.763488 | 1.006061 |
| perlbmk | 0.998348 | 1.002156 | 1.003752 |
| twolf | 0.606752 | 0.604319 | 0.994607 |
| vortex | 0.899877 | 0.901165 | 1.001780 |
| vpr | 0.609607 | 0.607476 | 0.994773 |

### B.9.8 CS: 16-1024; SID: prefetching (section B.9.1); BE: P4, REF: cache with prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | nopred | pred | reference |
| bzip2 | 0.917216 | 0.921920 | 0.995894 |
| crafty | 0.992506 | 0.993905 | 0.998487 |
| eon | 0.992776 | 0.995114 | 0.997800 |
| gap | 0.978004 | 0.980032 | 0.998056 |
| gcc | 0.741250 | 0.768759 | 0.978381 |
| gzip | 0.986588 | 0.992262 | 0.994463 |
| mcf | 0.584062 | 0.584789 | 0.999247 |
| parser | 0.954641 | 0.958779 | 0.996796 |
| perlbmk | 0.994662 | 0.996887 | 0.997725 |
| twolf | 0.887099 | 0.886876 | 0.999725 |
| vortex | 0.967143 | 0.967335 | 0.999883 |
| vpr | 0.896272 | 0.894973 | 1.001610 |

### B.9.9 CS: 16-1024; SID: prefetching (section B.9.1); BE: P4, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | nopred | pred | prednosam |
| bzip2 | 0.920998 | 0.925721 | 1.004123 |
| crafty | 0.994010 | 0.995411 | 1.001515 |
| eon | 0.994965 | 0.997309 | 1.002205 |
| gap | 0.979909 | 0.981941 | 1.001948 |
| gcc | 0.757629 | 0.785745 | 1.022096 |
| gzip | 0.992082 | 0.997787 | 1.005568 |
| mcf | 0.584502 | 0.585229 | 1.000753 |
| parser | 0.957709 | 0.961861 | 1.003214 |
| perlbmk | 0.996929 | 0.999160 | 1.002280 |
| twolf | 0.887343 | 0.887120 | 1.000275 |
| vortex | 0.967257 | 0.967448 | 1.000117 |
| vpr | 0.894831 | 0.893534 | 0.998392 |

### B.9.10 CS: 32-2048; SID: prefetching (section B.9.1); BE: P4, REF: cache with prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | nopred | pred | reference |
| bzip2 | 0.949939 | 0.952920 | 0.997358 |
| crafty | 0.996511 | 0.996672 | 0.999737 |
| eon | 0.987768 | 0.988905 | 0.999041 |
| gap | 0.976624 | 0.977354 | 0.999368 |
| gcc | 0.937608 | 0.961220 | 0.975527 |
| gzip | 0.985137 | 0.989219 | 0.995979 |
| mcf | 0.622963 | 0.623694 | 0.999821 |
| parser | 0.973187 | 0.975925 | 0.997812 |
| perlbmk | 0.992456 | 0.993438 | 0.999063 |
| twolf | 0.990162 | 0.989703 | 1.000419 |
| vortex | 0.976994 | 0.976854 | 1.000112 |
| vpr | 0.987709 | 0.986509 | 1.001071 |

### B.9.11 CS: 32-2048; SID: prefetching (section B.9.1); BE: P4, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | nopred | pred | prednosam |
| bzip2 | 0.952455 | 0.955444 | 1.002649 |
| crafty | 0.996773 | 0.996935 | 1.000263 |
| eon | 0.988715 | 0.989854 | 1.000960 |
| gap | 0.977242 | 0.977973 | 1.000633 |
| gcc | 0.961130 | 0.985333 | 1.025087 |
| gzip | 0.989115 | 0.993212 | 1.004037 |
| mcf | 0.623074 | 0.623806 | 1.000179 |
| parser | 0.975321 | 0.978065 | 1.002193 |
| perlbmk | 0.993387 | 0.994369 | 1.000938 |
| twolf | 0.989747 | 0.989288 | 0.999581 |
| vortex | 0.976884 | 0.976745 | 0.999888 |
| vpr | 0.986653 | 0.985454 | 0.998930 |

### B.9.12 CS: 8-256; SID: prefetching (section B.9.1); BE: P4, REF: cache with prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | nopred | pred | reference |
| bzip2 | 0.823514 | 0.829747 | 0.992594 |
| crafty | 0.928034 | 0.930056 | 0.996643 |
| eon | 0.941501 | 0.945751 | 0.994451 |
| gap | 0.936737 | 0.940457 | 0.995225 |
| gcc | 0.657607 | 0.688679 | 0.984055 |
| gzip | 0.892456 | 0.904153 | 0.986895 |
| mcf | 0.508231 | 0.508728 | 0.998787 |
| parser | 0.853173 | 0.858259 | 0.993820 |
| perlbmk | 0.906785 | 0.910087 | 0.994730 |
| twolf | 0.714506 | 0.714858 | 0.998645 |
| vortex | 0.818767 | 0.819573 | 0.998081 |
| vpr | 0.714811 | 0.715363 | 0.998376 |

### B.9.13 CS: 8-256; SID: prefetching (section B.9.1); BE: P4, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | nopred | pred | prednosam |
| bzip2 | 0.829659 | 0.835938 | 1.007462 |
| crafty | 0.931160 | 0.933189 | 1.003369 |
| eon | 0.946754 | 0.951027 | 1.005580 |
| gap | 0.941232 | 0.944969 | 1.004798 |
| gcc | 0.668263 | 0.699837 | 1.016203 |
| gzip | 0.904307 | 0.916159 | 1.013279 |
| mcf | 0.508848 | 0.509346 | 1.001215 |
| parser | 0.858478 | 0.863596 | 1.006219 |
| perlbmk | 0.911589 | 0.914908 | 1.005298 |
| twolf | 0.715476 | 0.715828 | 1.001357 |
| vortex | 0.820341 | 0.821149 | 1.001923 |
| vpr | 0.715974 | 0.716527 | 1.001627 |

## B.10 Queue Size Variations

### B.10.1 SID: queue size

| Name | Number of queue elements |
|---|---|
| normal | 5 |
| qsize2 | 2 |
| qsize10 | 10 |
| qsize20 | 20 |

### B.10.2 CS: 16-1024; SID: queue size (section B.10.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | | | | |
|---|---|---|---|---|---|---|
| | normal | qsize10 | qsize2 | qsize20 | qsize3 | qsize4 |
| bzip2 | 0.782843 | 0.783095 | 0.642141 | 0.783095 | 0.776612 | 0.781949 |
| crafty | 0.987439 | 0.987476 | 0.978510 | 0.987476 | 0.987072 | 0.987425 |
| eon | 0.999977 | 0.999977 | 0.999903 | 0.999977 | 0.999976 | 0.999976 |
| gap | 0.939804 | 0.940069 | 0.695415 | 0.940069 | 0.932176 | 0.938773 |
| gcc | 0.908137 | 0.908718 | 0.625103 | 0.908719 | 0.895684 | 0.906178 |
| gzip | 0.945378 | 0.945696 | 0.689998 | 0.945696 | 0.934316 | 0.944137 |
| mcf | 0.524216 | 0.524230 | 0.394130 | 0.524230 | 0.512693 | 0.521829 |
| parser | 0.921794 | 0.922072 | 0.821321 | 0.922073 | 0.916719 | 0.921397 |
| perlbmk | 0.999925 | 0.999925 | 0.999683 | 0.999925 | 0.999923 | 0.999925 |
| twolf | 0.845337 | 0.845373 | 0.794018 | 0.845373 | 0.842198 | 0.845107 |
| vortex | 0.957067 | 0.956905 | 0.937345 | 0.956905 | 0.955795 | 0.956866 |
| vpr | 0.871296 | 0.871305 | 0.794742 | 0.871304 | 0.869597 | 0.870710 |

### B.10.3 CS: 32-2048; SID: queue size (section B.10.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | | | | |
|---|---|---|---|---|---|---|
| | normal | qsize10 | qsize2 | qsize20 | qsize3 | qsize4 |
| bzip2 | 0.865854 | 0.865962 | 0.793254 | 0.865966 | 0.861990 | 0.865396 |
| crafty | 0.999456 | 0.999455 | 0.997750 | 0.999455 | 0.999456 | 0.999451 |
| eon | 0.999977 | 0.999977 | 0.999904 | 0.999977 | 0.999976 | 0.999977 |
| gap | 0.939902 | 0.940242 | 0.683104 | 0.940244 | 0.931189 | 0.938721 |
| gcc | 0.902791 | 0.903482 | 0.600477 | 0.903481 | 0.888973 | 0.900677 |
| gzip | 0.943702 | 0.944056 | 0.664122 | 0.944057 | 0.931054 | 0.942347 |
| mcf | 0.562288 | 0.562308 | 0.454802 | 0.562308 | 0.557039 | 0.562460 |
| parser | 0.955722 | 0.955847 | 0.896171 | 0.955847 | 0.952601 | 0.955424 |
| perlbmk | 0.999928 | 0.999928 | 0.999724 | 0.999928 | 0.999927 | 0.999928 |
| twolf | 0.999869 | 0.999869 | 0.999193 | 0.999869 | 0.999866 | 0.999869 |
| vortex | 0.969878 | 0.969865 | 0.957879 | 0.969865 | 0.969107 | 0.969844 |
| vpr | 0.994175 | 0.994176 | 0.983484 | 0.994176 | 0.994461 | 0.994057 |

### B.10.4 CS: 8-256; SID: queue size (section B.10.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | | | | |
|---|---|---|---|---|---|---|
| | normal | qsize10 | qsize2 | qsize20 | qsize3 | qsize4 |
| bzip2 | 0.641587 | 0.642578 | 0.392324 | 0.642584 | 0.628295 | 0.638861 |
| crafty | 0.920315 | 0.920184 | 0.873197 | 0.920184 | 0.918447 | 0.920086 |
| eon | 0.999557 | 0.999557 | 0.998483 | 0.999557 | 0.999540 | 0.999557 |
| gap | 0.934038 | 0.934296 | 0.720634 | 0.934296 | 0.928321 | 0.933219 |
| gcc | 0.913673 | 0.914070 | 0.675505 | 0.914071 | 0.903793 | 0.912112 |
| gzip | 0.887723 | 0.888014 | 0.669299 | 0.888014 | 0.877975 | 0.887879 |
| mcf | 0.448160 | 0.449805 | 0.287385 | 0.449810 | 0.433521 | 0.444409 |
| parser | 0.757447 | 0.758116 | 0.542952 | 0.758116 | 0.746233 | 0.756169 |
| perlbmk | 0.998348 | 0.998348 | 0.995829 | 0.998348 | 0.998436 | 0.998432 |
| twolf | 0.606752 | 0.607321 | 0.470432 | 0.607322 | 0.593912 | 0.604943 |
| vortex | 0.899877 | 0.899470 | 0.861409 | 0.899458 | 0.898312 | 0.900031 |
| vpr | 0.609607 | 0.612983 | 0.467289 | 0.612983 | 0.604226 | 0.611024 |

## B.11 Queue Clocking Variations

### B.11.1 SID: queue performance

| Name | Queue clocking in percent compared to main processor clocking |
|---|---|
| slow-1 | 50 % |
| slow-2 | 33 % |
| slow-3 | 25 % |
| slow-4 | 20 % |
| slow-6 | 14 % |

### B.11.2 CS: 16-1024; SID: queue performance (section B.11.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | | | |
|---|---|---|---|---|---|
| | slow-1 | slow-2 | slow-3 | slow-4 | slow-6 |
| bzip2 | 0.719410 | 0.716422 | 0.713533 | 0.710984 | 0.705109 |
| crafty | 0.924474 | 0.924315 | 0.924129 | 0.923977 | 0.923694 |
| eon | 0.940689 | 0.940688 | 0.940687 | 0.940687 | 0.940685 |
| gap | 0.881304 | 0.878163 | 0.875172 | 0.872210 | 0.866534 |
| gcc | 0.833467 | 0.829032 | 0.824703 | 0.820334 | 0.811546 |
| gzip | 0.884204 | 0.880155 | 0.876203 | 0.872465 | 0.866439 |
| mcf | 0.489847 | 0.485275 | 0.481720 | 0.478800 | 0.471676 |
| parser | 0.842894 | 0.840474 | 0.837913 | 0.835684 | 0.830635 |
| perlbmk | 0.936155 | 0.936152 | 0.936149 | 0.936146 | 0.936139 |
| twolf | 0.789686 | 0.787193 | 0.784510 | 0.782134 | 0.776965 |
| vortex | 0.900594 | 0.900428 | 0.900176 | 0.899943 | 0.899373 |
| vpr | 0.814487 | 0.812747 | 0.810984 | 0.809379 | 0.806128 |

### B.11.3 CS: 32-2048; SID: queue performance (section B.11.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | | | |
|---|---|---|---|---|---|
| | slow-1 | slow-2 | slow-3 | slow-4 | slow-6 |
| bzip2 | 0.796378 | 0.794487 | 0.792599 | 0.790908 | 0.786965 |
| crafty | 0.935405 | 0.935395 | 0.935384 | 0.935374 | 0.935352 |
| eon | 0.940674 | 0.940673 | 0.940672 | 0.940672 | 0.940670 |
| gap | 0.881663 | 0.878589 | 0.875660 | 0.872773 | 0.867278 |
| gcc | 0.830306 | 0.825892 | 0.821588 | 0.817214 | 0.808494 |
| gzip | 0.882704 | 0.878694 | 0.874738 | 0.871049 | 0.865100 |
| mcf | 0.525428 | 0.521680 | 0.518315 | 0.515263 | 0.508701 |
| parser | 0.875590 | 0.874260 | 0.872902 | 0.871642 | 0.868883 |
| perlbmk | 0.936145 | 0.936142 | 0.936139 | 0.936136 | 0.936130 |
| twolf | 0.933970 | 0.933965 | 0.933959 | 0.933954 | 0.933943 |
| vortex | 0.912701 | 0.912581 | 0.912407 | 0.912272 | 0.911997 |
| vpr | 0.929372 | 0.929118 | 0.928843 | 0.928582 | 0.928024 |

### B.11.4 CS: 8-256; SID: queue performance (section B.11.1); BE: P1, REF: cache without prefetching

| Benchmark | Speedups | | | | |
|---|---|---|---|---|---|
| | slow-1 | slow-2 | slow-3 | slow-4 | slow-6 |
| bzip2 | 0.588296 | 0.583474 | 0.578566 | 0.574420 | 0.564252 |
| crafty | 0.861776 | 0.860774 | 0.859633 | 0.858747 | 0.856640 |
| eon | 0.940226 | 0.940193 | 0.940148 | 0.940110 | 0.940010 |
| gap | 0.875777 | 0.872545 | 0.869411 | 0.866373 | 0.860315 |
| gcc | 0.839428 | 0.834990 | 0.830372 | 0.826361 | 0.817446 |
| gzip | 0.830935 | 0.825201 | 0.819340 | 0.813933 | 0.803322 |
| mcf | 0.418767 | 0.414535 | 0.410544 | 0.407110 | 0.399525 |
| parser | 0.693791 | 0.688209 | 0.682505 | 0.677527 | 0.666294 |
| perlbmk | 0.934667 | 0.934606 | 0.934542 | 0.934490 | 0.934345 |
| twolf | 0.566840 | 0.560992 | 0.554811 | 0.549880 | 0.538185 |
| vortex | 0.846521 | 0.845888 | 0.845088 | 0.844355 | 0.842820 |
| vpr | 0.569774 | 0.565619 | 0.561429 | 0.560772 | 0.550397 |

## B.12 Instruction Hashwalk

### B.12.1 SID: instruction hashwalk

| Name | Hashwalk for instructions |
|---|---|
| hashwalk | yes |
| nohashwalk | no |

### B.12.2 CS: 16-1024; SID: instruction hashwalk (section B.12.1); BE: P4, REF: cache without prefetching

| Benchmark | Speedups | |
|---|---|---|
| | hashwalk | nohashwalk |
| bzip2 | 0.922950 | 0.924926 |
| crafty | 0.993634 | 0.994142 |
| eon | 0.994952 | 0.994965 |
| gap | 0.979623 | 0.980140 |
| gcc | 0.782614 | 0.790891 |
| gzip | 0.992095 | 0.992358 |
| mcf | 0.594980 | 0.600986 |
| parser | 0.959436 | 0.961172 |
| perlbmk | 0.996943 | 0.996947 |
| twolf | 0.885721 | 0.892034 |
| vortex | 0.966806 | 0.967976 |
| vpr | 0.895346 | 0.897231 |

### B.12.3 CS: 32-2048; SID: instruction hashwalk (section B.12.1); BE: P4, REF: cache without prefetching

| Benchmark | Speedups | |
|---|---|---|
| | hashwalk | nohashwalk |
| bzip2 | 0.953122 | 0.953847 |
| crafty | 0.996719 | 0.996783 |
| eon | 0.988712 | 0.988718 |
| gap | 0.977281 | 0.977446 |
| gcc | 0.966990 | 0.967455 |
| gzip | 0.989329 | 0.989395 |
| mcf | 0.633881 | 0.640243 |
| parser | 0.976187 | 0.976995 |
| perlbmk | 0.993383 | 0.993390 |
| twolf | 0.989999 | 0.989839 |
| vortex | 0.977088 | 0.977403 |
| vpr | 0.986886 | 0.986453 |

### B.12.4 CS: 8-256; SID: instruction hashwalk (section B.12.1); BE: P4, REF: cache without prefetching

| Benchmark | Speedups | |
|---|---|---|
| | hashwalk | nohashwalk |
| bzip2 | 0.826920 | 0.839724 |
| crafty | 0.920062 | 0.933677 |
| eon | 0.937030 | 0.948152 |
| gap | 0.929650 | 0.942734 |
| gcc | 0.701278 | 0.713898 |
| gzip | 0.891550 | 0.912018 |
| mcf | 0.516596 | 0.518324 |
| parser | 0.850050 | 0.867799 |
| perlbmk | 0.888681 | 0.913562 |
| twolf | 0.712752 | 0.731837 |
| vortex | 0.788642 | 0.824354 |
| vpr | 0.708238 | 0.725181 |

## B.13 Dictionary Entries

### B.13.1 SID: dictionary

| Name | Dictionary Entries |
|---|---|
| dic1 | 1 |
| dic2 | 2 |
| dic4 | 4 |

### B.13.2 CS: 16-1024; SID: dictionary (section B.13.1); BE: P4, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | dic1 | dic2 | dic4 |
| bzip2 | 0.852710 | 0.863091 | 0.920998 |
| crafty | 0.958587 | 0.965119 | 0.994010 |
| eon | 0.969242 | 0.969478 | 0.994965 |
| gap | 0.946572 | 0.954282 | 0.979909 |
| gcc | 0.658761 | 0.672330 | 0.757629 |
| gzip | 0.967924 | 0.970724 | 0.992082 |
| mcf | 0.511751 | 0.524079 | 0.584502 |
| parser | 0.899292 | 0.910065 | 0.957709 |
| perlbmk | 0.962467 | 0.965707 | 0.996929 |
| twolf | 0.809767 | 0.822085 | 0.887343 |
| vortex | 0.891508 | 0.902901 | 0.967257 |
| vpr | 0.801462 | 0.816071 | 0.894831 |

### B.13.3 CS: 32-2048; SID: dictionary (section B.13.1); BE: P4, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | dic1 | dic2 | dic4 |
| bzip2 | 0.897643 | 0.904048 | 0.952455 |
| crafty | 0.977396 | 0.978896 | 0.996773 |
| eon | 0.964959 | 0.964979 | 0.988715 |
| gap | 0.948845 | 0.954420 | 0.977242 |
| gcc | 0.868699 | 0.876917 | 0.961130 |
| gzip | 0.967906 | 0.968862 | 0.989115 |
| mcf | 0.551593 | 0.562391 | 0.623074 |
| parser | 0.941280 | 0.945368 | 0.975321 |
| perlbmk | 0.965861 | 0.967098 | 0.993387 |
| twolf | 0.938989 | 0.944163 | 0.989747 |
| vortex | 0.923692 | 0.930182 | 0.976884 |
| vpr | 0.920904 | 0.930353 | 0.986653 |

### B.13.4 CS: 8-256; SID: dictionary (section B.13.1); BE: P4, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | dic1 | dic2 | dic4 |
| bzip2 | 0.746151 | 0.764382 | 0.829659 |
| crafty | 0.846679 | 0.862456 | 0.931160 |
| eon | 0.882661 | 0.894019 | 0.946754 |
| gap | 0.875501 | 0.887108 | 0.941232 |
| gcc | 0.608448 | 0.630439 | 0.668263 |
| gzip | 0.829484 | 0.846965 | 0.904307 |
| mcf | 0.442257 | 0.456394 | 0.508848 |
| parser | 0.756815 | 0.777580 | 0.858478 |
| perlbmk | 0.827616 | 0.839791 | 0.911589 |
| twolf | 0.633486 | 0.651598 | 0.715476 |
| vortex | 0.704457 | 0.721846 | 0.820341 |
| vpr | 0.629868 | 0.647935 | 0.715974 |

## B.14 Protected Kernel Area

### B.14.1 SID: protected kernel

| Name | Dictionary Entries |
|---|---|
| dic1 | 1 |
| dic2 | 2 |
| dic4 | 4 |

### B.14.2 CS: 16-1024; SID: protected kernel (section B.14.1); BE: P4, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | kernel64 | kernelall | user |
| bzip2 | 0.920998 | 0.854658 | 0.920878 |
| crafty | 0.994010 | 0.960730 | 0.994223 |
| eon | 0.994965 | 0.937545 | 0.994971 |
| gap | 0.979909 | 0.915334 | 0.979905 |
| gcc | 0.757629 | 0.686372 | 0.754159 |
| gzip | 0.992082 | 0.927103 | 0.991981 |
| mcf | 0.584502 | 0.557165 | 0.584581 |
| parser | 0.957709 | 0.893263 | 0.958004 |
| perlbmk | 0.996929 | 0.946504 | 0.996970 |
| twolf | 0.887343 | 0.814512 | 0.887854 |
| vortex | 0.967257 | 0.921698 | 0.967361 |
| vpr | 0.894831 | 0.828391 | 0.896103 |

### B.14.3 CS: 32-2048; SID: protected kernel (section B.14.1); BE: P4, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | kernel64 | kernelall | user |
| bzip2 | 0.952455 | 0.886562 | 0.952316 |
| crafty | 0.996773 | 0.965553 | 0.996764 |
| eon | 0.988715 | 0.926354 | 0.988716 |
| gap | 0.977242 | 0.913049 | 0.977253 |
| gcc | 0.961130 | 0.906560 | 0.961809 |
| gzip | 0.989115 | 0.924711 | 0.989142 |
| mcf | 0.623074 | 0.601883 | 0.621023 |
| parser | 0.975321 | 0.914382 | 0.975314 |
| perlbmk | 0.993387 | 0.941112 | 0.993386 |
| twolf | 0.989747 | 0.931960 | 0.989336 |
| vortex | 0.976884 | 0.934403 | 0.976801 |
| vpr | 0.986653 | 0.931099 | 0.987417 |

### B.14.4 CS: 8-256; SID: protected kernel (section B.14.1); BE: P4, REF: cache without prefetching

| Benchmark | Speedups | | |
|---|---|---|---|
| | kernel64 | kernelall | user |
| bzip2 | 0.829659 | 0.711602 | 0.830251 |
| crafty | 0.931160 | 0.843735 | 0.932341 |
| eon | 0.946754 | 0.821645 | 0.949728 |
| gap | 0.941232 | 0.810715 | 0.940604 |
| gcc | 0.668263 | 0.586472 | 0.669378 |
| gzip | 0.904307 | 0.769291 | 0.905204 |
| mcf | 0.508848 | 0.466415 | 0.511380 |
| parser | 0.858478 | 0.726833 | 0.859269 |
| perlbmk | 0.911589 | 0.780421 | 0.913485 |
| twolf | 0.715476 | 0.602229 | 0.721194 |
| vortex | 0.820341 | 0.696727 | 0.819663 |
| vpr | 0.715974 | 0.605357 | 0.719216 |

# Bibliography


[1] ARM. *AMBA Specification (Revision 2.0)*. ARM, `http://www.arm.com`, arm ihi 0011a edition, May 1999.

[2] T. W. Arnold and L. P. Van Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. *IBM Journal of Research and Development*, 48(3/4):475–487, 2004.

[3] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. Cryptology ePrint Archive, Report 2001/069, 2001. `http://eprint.iacr.org/`.

[4] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of USENIX 2005 Annual Technical Conference*, pages 41–46, 2005.

[5] Philippe Biondi and Fabrice Desclaux. Silver needle in the skype. In *BlackHat Europe*, March 2006.

[6] Andrea Bittau, Mark Handley, and Joshua Lackey. The final nail in WEP's coffin. In *S&P*, pages 386–400, 2006.

[7] Bram Cohen and Ben Laurie. AES-hash. `http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/aes-hash/aeshash.pdf`, May 2001.

[8] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2. `http://www.x86.org/ftp/manuals/tools/elf.pdf`, May 1995.

[9] Intel Corporation. Pentium II Processor Developer's Manual. `http://download.intel.com/design/PentiumII/manuals/24350201.pdf`, October 1997.

[10] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. `http://www.intel.com/design/processor/manuals/253665.pdf`, November 2006.

[11] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M. `http://www.intel.com/design/processor/manuals/253666.pdf`, November 2006.

[12] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z. `http://www.intel.com/design/processor/manuals/253667.pdf`, November 2006.

[13] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. `http://www.intel.com/design/processor/manuals/253668.pdf`, November 2006.

[14] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2. `http://www.intel.com/design/processor/manuals/253669.pdf`, November 2006.

[15] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, Redmond, WA, USA, 2001.

[16] IBM Cryptographic Products. IBM PCI Cryptographic Coprocessor: General Information Manual, May 2002.

[17] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. `http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael-ammended.pdf`, 1999.

[18] T. Dierks and E. Rescorla. RFC 4346: The Transport Layer Security (TLS) protocol version 1.1, April 2006. Status: PROPOSED STANDARD.

[19] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[20] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation. Methods and Techniques*. NIST, 2001.

[21] Konrad Eisele. Design of a memory management unit for system-on-a-chip platform "LEON". `http://www.iti.uni-stuttgart.de/LeonMMU/MMU.pdf`, November 2002.

[22] Donald L. Evans, Philip J. Bond, and Arden L. Bement. *FIPS PUB 140-2: Security Requirements for Cryptographic Modules*. National Institute of Standards and Technology, May 2006.

[23] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 133–143, New York, NY, USA, 1997. ACM Press.

[24] Scott R. Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of rc4. In *SAC '01: Revised Papers from the 8th Annual International Workshop on Selected Areas in Cryptography*, pages 1–24, London, UK, 2001. Springer-Verlag.

[25] William F. (William Frederick) Friedman. *The index of coincidence and its applications in cryptanalysis*, volume 49 of *A cryptographic series*. Aegean Park Press, Laguna Hills, CA, USA, 1987.

[26] Jiri Gaisler. *LEON2 Processor User's Manual - XST Edition (Version 1.0.24)*. Gaisler Research, `http://www.gaisler.com/doc/leon2-1.0.24-xst.pdf`, 2003.

[27] Blaise Gassend, Dwaine Clarke, G. Edward Suh, Marten van Dijk, and Srinivas Devadas. Caches and Hash Trees for Efficient Memory Integrity Verification. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, February 2003.

[28] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon physical random functions. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 148–160, New York, NY, USA, 2002. ACM Press.

[29] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[30] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

[31] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The nizza secure-system architecture. In *Proceedings of the 1st International Conference on Collaborative Computing: Networking, Applications and Worksharing*. IEEE, December 2005.

[32] Mike Hendry. *Smart Card Security and Applications, Second Edition*. Artech House, Inc., Norwood, MA, USA, 2001.

[33] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, third edition, 2003.

[34] Alireza Hodjat and Ingrid Verbauwhede. Speed-area trade-off for 10 to 100 Gbits/s throughput AES processor. In *2003 IEEE Asilomar Conference on Signals, Systems, and Computers*, November 2003.

[35] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. TCP: Tag Correlating Prefetchers. In *The Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*, page 317, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[36] Andrew Huang. Keeping secrets in hardware: the microsoft Xbox(TM) case study. Technical Report AIM-2002-008, MIT TR, May 2002.

[37] Intel Corporation. LaGrande Technology Architectural Overview. `ftp://download.intel.com/technology/security/downloads/LT_Arch_Overview.pdf`, September 2003.

[38] Intel Corporation. LaGrande Technology Preliminary Architecture Specification. `ftp://download.intel.com/technology/security/downloads/PRELIM-LT-SPEC_D52212.pdf`, May 2006.

[39] Ecma International. ECMA-262: ECMAScript Language Specification. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, December 1999.

[40] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 252–263, New York, NY, USA, 1997. ACM Press.

[41] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[42] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Introduction to differential power analysis and related attacks. `http://www.cryptography.com/resources/whitepapers/DPATechInfo.pdf`, 1998.

[43] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 104–113, London, UK, 1996. Springer-Verlag.

[44] François Koeune and François-Xavier Standaert. *A Tutorial on Physical Security and Side-Channel Attacks*, volume 3655 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006.

[45] H. Krawczyk, M. Bellare, and R. Canetti. RFC 2104: HMAC: Keyed-hashing for message authentication, February 1997. Status: INFORMATIONAL.

[46] Markus G. Kuhn and Ross J. Anderson. Soft tempest: Hidden data transmission using electromagnetic emanations. In *Proceedings of the Second International Workshop on Information Hiding*, pages 124–142, London, UK, 1998. Springer-Verlag.

[47] Hoon Jae Lee, ManKi Ahn, Seongan Lim, and Sang-Jae Moon. A study on smart card security evaluation criteria for side channel attacks. In *ICCSA (1)*, pages 517–526, 2004.

[48] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers, 1999.

[49] D. Lie, C. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware, 2003.

[50] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software, 2000.

[51] David J. Lie. *Architectural support for copy and Tamper-Resistant software*. PhD thesis, Stanford University, December 2003.

[52] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[53] Hans Löhr, Hari Govind V. Ramasamy, Ahmad-Reza Sadeghi, Stefan Schulz, Matthias Schunter, and Christian Stüble. Enhancing grid security using trusted virtualization. In Bin Xiao, Laurence Tianruo Yang, Jianhua Ma, Christian Müller-Schloer, and Yu Hua, editors, *4th International Conference on Autonomic and Trusted Computing*, volume 4610 of *Lecture Notes in Computer Science*, pages 372–384. Springer-Verlag, 2007.

[54] Robert Love. *Linux Kernel Development*. Novell Press, 2005. LOV r 05:1 1.Ex.

[55] Steven R. McQueen. Basic RSA encryption engine. `http://www.opencores.org/projects.cgi/web/basicrsa/overview`, October 2003.

[56] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, 1997.

[57] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, `http://www.cacr.math.uwaterloo.ca/hac/`, October 1996.

[58] R. C. Merkle. Protocols for public key cryptosystems. In IEEE, editor, *IEEE Symposium on Security and Privacy*, pages 122–134, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1980. IEEE Computer Society Press.

[59] Microsoft. Next-generation secure computing base. `http://www.microsoft.com/resources/ngscb/`, June 2006.

[60] S. H. K. Narayanan, M. Kandemir, R. Brooks, and Kolcu I. Secure execution of computations on untrusted hosts. In Michael Pinho, Luís Miguel; González Harbour, editor, *Reliable Software Technologies – Ada-Europe 2006 11th Ada-Europe International Conference on Reliable Software Technologies, Porto, Portugal, June 5–9, 2006, Proceedings*, pages 106–118, June 2006.

[61] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard*. National Institute for Standards and Technology, Gaithersburg, MD, USA, April 1995. Supersedes FIPS PUB 180 1993 May 11.

[62] National Institute of Standards and Technology. *FIPS PUB 46-3: Data Encryption Standard (DES)*. National Institute for Standards and Technology, Gaithersburg, MD, USA, October 1999. supersedes FIPS 46-2.

[63] National Institute of Standards and Technology, editor. *Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197, November 2001.

[64] National Security Agency (NSA). AES VHDL model. `http://csrc.nist.gov/CryptoToolkit/aes/round2/r2anlsys.htm`, December 1999.

[65] James Newsome, David Brumley, and Dawn Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the* $13^{th}$ *Annual Network and Distributed Systems Security Symposium*, 2006. `http://www.cs.cmu.edu/~dbrumley/`.

[66] Nathanael Paul and David Evans. Comparing Java and .NET security: Lessons learned and missed. *Computers & Security*, 25(5):338–350, 2006.

[67] Richard P. Paul. *SPARC Architecture Assembly Language Programming, & C*. Prentice-Hall, Inc, 1994.

[68] Colin Percival. Cache missing for fun and profit. BSDCan 2005, `http://www.daemonology.net/papers/htt.pdf`, May 2005.

[69] Birgit Pfitzmann, James Riordan, Christian Stüble, Michael Waidner, and Arnd Weber. The PERSEUS system architecture. In Dirk Fox, Marit Köhntopp, and Andreas Pfitzmann, editors, *VIS 2001, Sicherheit in komplexen IT-Infrastrukturen*, pages 1–18. Vieweg Verlag, 2001.

[70] Jörg Platte, Raúl Durán Díaz, and Edwin Naroska. A new encryption and hashing scheme for the security architecture for microprocessors. In Herbert Leitold and Evangelos P. Markatos, editors, *Communications and Multimedia Security*, volume 4237 of *Lecture Notes in Computer Science*, pages 120–129. Springer Berlin / Heidelberg, October 2006.

[71] Jörg Platte, Raúl Durán Díaz, and Edwin Naroska. An operating system design for the security architecture for microprocessors. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS'06: Proceedings of 8th Conference on Communications and Multimedia Security*, volume 4307 of *Lecture Notes in Computer Science*, pages 174–189. Springer Berlin / Heidelberg, December 2006.

[72] Jörg Platte and Edwin Naroska. A combined hardware and software architecture for secure computing. In *CF '05: Proceedings of the 2nd conference on Computing Frontiers*, pages 280–288, New York, NY, USA, May 2005. ACM Press.

[73] Jörg Platte, Edwin Naroska, and Kai Grundmann. A cache design for a security architecture for microprocessors (SAM). In Werner Grass, Bernhard Sick, and Klaus Waldschmidt, editors, *Architecture of Computing Systems*, volume 3894 of *Lecture Notes in Computer Science*, pages 435–449. Springer Berlin / Heidelberg, February 2006.

[74] Bart Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke Universiteit Leuven (Belgium), February 1993.

[75] W. Rankl and Wolfgang Effing. *Smart Card Handbook*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[76] R. Rivest. RFC 1321: The MD5 message-digest algorithm, April 1992. Status: INFORMATIONAL.

[77] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, February 1978. The basics of trap-door functions and the famous RSA public key cryptosystem are presented in this paper.

[78] Brian Rogers, Yan Solihin, and Milos Prvulovic. Memory predecryption: hiding the latency overhead of memory encryption. *SIGARCH Comput. Archit. News*, 33(1):27–33, 2005.

[79] Ishan Sachdev. Development of a programming model for the AEGIS secure processor. Master's thesis, Massachusetts Institute of Technology, May 2005.

[80] Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, pages 44–60, London, UK, 1998. Springer-Verlag.

[81] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[82] P. Schramm, E. Naroska, P. Resch, J. Platte, H. Linde, G. Stromberg, and T. Sturm. A service gateway for networked sensor systems. *IEEE Pervasive Computing*, 3(1):66–74, 2004.

[83] Timothy Sherwood, Suleyman Sair, and Brad Calder. Predictor-directed stream buffers. In *International Symposium on Microarchitecture*, pages 42–53, 2000.

[84] Weidong Shi, Hsien-Hsin S. Lee, Chenghuai Lu, and Mrinmoy Ghosh. Towards the issues in architectural support for protection of software execution. *SIGARCH Comput. Archit. News*, 33(1):6–15, 2005.

[85] SPARC International Inc. *The Sparc Architecture Manual Version 8*. SPARC International Inc., `http://www.sparc.com`, 1991.

[86] Michael Steil. 17 mistakes microsoft made in the Xbox security system. `http://www.xbox-linux.org`, presented on 22nd Chaos Communication Congress, December 2005.

[87] Felix Streichert and Holger Ulmer. JavaEvA - a java framework for evolutionary algorithms. Technical Report WSI-2005-06, Centre for Bioinformatics Tübingen, University of Tübingen, 2005.

[88] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Hardware mechanisms for memory integrity checking, 2002.

[89] Gookwon Edward Suh. *AEGIS: A Single-Chip Secure Processor*. PhD thesis, Massachusetts Institute of Technology, September 2005.

[90] Sun Microsystems. Java card security white paper. `http://java.sun.com/products/javacard/JavaCardSecurityWhitePaper.pdf`, October 2001.

[91] Erik Tews, Ralf-Philipp Weinmann, and Andrei Pyshkin. Breaking 104 bit wep in less than 60 seconds. Cryptology ePrint Archive, Report 2007/120, 2007. `http://eprint.iacr.org/`.

[92] Trusted Computing Group. TPM main part 1 design principle, specification version 1.2, revision 94. `https://www.trustedcomputinggroup.org/groups/tpm`, March 2006.

[93] Trusted Computing Group. Trusted computing group. `https://www.trustedcomputinggroup.org/`, 2006.

[94] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions md4, md5, haval-128 and ripemd. Cryptology ePrint Archive, Report 2004/199, 2004. `http://eprint.iacr.org/`.

[95] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Collision search attacks on sha1. `http://theory.csail.mit.edu/~yiqun/shanote.pdf`, february 2005.

[96] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In *SRDS*, pages 260–. IEEE Computer Society, 2003.

[97] Yves Younan, Wouter Joosen, and Frank Piessens. A methodology for designing countermeasures against current and future code injection attacks. In *IWIA '05: Proceedings of the Third IEEE International Workshop on Information Assurance (IWIA'05)*, pages 3–20, Washington, DC, USA, 2005. IEEE Computer Society.

[98] Yves Younan, Wouter Joosen, and Frank Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In *ICICS*, pages 379–398, 2006.

[99] Yves Younan, Davide Pozza, Frank Piessens, and Wouter Joosen. Extended protection against stack smashing attacks without performance loss. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 429–438, Washington, DC, USA, 2006. IEEE Computer Society.

[100] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. *SIGPLAN Not.*, 39(11):72–84, 2004.