Hardware-Enforced Fine-Grained Isolation of Untrusted Code

by Eugen Leontie

B.S. in Computer Science, June 2004, "Politehnica" University Of Bucharest
M.S in Computer Science, June 2005, "Politehnica" University Of Bucharest

A Dissertation submitted to

the Faculty of
The School of Engineering and Applied Science
of The George Washington University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

August 31, 2012

Dissertation directed by

Bhagirath Narahari,
Professor of Engineering and Applied Science
Rahul Simha,
Professor of Engineering and Applied Science

The School of Engineering and Applied Science of The George Washington University certifies that Eugen Leontie has passed the Final Examination for the degree of Doctor of Philosophy as of July 25, 2012. This is the final and approved form of the dissertation.

Hardware-Enforced Fine-Grained Isolation of Untrusted Code

Eugen Leontie

Dissertation Research Committee:

Bhagirath Narahari, Professor of Engineering and Applied Science, Dissertation Co-Director
Rahul Simha, Professor of Engineering and Applied Science, Dissertation Co-Director
Gabriel Parmer, Assistant Professor of Computer Science, Committee Member
Nan Zhang, Associate Professor of Computer Science, Committee Member
Guru Prasadh Venkataramani, Assistant Professor of Engineering and Applied Science, Committee Member

Abstract of Dissertation

Hardware-Enforced Fine-Grained Isolation of Untrusted Code

The concept of isolation in a computer system was first considered when the need for rare and expensive computing resources forced businesses to allow multiple individuals to simultaneously access mainframe computers. Once the information and data assets of multiple people shared the same physical medium, confidentiality and fairness issues arose. They were addressed with hardware and software mechanisms that isolated data and computing assets of different users. The personal computer era did not eliminate the need for access restriction on stored data. As programming paradigms were introduced, the security mechanisms strove to keep up. Both operating systems and consumer applications became overly complex, with intricate sharing and isolation requirements. Currently used architectures prevent users and processes from accidentally or maliciously altering the memory space of other entities sharing the same physical resources. The partitioning mechanisms rely on the operating system with support from the hardware architecture to enforce the isolation and mitigate the sharing of information and resources.

Yet, the isolation mechanisms are insufficient for current software development cycles and architectures. Software is rarely created by a single trusted entity. It is rather a well orchestrated collage of multiple libraries, open source code and run-time loaded plug-ins, created by multiple vendors. When linked into the final software application, all code runs with the same privileges and shares access to data structures. Under current implementations of memory protection mechanisms this is known to be a major source of security problems. Unless well isolated, software vulnerabilities in uncritical sections of the application can easily affect the safety critical regions and compromise the entire system.

This research addresses the problem of how to reliably engineer systems when mutually untrusted code segments originating from multiple vendors are bundled into a single application. The solution enforces the least privilege security principle and enhances the accountability of software faults. It is designed to offer fine-grained isolation within one memory space. This research effort presents a mechanism that allows a high level of customization to reflect specific security policies and performance requirements. Preliminary prototypes and performance measurements show acceptable performance overhead while adding strong security enforcement. The thesis also quantifies the benefit of additional hardware resources to speedup security checks, identifies typical scenarios used in modern programming paradigms, and describes compiler techniques to abstract the security hardware from the application developer. As performance impact is a very important factor in the adoption of any such hardware addition, we use industry standard benchmarks to assess the overheads incurred by applying different protection mechanisms and show how optimization techniques affect the run-time performance.

Table of Contents

Abstract of Dissertation . . . . . **iii**

1 Introduction **1**

1.1 Motivation 2

1.2 Objectives 4

1.3 Summary of Approach 7

1.4 Contributions 8

1.5 Summary of accomplishments 12

1.6 Outline 13

2 Related Work **14**

2.1 OS and hardware support for isolation 14

2.2 Fine-Grained Memory Protection 16

2.3 Capability systems 22

2.4 Virtualization 23

2.5 Information Flow Tracking 26

2.6 Stack and Heap Protection 28

2.7 Control flow 33

2.8 Architectural Support for Debugging 34

2.9 Physical Security 36

2.10 Recovery 37

3 Problem model **39**

3.1 Source code 40

3.2 Plug-ins 42

3.3 Device Drivers 43

3.4 Buffer Overflow 44
3.5 Conclusion 46

4 Solution Details **47**

4.1 Permission tables 47
4.2 Container Manager 51
4.3 Container Identification Table 54
4.4 Static container permission record 55
4.5 Memory Permission Types 56
4.6 Dynamic Memory Challenges 64
4.7 Sparse Data Structures 66
4.8 Architectural challenges 67
4.8.1 Pipeline Integration 69
4.9 Accelerate Permission List Searches 71
4.10 Object Oriented Design 73
4.10.1 Container private data 76
4.10.2 Exceptions 79
4.11 Containers in multiprocessing context 86
4.11.1 Concurrency Support 88
4.12 Performance Considerations 90
4.13 Other Considerations 91

5 Experimental Results **92**

5.1 Initial Experiments 92
5.2 Advanced Experiments 98

6 Conclusions **108**

Chapter 1 – Introduction

Software development can be trivially reduced to the process of translating user requirements into a form that a computer system can interpret and execute. The complexity of such systems vary from dedicated appliances that only perform a simple set of tasks to collaborating massively distributed supercomputers. Yet at the very base of each is a computer program, the result of translating high level specification requirements into machine language. The advances in hardware architectures, the evolution of software paradigms, all motivated by a constant demand from consumers, led to complex design and implementation models for the software systems. Writing a program is not a task for a single engineer but a collaborative effort of software architects, library vendors, programmers, contractors, testers and tools. Using third party libraries, tools and source code - either proprietary or open source - is a typical choice of system architects in order to reduce the production costs and the time to market of their products.

Validating the correctness of code, tools and libraries is in itself a complex task, and there are only a few trivial cases where guarantees can be offered. Most of the time one has to rely on the developer community or the reputation of a vendor for the correctness and reliability of the code. Once the software development process is complete, all bundled code runs under unified credentials and shares the memory space of the whole process. With currently deployed protection mechanisms this is known to be a major source of security related problems. Unless well isolated, software vulnerabilities in uncritical sections of the application can easily affect safety critical regions of memory.

In this research effort, we design a method to isolate software modules from each other. The architectural and software design techniques are directed to thwart attacks that result in unauthorized access to memory and to maxi-

mize the accountability of the code where the attack originated. We design a hardware unit to act as a reference monitor and check each memory access for compliance to a strict set of rules. This unit will manage granting access for principals (snippets of code that execute on the processors) to the memory regions they try to access, as well as mitigate the interactions between principals. We show how the hardware checker is built, how to derive the rules for typical software protection policies and what the performance impact of our design is.

1.1 Motivation

Two trends in the software industry today are accelerating the incorporation of untrusted code in applications. In the first, feature-rich applications – examples include browsers and media players, as well as servers [100] – are explicitly designed to allow users to customize their experience by downloading third-party *plugins*, executables that are directly incorporated into the execution environment of the application. The second trend is the continuing growth of open-source software, which not only allows sophisticated users to modify code but to easily share such modified code with other users. In both cases, by design, untrusted code runs in the same *process* as the application and with the same privileges, for which reason existing OS-level protection is insufficient to protect the application from a malicious plugin or from vulnerabilities therein [57]. For example, a report by Symantec [103] shows that no fewer than 300 threats for web browsers originate from security flaws in popular add-ons (e.g. JVM, QuickTime, Adobe PDF reader, Adobe Flash player) [31].

For many years now, commodity hardware and operating systems have offered protection at the page-level (process `A` may not access page `B`) and at the process-level (process `X` cannot access process `Y`'s memory). Recently, several

research efforts have focused on offering memory protection to processes (or threads) in smaller chunks, as small as a single word [114], or have proposed carefully managing the interactions between processes [36, 35]. While valuable for processes, such *process-oriented memory protection* approaches have two drawbacks for the kinds of applications we target – applications that use plugin architectures or incorporate open-source packages. The first drawback is that, to use process-oriented memory protection, software that is written using traditional function call-return semantics (in languages such as C or C++) needs to be rewritten to separate functions into processes – a significant burden on developers, especially for legacy code. The second drawback is that inter-process communication (IPC), necessary for an application divided into multiple processes, carries much more overhead than a simple function call, even with operating systems that optimize IPC. In view of these drawbacks, what is desirable is the type of isolation semantics provided by Java: an application can load untrusted code into a pre-defined security sandbox, can call functions in that code, and has the (virtual) machine enforce security. While the Java Virtual Machine provides such isolation for bytecode, our goal is to provide the same for native applications.

Several other scenarios motivate the need for fine-grained memory access protection for embedded devices. Most modern gadgets on the market, such as mobile phones, GPS navigators, and game consoles, are operated by a complex operating system (OS) with several layers of third party applications. Turning a GPS navigator into a movie player, or forcing the game console into running unlicensed software, seems now a trivial exercise, as hacks appear days after the release of these devices. Breaking the manufacturer's protection mechanism is seldom done through custom wiring or sophisticated physical attacks, but by exploiting vulnerabilities in the installed applications. For example, a flaw in the software the iPhone and Sony PSP use to handle tiff

files [1],[2] can be exploited to allow users to run code that was not originally signed by the manufacturer, extending the use of the underlying hardware to run unlicensed applications. Once a hack is published, a non-technical person can follow the instructions and reproduce it with ease. Although these attacks seem to cause only marginal revenue loss for the manufacturer, similar attacks may target cryptographic devices, disclosing security sensitive data, such as private keys. If users can exploit software flaws to bypass licensing restrictions, a malicious user could potentially exploit software flaws to cause supposedly secure and trusted devices to exhibit arbitrary behavior. Wrapping well designed memory constraints around all applications can isolate software flaws and prevent the escalation of attacks. The performance of an application is decreased by enforcing memory access permissions, but the penalty can be reduced by eliminating memory copy operations between layers of the OS or among application contexts.

1.2 Objectives

Even in systems that have security as their main objective, design and implementation faults hide vulnerabilities that can damage the system. The vulnerabilities are usually concealed within intricate levels of complexity. The uncertainty of not knowing when a vulnerability will maliciously or accidentally be activated may always threaten the security of protected assets. Another issue is that after deployment, vulnerabilities are typically found by hackers rather than testers, with considerable losses. Architecture and design strategies that systematically eliminate vulnerabilities are a topic of a lot of research activity, yet no complete method applicable to large complex systems exists. We argue that system software developers should consider code modules as collaborating, untrusted entities, and the failure of one or more of these entities should

be expected and mitigated. Failing or vulnerable code modules should only have restricted access to the memory system, thus minimizing the impact of a failure to other components and to the stability of the system. We next describe a set of security design principles and explain how our solution will achieve each goal:

Least privilege Every code module should operate in strictly controlled isolation. Access to memory should be restricted to a minimal set that allows the module to complete its job. Any deviation from expected behavior can only corrupt or compromise the information within its permitted bounds. This also increases the chances for a violation to be detected by the monitoring hardware, minimizing the code base that needs to be audited in case of a malfunction.

Complete mediation Every memory access is checked against the list of permissible actions. The source of every memory access is automatically derived based on the definition of code modules and it is the responsibility of the reference monitor to keep track of every memory access and who generated it. This way a change in the active code module will reflect a reload of permission lists. This prevents unauthorized delegation of authority and avoids mixes between modules and permissions. Another resource that we propose to monitor is processing time. The reference monitor times each module and interrupts the execution if a timeout is reached.

Restricted access Unless explicitly granted permission, by default, each code module will be denied access to memory and prevented from executing. This conservative design differs from approaches supported by current architectures, where the protection algorithms and mechanisms attempt to identify situations where access should be refused. Such designs are

prone to misidentify protection scenarios, thus leaving room for undetected vulnerabilities. If a default denial is adopted as a base policy, a faulty implementation of permission assignments will more likely lead to an access refusal - a situation that can be detected and rectified. In the opposite approach, an imperfect design of an explicit denial policy will fail with an undetected and possibly critical information leak or corruption.

Ease of adoption It is essential that for ease of deployment, human factors need to be taken into consideration. Overly complex schemes will more likely be circumvented or disabled, whereas simple, easy to understand and implementable mechanisms are more likely to be respected. This is why we rely on back-end tools for most permission extraction, while offering a simple and efficient API that the programmers and designers can use without too much disruption in the design process. We argue against redesign of programming languages, as these typically have a slow acceptance rate. Instead, our design brings a minimal yet powerful API to the programming level, adds minimal changes to the underlying hardware architecture, and offers legacy support for existing code and libraries. Backward compatibility of legacy code can be fully supported and enhancements easily added even if the source code is not available for instrumentation.

Hardware Simplicity The monitor's logic (a hardware unit integrated into the processor) has to be simple and easy to verify for correctness. Inspection of both the algorithms and the actual implementation of the unit should be easy to validate by human examination. Coverage testing is also eased if the state space of the checker is relatively small.

1.3 Summary of Approach

As both current techniques and state-of-the-art solutions do not offer sufficient protection for current software security requirements, we propose a solution designed to support the described objectives. At the core of this solution is the reference monitor. It consists of a hardware module that checks every memory request generated by the processor. The monitor keeps a list of all code modules in a process, and keeps track of the currently executing context. It is able to load on each context switch the set of allowed actions for each module and enforce each memory access generated at runtime to conform with the restrictions. At the same time, the execution time is tracked for each context and verified against a predefined timeout. If a violation occurs, the reference monitor triggers a recovery protocol, also defined at the code module level.

A second part of our solution is to analyze typical requirements of programming structures and paradigms and provide a mechanism for defining the set of restrictions the reference monitor is responsible for enforcing. We take into consideration software engineering challenges, such as availability of source code and ease of adoption, compiler tool chain designs for automatic extraction of program properties, and source code annotations for dynamic memory structures.

Optimizations and validation are another core part of our research. We explore multiple cache configurations to reduce the overhead of loading permission lists from memory and exploit locality properties of the code. An aspect with a great impact on performance is clustering techniques for grouping closely related code modules into functional units with unified protection.

1.4 Contributions

Our architectural changes are designed with the particular goal of restricting the access of well-separated code regions to only memory locations that they need for correct functionality, as required by the principle of least privilege. Our design achieves the following properties: first, it can arbitrarily assign permissions to regions of code as small as the size of an addressable word in memory; second, in order to increase accountability, it reduces the size of a code module to an individual function (or a grouping of functions); third, it shows how the permission tables are derived from the program structure. The design is scalable, flexible and covers general scenarios such as self-modifying code, pre-compiled libraries or assembly imported into code. We also add concurrency support to the container model: instead of having multiple execution sequences (threads) inside a single protection domain, containers supports multiple protection domains inside a single execution sequence.

At the micro architecture level, the hardware modifications are flexible and we can easily adjust our technique to different pipeline architectures. For example when tight coupling with all stages of the pipeline is not possible, we developed a stall and validate strategy. Alternatively a fetch-forward solution optimizes the permission validation by forwarding instruction addresses to the permission checker.

Our testbench includes a specialized cycle accurate processor simulator, with plug-ins for memory tracing, instruction set extensions and hardware modules enhancements. For multithreaded applications our reference monitor requires systems software support for which we modified RTEMS (Real-Time Executive for Multiprocessor Systems) - a POSIX complaint embedded operating system. On this system, we evaluated our implementation of architectural and software modifications using a comprehensive set of commercially repre-

sentative embedded systems benchmarks and several synthetic microbenchmarks that stress specific aspects of the system implementation and design.

Based on the same protection principle, we identified another major utilization for our hardware. Access specifiers in Object-Oriented Programming (OOP) limit object interactions to publicly defined interfaces, but only with compile-time checks. Thus C++ programmers are taught that private is not secure, because encapsulation is broken easily if a program uses direct memory access (pointers). We show how we can enforce encapsulation for C++ programs.

Our work improves the state-of-the-art by demonstrating that modern systems can use fine-grained principals and protection domains efficiently even when considering complex processor pipelines and concurrent applications. This work differs from existing approaches in the following ways:

The granularity of protection (in both terms of code and data) While state-of-the-art solutions allow memory permissions to be assigned to small granular memory regions [114], the principal targeted is still coarse. The protection domain targeted is at the process or thread level, and changing the protection domain requires a trap call into the kernel space to mediate the security context switch. Since our attack model splits a process into a multitude of mutual untrusted parts that frequently switch execution context, we propose a fast context switch unmediated by the supervisor process. Typical benchmarks analysis revealed the size of the access permission list associated with function calls to be small enough such that fast content addressable memory could be used for parallel search when matching memory calls against permission lists.

An intra-process protection mechanism without kernel calls Most dedicated embedded systems are built on a fixed physical memory space,

often with a minimal operating system statically linked into the same address space as the software bundle. Principal detection, access control checks, and delegation for both static and dynamic memory can be implemented in a hardware unit that can be conveniently used both with physical and virtual memory space systems.

A hybrid approach for static and dynamic memory allocation Dynamic memory access control poses the following challenges: the location of data structures is not known until run-time, when the allocation occurs. The reference monitor needs to have a mechanism to attribute permissions to these types of structures, and the CPU needs a set of operations to execute in cooperation with the reference monitor. While a completely dynamic access control model [85] describes such primitives, it has two drawbacks: it does not utilize the information that the compiler can generate about static data structures – with a heavy impact on the performance penalty – and with only a limited storage for the access control lists can only protect a small fraction of memory regions. Our model accounts for both statically derived permissions and dynamic memory allocation.

A time constraint for container execution A malicious or faulty implementation can drive a system into a continuous loop that renders the device inoperable. If time constraints are known for operations, we believe they should be enforced by the reference monitor. For each code module we add one extra constraint: execution time. This metric can be either defined by the manufacturer or obtained during the product testing and can be described both in terms of a maximum or a minimum bound. While we can easily see that an upper bound detection is a preventive measure against processing resources takeover by a mod-

ule, a lower bound can be also useful to detect software malfunctions. For example, if a block cypher encryption algorithm finishes in less than the number of rounds it should, it is a clear indication that either the implementation is faulty or that the original control flow is subverted.

Permission cashing with minimum impact on the cache architecture While most existing processor extensions for memory access control require a special caching [6] or base registers extension [114], our design relies on a typical caching scheme and for accelerating access control lists from memory by either using the existing cache system, or by adding an extra permission cache that shares the same implementation with the typical code or data caches available on the system.

Strictly controlled permission delegation We added a delegation permission to the typical authorization mechanisms (read, write, execute). This allows us to strictly control how code modules can transfer permissions to other modules. The reference monitor checks that no escalation of authorizations occurs. For example, a module that has read and write authorization on a given memory region can only delegate a subset of its authorization (read or write, but not execute) on a subset of the same memory region to other modules that it calls. The extra delegate permission can restrict access to memory such that a module can operate on the memory region but can not pass the permissions on the same memory regions to other modules.

Validation and performance measurements Our testing procedure uses trace data from typical benchmark suites from the embedded systems world, creates access control lists based on the trace data and simulates the execution of the reference monitor. The evaluation we obtain does not rely on artificially generated protection policies, but on the actual

requirements of typical software deployed in embedded devices.

1.5 Summary of accomplishments

Our work to date consists of the architecture model and algorithms for the hardware reference monitor, the compiler and simulator instrumentations for extracting the software security properties and the simulation platform for validating the performance overhead of our proposed techniques for a target architecture [49, 50].

For our initial set of evaluations, we implemented a cycle accurate simulation environment that will provide a precise, independent evaluation of the performance penalties incurred on a comprehensive set of benchmarks. Our simulator was based on the SimpleScalar toolset [7], which was modified to simulate the reference monitor's operations. Another major change to the simulator was adding run-time profiling capabilities that allow us to extract program properties, which are used for extracting the permission definitions for each individual code module, as well as for clustering modules.

In order to evaluate our work in more complex environment, including multiprocessing, OS support and processor pipeline integrations we upgraded our simulation environment. Our new evaluation platforms consists of a combination of systems that allow for both cycle accurate performance measurements, full system simulation and flexible modeling of hardware characteristics. WindRiver Simics [59] is the supporting system simulator and stands at the root of the platform. GEMS/Opal [61] is a separate module that allows a cycle accurate micro-architecture simulator and cache model for simulated architecture. The processor model we use is modeled after the UltraSPARC III architecture. We modeled our fine grained memory protection module as a collection of GEMS extensions and Simics plugins. The software environment is based

on the POSIX compliant real-time embedded operating system: RTEMS [78], enhanced with a new Board Support Package for the Sparc64 architecture.

On this platform we implemented, evaluated and shown that HW enforced fine-grained memory protection is a viable solution for security-critical applications. The fine granularity of protection allows for memory access control without requiring programmers to hand-code page alignments or use inter-process communication when changing principals or protection domains. [48]. We also proposed a mechanism for protecting visibility attributes in object oriented languages such as C++ and analyzed current exception implementations and how they affect the container based protection solution.[47].

1.6 Outline

The remainder of this document is organized as follows. In Chapter 2, we review existing software protection techniques, both closely related to our scheme and general mechanisms that can be combined with our scheme for added security. Chapter 3 describes the problem model and in Chapter 4 we present architectural and implementation details of our scheme. Chapter 5 is dedicated to performance measurements and quantitative analysis, while Chapter 6 concludes the theses and underlines our view of future research trends and follow-on work.

Chapter 2 – Related Work

The idea of isolation dates back to the first generation of time-sharing machines. We adopt the terminology of Saltzer and Schroeder [79] for memory protection: a *domain* is the set of objects (memory) that currently may be accessed by a *principal*, the entity to which authorizations are granted. We use principal to mean the code that executes, and in our work the principal is as fine-grained as a single function invocation. Note that protection domain is often used interchangeably with principal, to refer to the protection context with which code executes; we find it convenient to differentiate between the memory being accessed and the code doing the accessing, so we will stick with domain and principal.

This section provides an overview of past research on protecting computer information from unauthorized access. The focus will be on processor and memory architecture designs, as these are closely related to this thesis topic.

2.1 OS and hardware support for isolation

Typical multi-user computing systems rely on the OS to enforce security policies and prevent each application from corrupting or disclosing address spaces belonging to other applications or to acquire higher level privileges usually reserved for the OS kernel. In order to enforce this isolation, the OS uses hardware primitives specifically designed for this purpose, such as the paged virtual memory or the segmentation systems. The assignment and revocation of permissions is managed by the OS, with the process as the principal.

Although paged-based virtual memory systems allow for process isolation and some controlled sharing (for example, by mapping the same multiple virtual addresses to the same physical address), the granularity of permission

is coarse as permissions can only be assigned to full pages (the typical page size is 4KB). Since the page is the smallest domain, an application that needs to isolate small chunks of memory must place each chunk in its own page. Another disadvantage is that any operation that requires changes in the page tables (either reloading or explicit modifications) involve system calls and context switches, which would be inefficient for the frequent changes required by our isolation objectives. Thus, a multithreaded application cannot isolate the memory accesses among its threads when the process is the smallest principal. The implication for high performance servers is that all services belonging to a particular application share a principal, and therefore will have identical permissions to access memory.

As opposed to the fixed size granularity of page-based systems, segmentation (another available memory protection mechanism in today's processors), allows division of memory in several variable length memory regions called segments. In order to access a memory location, a program needs to first set the segment that it needs access to and then provide an offset within that segment. The segment permissions and mappings are managed, just like paging systems, by the OS. The permission is encoded as a level, a process with a low level access permission can not access a higher level segment, unless by explicitly requesting a privilege switch. This can be accomplished by using system calls, call gates, traps and interrupts. For example, multiple private stacks for functions within one program can be created using a unique dedicated segment for each of the stacks. Frequent context switches still are problematic since sharing and changing the segment mapping still call the OS, not to mention that a typical memory access involves two operations, first to choose the segment descriptor and then the offset. OS/2 [23] is among a few modern commercial OS that used the full featured segmentation system, most other open or commercial operating systems disregard or make only limited usage

of segmentation.

2.2 Fine-Grained Memory Protection

The following state-of-the-art research are closely related to the work in this thesis. They all address the problem of fine-grained memory access control. Upon close examination, they suffer from several shortcomings when applied to our problem space. The shortcomings come either in the area of scalability ([85]), inadequate principal granularity ([114],[5]) or address a significantly different problem ([19]).

Page-level permissions have been employed to create read-only regions of data for canary-like protections schemes [19]. These schemes group memory objects with the same permissions together, and place different groups in different memory pages, so the OS can differentiate between access permissions in the same application memory space and allow different applications to share some memory content. Such systems have a coarse granularity level, and performance degrades quickly as the number of access levels increases. For every other protection domain being accessed, a page table access and permission check occurs, requiring an invocation of the supervisor with a high memory penalty. They also fail to provide the mechanism to protect the same data region differently from separate code zones in the same application, because the data region is protected by the access control of the memory page in which it resides.

Pande et al. [121] use extensive compile time analysis to identify protection zones (memory objects) and mark them "Writable" before each write operation, defaulting back to "Read-Only" when the operation completes. Only one bit per protection zone is used for permission specification. Buffer overflows are thus prevented from affecting memory objects that are not part of the cur-

rent writable memory object. The high frequency of new primitive invocation leads to high overheads, and is vulnerable to maliciously compiled code that misuses the architecture to gain access to critical regions.

InfoShield by Shi et al. [85] takes a step further in assuring that only legitimate code zones can gain access to protected memory regions. It does that by implementing a hardware reference monitor that holds the access permissions for a limited number of memory locations. The permissions are handled by a set of new specialized instructions that allow for creation, verification, clearing and secure moving. The assumption is that only a few security critical memory locations exist during a process execution, and that a single fixed size table is sufficient to cover all protection needs for a program. Due to spatial constraints in the run-time protection tables only a few critical regions can be protected. In typical use cases, keys, passwords, and other secret information are protected against malicious memory scans. The limited protection zones, as well as the performance penalties that come with the extra instructions executed, make the scheme impractical to extend across the entire memory address space.

Having a statically built permission table raises the problem of how to better create, store, and manipulate the table in memory, while still making efficient permission checks. A data-centric approach is to designate permissions for every address range (start/end address pairs, or start address with size) representing each memory object. When a load/store operation is executed, the corresponding permission is fetched from memory together with the content of the address and the designated hardware enforces the permission check. Caching of permissions can be done in conjunction with the existing hardware architecture [5] or by using a separate caching structure [114, 87].

Mondrian Memory Protection [114] uses a hierarchical permission structure to offer word level permission granularity. Traversing the trie structure

for permission validation is performed by a hardware implemented checker. A TLB-based caching structure accelerates permission checks based on locality. The encoding mechanism for encoding the permissions is very compact, thus minimizing the amount of memory that needs to be reserved for permission storage. Unfortunately, compaction makes permission changes complex, and frequent changes to the permissions results in frequent trie node insertions and deletions which are difficult to support in hardware and slow if implemented in software. Also, Mondrian is targeted towards protecting large code modules whose interaction is mediated by an OS/supervisor. As such, the OS/supervisor needs to be invoked every time a security context switch occurs. (In our work, the security context switch is achieved by the hardware, thus frequent context switches are possible with minimal overhead.)

Another shortcoming is the lack of primitives to handle dynamic memory. In the Mondrian approach, permissions are mostly static; for dynamic memory, permissions are passed through the operating system, which means that code has to instrumented with such OS-calls or these calls have to be inserted by the programmer. Furthermore, the OS needs to be augmented to handle these calls. Since memory allocation changes the permissions lists, special primitives need to be designed in order to allow this interaction for requesting and/or granting permissions. Our approach to handling dynamic memory uses a stack-like approach so that permissions are not persistent but aligned with the activation record of the code. The Mondrian memory protection is entirely focused on data and does not extend to cover code protection and control flow, whereas this feature is central to our approach. In general, however, their seminal work provides a foundation upon which we build additional features.

Arora et al. [5] show a solution for data properties enforcement at run-time. By tagging data addresses with an additional security tag, some properties can be enforced at run-time. For example using a single bit SECTAG can mark

fine grained read-only memory regions. The checker (reference monitor) can be programmatically set to interpret and enforce the value of the security tag as specified by a program-wide security policy. One policy example is to label code zones with specific security attributes. All zones with the same label share the access on all data regions with the SECTAG equal or less in value. A great part of the work consists in describing the caching mechanism for the security tag. This is accomplished by changing the cache hierarchy in two steps. First, the L2 cache is enhanced to automatically load the associated tags for the cache lines stored. The inner L1 cache controller expands the cache line with the security attributes for faster checking. The checker directly uses the L1 security tag information for enforcing the policy. Even though the SECTAG length can be varied based on the security policy, the length is fixed once the processor architecture is decided. Any changes in SECTAG length means changing the internal cache layout and size.

While their solution offers some flexibility in terms of supported policies, it does not allow arbitrary limitations for code zones. The authors also assume that the code itself is not intended to be malicious and will not explicitly disable the protection mechanism. This solution offers limited support for multiple code-zone protection, and implies the arbitration of an operating system. If such a technique is used with a fine-grained division of source code, the overhead of the context switch becomes intolerable for most applications as the number of security attributes increases. The proposed solution lacks a static extraction of data properties and solely relies on code instrumentation. Without a statically built permission tags (a permission map of the entire address space) the model does not suit a system with small code partitions and frequent security context switching. Extending the existing solutions would either mean high penalties on cache misses, due to fetching a list of permissions for every memory object, or creating a permission map for every partition of

the code.

MESA [87] describes an architecture for a secure and authenticated memory system. It combines both access control and tamper-proof (encryption based) authentication in a unified system. Code and data are divided into memory capsules that can be privately owned by executing principals. The principals correspond to executing code in a typical computing system such as dynamic libraries, shared libraries or application specific executing code. Managing all these is a secure kernel that keeps the mapping between memory capsules, principals and processes. MESA relies on private heap and stack space for every principal. This includes separate encryption keys and authenticated access to the specialized memory capsules. Although no specialized heap manager is required, the principals need to explicitly issue instructions to switch between private stacks.

Controlled sharing of resources is problematic in MESA as the private heap and stacks force inter-principal sharing to require shared keys and explicit permission delegation. Sharing is permitted only by using specialized data types (secure pointers) that encode both the memory location, size and a location ID into a hardware access control table named the Secure Pointer Buffer. The buffer is a shared hardware controlled table managed (similar to Infoshield, but for sharing as oppose to restricting) that is prone to scalability problems. The restrictions imposed by the architectural design of MESA require major revisions to how software compilers manage their control structures. MESA would be difficult to implement if small disjoint principals (such as placing every function in a principal) require frequent and secure sharing of resources (parameter passing).

Reconfigurable hardware poses a similar challenge to protecting the memory space. While processor-based systems often have some separation mechanisms for protecting processes in sharing the memory space, current systems

built based on reconfigurable hardware have a flat memory addressing scheme. Since soft cores are synthesized from a wide source of IP providers, vulnerabilities or malicious cores can issue unauthorized memory requests, corrupting or revealing sensitive data. [34] shows that a compiler can translate an access policy directly into a synthesized circuit that can act as an execution monitor and enforce the required memory access policy. The authors provide a few example policies such as compartmentalization, access control, secure handoff, Chinese wall and redaction. The specialized compiler takes the access policy expressed as a formalized language through a series of transformations into a state machine then into a hardware description language, such as Verilog, that represents the hardware reference monitor. Either distributed or centralized, an arbiter validates every single access to memory insuring their validity and allowing the memory request to pass between modules separated by a *moat*, thus creating a *bridge* for access [65]. While this solution is well suited for separating untrusted soft cores, it does not allow a more granular separation for multiple principals that share the same hardware. For example, when one or multiple processor cores share the same reconfigurable fabric, the software adds a new layer of entities that share the access rights of the processor core. Thus the hardware-only protection is inefficient as a system-wide security mechanism.

[121] describes a memory protection mechanism targeted against remote attacks that target data corruption. Since most buffer overflow protection techniques only deal with deviation of the program control flow, this work tries to fill the gap for attacks that target any other non-critical data in memory (such as the return address on the stack) and are not intended to divert the control flow. As such, the access control for a memory region is limited to either writable or read-only and the compiler is responsible to insert the code instrumentation to correctly label data memory. Since memory is only

modified by store instructions, checking and permission setting is performed before a store. Thus for a full and consistent protection, the entire memory space is initially read-only and only before each store instruction the bounded memory range is set writable. This prevents injected or incorrect code from accessing unauthorized locations in memory. The compiler analysis is responsible for detection of protected memory ranges and optimizations designed to limit the performance overhead of the inserted instrumentation code. A few important limitations of this design are worth mentioning. First, the protection mechanism does not prevent any attack that is able to insert memory access instructions. That alone eliminates using the protection scheme in scenarios with unstrusted code. Another limitation is that no control flow attacks can be prevented. Lastly, the limitation on only two permission rights is not sufficient for a comprehensive protection mechanism with multiple untrusted entities sharing the memory space.

2.3 Capability systems

Capability systems [24, 15, 64] rely on authorization tokens to grant access to resources. A principal cannot access data unless it has the key to unlock the protected resource. A capability is such a key that needs to be passed from principal to principal (protection domain to other protection domains) without allowing anyone to forge them. As the capabilities are created and passed around, it becomes very difficult to manage them. As such, a mix of hardware [63] and OS support is required for book-keeping, verification, searching and revocation [80].

Capability systems bear some similarities with our access control mechanism. Both are designed to allow strictly controlled access and sharing to information. The main difference is how the management of permissions is

performed. Capability-based systems require that every entity should manage its own capabilities. In a sense a capability is just another piece of information that one protection domain owns and can perform a (limited) set of operations on. Copying, transferring, subsetting, deletion are the typical operations performed on capabilities. Since not all operations are allowed (e.g. extending a capability), it is common for implementations to keep capabilities in a separate domain from regular application data. This way the system can enforce restriction on the type of operations performed. If capabilities can be transferred back and forth between domains, any regular data operation can be used to manipulate capabilities as well. A typical scenario for a capability-based system follows. An entity (program) is given a set of capabilities, at least one for its own private space, and potentially others for sharing resources. The entity is responsible for loading a capability in special hardware registers before accessing resources. A monitor checks the access against this set of registers. Grating access to the entity's resources is possible by creating copies (or subsetting) the owned capabilities and transferring them to other protection domains. Super-setting is typically disallowed, otherwise one could elevate access to any resource. Without clear management of capabilities, the two most problematic issues of capability-based systems are: permission revocation (once granted, a capability can not be taken back) and delegation control (once granted, a capability can be passed to other entities).

2.4 Virtualization

Virtualization (in a software security context) is an isolation technique that abstracts the hardware from the operating system. It allows independent instances of a hosted software (in most cases an operating system) to run in complete isolation from other instances. There are both software and hardware

virtualization techniques. Software virtualization, also referred to as emulators or simulators, provide an abstraction of the physical machine to the hosted OS, and implement in software the necessary translation mechanism and security verifications. Hardware solutions implement the resource separation by either duplicating some of the hardware resources or by relying on reference monitors to mitigate the resolution of resource accesses.

Many research directions have as basis virtualization techniques. Many use the hypervisor as run-time monitors for the targeted host systems, in order to detect security vulnerabilities or detect attacks [83], [111], [25], [88],[75] or reverse engineer attacker behavior in infected systems [76],[33]. Others show that the virtualization techniques [77] are still insufficient against some class of attacks.

The following two techniques : NaCl and Arm TrustZone are detailed as representative samples for typical uses of virtualization for security applications.

The Native Client (NaCl) [110] is a sandboxing mechanism designed for safeguarding untrusted execution of native compiled binaries. The framework specifically targets the web environment, in which browsers need near native execution performance of applications, and where the security model of plugins and active-x components is not sufficient. The security model relies on a trusted runtime engine, a static analyzer of binary code and fundamental properties of the x86 processor architectures (segments, paging and process isolation). The analyzer rejects any binary that does not conform to a set of strict rules, designed to prevent any unsafe code from starting execution. The restrictions are: the binary is statically linked with a precise memory layout and the text segment will not be written beyond program loading; all indirect control transfers are implemented using a pseudo-instruction called "nacljmp" (all jump and call instructions are prohibited); the text section is padded

with "halt" instruction(s); and all instructions are 32-byte aligned. The binary disassembly is aided by enforcing all instructions to be reachable through fall-through sequence and the direct control transfers will always target valid instructions. The binary is also scanned for unsafe op-codes, and no direct OS calls are allowed. The runtime provides some API for typical operations, and uses a trampoline call gate and springboard return gate for calls between the trusted and untrusted code. With these limitations, the framework allows for a set of communications channels, mostly from the browser and the javascript engine, but also between modules. Threading is supported using a POSIX-style model. The performance overhead is reasonable, topping at 12% with an average of 5%, and major causes for degradation come from the 32-bit alignment requirement and from the restricted control flow mechanism.

One solution specifically designed for embedded devices is Arm TrustZone [113]. The TrustZone CPU allows the software to view two separate virtual processors, without duplicating the physical core, by only relying on hardware extensions within the CPU with less than 5% space overhead. Two operating worlds are supported: one secured and one normal. The secure state has more privilege, a OS in this mode being able to access all hardware resources, while the normal world can only have direct access to a fraction of the peripherals, though it may access the rest, mediated through the secure mode. The memory space is split as well, with a portion of the space inaccessible in the normal mode of operation. The security state of memory accesses is explicitly transmitted for all the system through the bus interface (and extended protocol ARM AXI, that uses 2 extra signals to encode the security state for read and write accesses). Since there is only one CPU shared by both worlds, a separate mechanism, called a Hypervisor, runs in a state called monitor mode. This mode needs to have the most robust and trusted code as it is responsible to transfer control and change between the secure and non-secure modes. The

monitor code needs to securely store and restore the register state belonging to each world during transition and for restarting execution in the appropriate mode. Note that the context state is stored in the memory space of the secure world, and has no reserved secured memory. As an application for the TrustZone platform, TrustZone Software services are implemented as part of the Secure Operating System. The normal OS and applications can interact and call services exported through an API. The normal OS needs to implement a Secure Calls Driver to interface with the secured applications. Among the services offered are secure storage and cryptographic services.

The separation mechanism of ARM TrustZone offers good isolation between the execution modes, while having only small resource utilization overhead (in terms of silicon space used). The granularity of the protection is coarse. The memory space is divided into only two zones, with a hierarchical access policy. The secure mode being able to fully control the normal mode and vulnerabilities in each of the domain can still affect a very large memory space (any exploitable code in the secure OS will render the full system inoperable).

2.5 Information Flow Tracking

This field of work concentrates on tracking the "quality" of data as it originates from various sources. Based on the source of data that a program processes (IO, memory, registers, etc), trust levels are associated with these sources. To simplify, data comes from either "trusted" or "untrusted" sources. Depending on program semantics the information channels are tracked in order to determine if untrusted data tainted the trusted information, or the reverse, if trusted – thus secure – data leaks into untrusted information that can divulge secrets to the unsecure channels. The problem is applicable in secure

computation environments, where it is important to identify all possible communication channels either explicit (direct variable assignments) or implicit (control flow, timing, cache content [67], power etc). Traditional approaches to track information flow through static and dynamic tracking ([21],[95], [74]) have the great disadvantage that even very few untrusted sources can *spoil* or *taint* trusted data very fast, making almost all computations impractical. Such systems do not enforce memory access, allowing code from untrusted compilers to bypass the security policies. Even with good compiler instrumentation, the escalation of data tagging from trusted to spurious is fast and makes such systems as impractical as the theoretical multilevel security policies [9].

Fundamental restrictions on the traditional architecture, like in [102], achieve full control over the information leakage although with serious limitations on the computation model and high resource utilization. The proposed architecture shadows traditional gates with trust tracking logic. AND, OR and MUX are augmented to compute the trust level of the output. Any higher level logic is built using these basic gates. A major architecture change deals with Program Counter operations. Since jumps and conditional branches with untrusted operations automatically mark the PC as untrusted, all conditional logic is implemented using predication (execute an operation conditionally on the value of predicated registers). Loops are severely limited to statically bounded iterations. A special instruction *countjump* allows non-linear execution. Memory operations allow only direct and loop-relative accesses, as indirect accesses are completely forbidden (because an untrusted index forces the full memory state to be marked untrusted or tainted—an undesirable effect). The resulting processor is a slower, larger and severely limited core, but whose operations can guarantee full tracking of information flow.

A flexible architecture for taint propagation is described by [105]. This work's strength is the ability to allow flexible (limited programmable) propa-

gation policies. The taint information is associated with each memory word, but not by widening the width of memory cells, but by keeping a compacted flat data structure (array of bits) in a separate memory segment, and allowing the possibility of using separate or merged caches. The hardware support includes new pipeline stages (to the end of the pipeline) that perform the taint propagation. The commit stage forwards the operation (opcode and operands) to the FlexiTaint engine that deals with the prefetching, computing and storing the taint data (in specialized TaintRegisters and sending to the TaintCache). The taint rules are high-level defined (software) based on the operation and operands and two specialized instructions, taintr and taintm, allow for high level operations that can not be automatically detected at the ISA level. Because executing the taint algorithm may have a big performance impact, common case operations are cached using a custom designed unit.

2.6 Stack and Heap Protection

A wide range of software-only approaches exist for buffer overflow prevention and detection that can be divided into static analysis tools and code instrumentation techniques. Some solutions in the first category are FlawFinder [81], BOON [109], ITS4 [107], MOPS [108], StackOFFence [58], and PFD [22]. These approaches use source code analysis to detect potential vulnerabilities in the code before execution. The two main advantages of source code analysis are zero performance overhead at runtime and offline vulnerability detection. However, static methods do not protect from runtime attacks, and are prone to producing false positives. In the second category of software solutions, the best known are StackGuard [19] and ProPolice [26]. These approaches rely on the inability of an attacker to guess the value of a randomly generated "canary" value placed just before the return address on the stack. Unfortunately,

more complex overflow attacks and format string attacks can reveal the value of the canary or change the return address without altering the canary.

Two code instrumentation techniques do support runtime detection of attacks and are particularly relevant because they both use a shadow stack protection mechanism similar to our RA stack. The first approach, the Return Address Defender (RAD) [14], performs source modification of each function's prologue and epilogue to include special instructions to store a copy of the return address in a particular area of the data segment. Protection of the shadow stack memory region is handled by the operating system. The second approach uses dynamic instruction stream editing (DISE) to perform the same modifications only using binary instrumentation [16]. The advantage of a scheme based on binary instrumentation is that source code does not need to be exposed to the protection mechanism. However, in order to make the RA stack read-only these schemes rely on OS primitives to manage page-table permissions which incurs high performance overhead. In some environments the software-only solution suffices, however, in the embedded world, the added overhead is unacceptable. Even worse, the OS and memory protection mechanisms might not even be available.

Hardware-assisted approaches to buffer overflow protection improve upon accuracy and performance of software-only schemes for dynamic attack detection by using a variety of techniques. The most closely related to our solution include approaches that shadow the return address in hardware by creating a return address stack or monitoring the location of the return address for any unauthorized modifications [45, 68, 117, 37, 40, 82]. We will briefly review these closely related works. Other hardware-supported solutions tend to focus on protecting control-data in the general sense, including all branches and jumps, not just the return address [20, 89, 94, 120, 12].

Secure Return Address Stack (SRAS) [45] uses processor modifications,

which includes ISA (Instruction Set Architecture) changes, additional logic, and storage space, to implement a shadow stack in hardware. Unlike the regular system stack, the shadow stack only holds return addresses. On a function call (CALL), the return address is pushed to the regular stack and the shadow stack. On a return (RET), the return address is popped from both stacks and compared. To handle function call nesting, the operating system is modified to handle secure overflow storage. The spill-over of the secure stack is stored in a special part of memory, which is accessible only to the kernel. The kernel is responsible for managing a separate spill-over secure stack for each process.

Several solutions for non-LIFO control flow are proposed for SRAS. Because `goto`s are generally considered a bad programming practice, they are not allowed. The `setjmp()/longjmp()` situation is handled in four ways. In the first case, non-LIFO flow is prohibited altogether. Second, if non-LIFO is detected then SRAS is disabled, allowing execution to continue without protection. Third, `setjmp()/longjmp()` executes, but only if the return is already somewhere on the secure stack; this requires the compiler to insert `sras_pus()` and `sras_pop()` functions. The fourth solution identifies the non-LIFO control flow situations dynamically and has separate handling and dynamic stack adjustments; this last approach works fine as long as all non-LIFO cases are known in advance (a new non-LIFO methodology would not be detected on the fly).

SmashGuard [68], like SRAS, uses a processor and operating system modification technique. The semantics of CALL and RET instructions are modified to store a function return address inside a special memory-mapped hardware stack and compared upon the function's return. All context switching and spill-over stack growth is handled by modifying the operating system to include additional security kernel functions.

The non-LIFO (`setjmp()/longjmp()`) situations are handled in SmashGuard through library rewriting, so that `setjmp()/longjmp()` is handled by a "jump-through-register" rather than regular jump. This creates stack inconsistencies upon return address verification, which are handled by popping addresses off the stack until reaching the correct address. If the end of the stack is reached, then an error or attack has occurred (i.e., the return address verification has failed), so the execution is halted. The drawback of this approach is that an attack, which forces a jump on an executable stack would succeed. Also, if a function is called multiple times in a loop, in order to pop the correct version of the function's return address off the stack–not just the first occurrence–SmashGuard also stores the corresponding value of the stack pointer.

Xu et al. [117] present a scheme that splits the stack into two pieces: the *control* stack, which holds the return addresses, and the *data* stack, which holds everything else. They propose a software solution, which involves compiler modification, and a hardware solution, which modifies the processor and semantics of CALL and RET. In the software-only solution, the compiler allocates and manages the additional control stack. During each function prologue, the compiler saves the return address to the control stack, which resides in memory securely managed by the operating system. The compiler restores the return address from the control stack onto the system stack in the function epilogue. Simulation yielded significant performance overheads, which led the researchers to a hardware solution.

The hardware redesign is very similar to SmashGuard and SRAS, although there are additional operating system changes to handle the two stacks. The processor is modified to change CALL and RET instructions to store and verify the function return addresses. However, even though an extra `jump_buf` structure is mentioned for `setjmp()/longjmp()` cases, the non-LIFO problem

is currently not addressed, and neither is stack spill-over. Thus the authors claim no performance penalty for the hardware solution.

Secure Cache (SCache) [37] uses cache memory to protect the return address, by providing replica cache lines in which the return address is shadowed. This way, a corrupted return address will have a different value in cache than its replica. By preventing the eviction of replica cache lines, the detection technique improves. In addition, the non-lifo cases do not cause a problem for SCache, because the return address is not explicitly checked against a known value. A drawback to using cache space for storing the return address is that the SCache approach is sensitive to cache parameters and behavior.

Kao and Wu [40] present a scheme to detect if the return address is modified without explicitly storing the good return address for verification. Two registers are added to store the current return address and the previous frame pointer. A valid bit that acts as an exception flag is also added. If a memory store is issued to either a return address on the stack or to the frame pointer, then the flag is raised to signal a violation. When the function returns, if the return address is to the local stack or to the current return address, then execution is halted. The authors claim that only two levels of function calls need monitoring. As with SCache, the non-lifo situations are not a problem in this scheme, because the return address is not explicitly checked; however, if non-lifo control flow is not detected, then a `longjmp` to a function that is exploited by a buffer overflow might remain undetected.

The existing hardware solutions for runtime return address protection naturally yield much smaller performance penalties than software-only approaches. To accomplish such improvement, processor modifications are made to include a hardware stack (or support for software shadow stack) and some additional processing logic.

Just like the traditional buffer overflow [17] attack targets the return ad-

dress stored on the stack or function pointers [18] to launch attacks, heap metadata (the run-time bookkeeping managed by dynamic memory routines like malloc) has also been a target of overflows that manage to divert the control flow or arbitrarily change values in memory [27],[41]. The vulnerability is caused by the way a free operation consolidates the empty heap chunks. If the pointers originally set by malloc get overwritten by an attack, the free routine can be tricked into writing at an arbitrarily set memory location, with a value set by the attack vector. The solutions proposed are limited to a) solving this unique threat and are based on either changing the original metadata structure so it is less vulnerable to attacks, b) placing read-only guards before and after the heap metadata [84],[106] or c) moving the metadata to a separate memory region than the allocated data as in [41]. Although the heap protection techniques help reduce the likelihood of heap attacks, they do not offer a complete solution, leaving room for more powerful attacks and they are not intended to solve memory access control.

2.7 Control flow

A major part of software security research focuses on statically determining the control flow graph of a program and then enforcing compliance at run-time. Since corrupted data can lead to deviations from the programmer intended behavior (through buffer overflow or corruption of function pointers), run-time monitors compare the statically extracted control flow graph with the current execution pattern. Either through software-only instrumentation or using architectural support, the basic idea is to derive a static characteristic of the program, create an encoding for it and design the reference monitor that ensures the run-time behavior follows the static representation. Program execution flow is typically characterized through a directed graph, with nodes

being typically code that executes in a straight line (e.g. a basic block) and vertices representing control flow operations (jumps, branches and calls). DFI [13] and CFI [3] use inline guards to enforce a conservative permitted behavior. Others, using hardware support, encode valid entry and exit paths into basic blocks using either unique ids, or bit maps and relying on hardware support for verification [119], [30]. Another higher level of control flow checks is to make sure that function calls (API calls between software components) are performed correctly. Validation of function level flows relies on the assumption that inner function control flow is correct (or validated through other techniques) and only ensures that the call graph is correctly enforced. In this category are many hooking techniques to create wrapper code for existing legacy code, either for security ([116]), reliability (Nooks [97], Shadow Drivers[96]) or other functionality enhancements (like debugging: [112], [42]).

2.8 Architectural Support for Debugging

Although not intended for security applications, but for debugging and monitoring memory accesses, MemTracker[106] offers the architectural solutions for configurable state checking of memory. This solution stores a compacted state array for every memory word, and uses a programmable state transition table to define the exception behavior and the transition rules. Checks and changes to the state are done as an added stage to the traditional pipeline, preserving in-order execution of load and store operations. This approach can be used for some security applications, coding the state of memory words with security attributes. Unfortunately, this mechanism is designed only for infrequent and coarse protection domain changes as each protection domain needs a full map of the memory space. The paper has a detailed and thorough description of pipeline integration techniques and strategies for minimizing execution delays,

techniques that we adapt for principal detection and memory access control.

Designed as well for easing debugging, iWatcher [122] uses architectural enhancements to allow for memory monitoring. A programmer can specify several memory ranges to monitor and type of operation (read or write), and an action to be taken in case the memory is accessed. At the architecture level, iWatcher uses cache line tagging (2bits/word) for small monitored memory regions and a Range Watch Table (RWT), for large monitored memory regions. After detection the triggered action is multiplexed by a single entry function, whose pointer is kept in a special register. In order to allow only committed instructions to trigger debugging actions, the reorder buffer (ROB) logic is modified and each entry is enhanced with trigger bit and each load/store queue with 2 additional bits. iWatcher also describes several techniques that use thread level speculation to accelerate the execution of the triggered action. The detection and reaction state is much smaller in iWatcher than in our security application. Without needing to associate several principals with the memory accesses, only Execute, Read, Write accesses need to be verified.

As an application of iWatcher, it is worth mentioning: fine grained memory access control used for implementing Transactional Memory(TM) [8]. There are 2 flavours of TM: Software Transactional Memory(STM) and Hardware Transactional Memory(HTM). As expected, STM has the advantage of being able to handle complex scenarios and flexible conflict management policies, while being slow for some cases. HTM is fast, yet can either handle restricted simple policies (bounded TM) or gen be overly complex for handling any complexity (unbounded TM). A mixed implementation (HTM and STM) has the advantage of mitigating the shortcomings of both, yet it is difficult to achieve for one main reason. If conflicts occur between a HTM and a STM the HTM should be aware of the STM resolution policies. This defeats the principle that the HTM should only offer supporting primitives for TM and not a fixed im-

plementation. With the aid of iWatcher's memory debugging features, BTM (best-effort hardware TM) and USTM (User Fault-On Software TM) can co-exist. Accesses by the HTM are prevented from violating STM transactions atomicity by using a mechanism that marks cache line with a fault-on-read and a fault-on-write bit. Faults that result by violating accesses to the cache line result in exceptions that are used for conflict resolution by handlers in the STM.

2.9 Physical Security

Encrypted Execution and Data (EED) platforms focus mostly on preventing highly skilled attackers from physically probing the system busses and memory modules. Among these are [53] and the work of [124] and [123]. In the latter, the authors study the problem of information leakage when an attacker extracts patterns of access in an EED platform and matches those patterns against a database of well-known patterns extracted from open-source software or from unencrypted executables run inside a debugger. Their findings suggest that many algorithms can be identified by observing their memory access pattern and that this signature pattern can itself lead to both information leakage as well as additional types of attacks. They propose address randomization to foil such attacks and study the performance of a specific architecture for address randomization. The AEGIS architecture [93] greatly improves on its predecessor (XOM [56]) and presents techniques for control-flow protection, privacy, and prevention of data tampering. A common theme in past work on EED platforms is the requirement of the design of new processors. These proposed architectural changes, to the processor and instruction set, incur a high cost in terms of design and require a buy-in from chip manufacturers.

Physical probing of devices is also investigated by a strong community that

focuses its research on Differential Power Analysis (DFA) [43]. Although the attacks are targeting breaking encryption keys of common encryption algorithms [91], [66], any other control flow or algorithm design can be theoretically revealed by this approach. EED platforms are still vulnerable to power analysis and benefit from countermeasures such as the DFA resistent implementations of DES and AES [4] or secure IC design [101]. Specific techniques to protect FPGA circuits have been designed by [118].

2.10 Recovery

Recovery-Oriented Computing [70] is a model introduced for highly dependable computing platforms, especially for distributed Internet services. Designed to provide a solution to a wide range of failure scenarios, the concept underlines the importance of providing a recovery mechanism from crashes rather than simply adding redundancy for fault tolerance and eliminating all possible points of failure. The same principles are equally important in embedded systems where, because of the highly distributed nature and deployment scenario, skilled operator assistance and support is difficult to provide for failing devices. [99] presents a design for processor and memory hierarchy to support speculative execution of faulty software and the ability to rollback and re-execute. Speculation in general is implemented with the aid of a set of buffers that hold the state of registers and memory. While the execution is in speculative mode, the memory content is only buffered and only after the commit decision is made the data is saved, otherwise it is discarded. The contribution of this work is to allow very long sequences of speculative execution using FPGA backed architectural support. The purpose of the system is on-the-fly debugging of production code. Changes in the ISA and compiler support allow easy transition between the three execution states: normal, speculative and re-execute.

The transition between states and the speculation decision is achieved throw Observation Points, inserted by the compiler. Although only the hardware support is described in detail, the role of the software is underlined as being essential—but deferred to be a software engineering problem. The hardware modifications include cache controller changes to allow storage of speculated data and register checkpointing. While in speculative mode, no speculative dirty cache line is retired to memory, only cache line that hold no speculative data are allowed to be written. Commit or invalidate require a walk through all cache lines marked speculative and either discard or marked them valid. Since register checkpointing is costly in software (explicit instructions to save register content to some memory location), a Register Checkpointing State Machine is responsible for copying the register content to a rollback buffer and restoring the appropriate state once speculation is done.

This solution has the benefits of utilizing only a small amount of hardware resources, mainly because it uses the available cache as the memory checkpointing buffer. Although the approach may trash cache space in frequent mis-speculated executions, the small space overhead outweighs the disadvantages.

Chapter 3 – Problem model

The architectural and compiler enhancements that we propose are directed to thwart attacks that make unauthorized accesses to memory. These attacks are often a critical first step in carrying sophisticated high-level attacks, causing intellectual property loss, denial of service attacks, malware deployment, critical data corruption. This chapter will describe a typical software development model and how vulnerabilities can affect each stage of program creation.

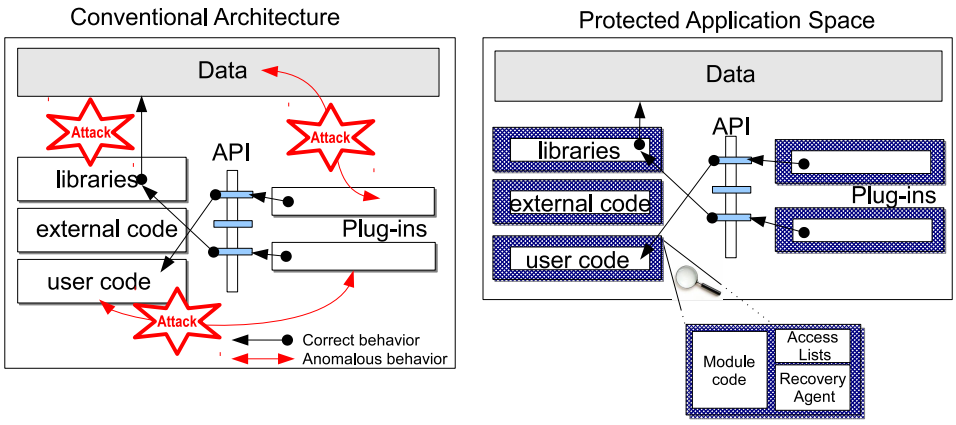

Figure 3-1: Attack vectors and containment

Modular software design can greatly increase the productivity of a program engineering team. In the general sense, any collection of data and code that defines a set of operations related to a particular task can be named a software module or component. Reusability is the key motivation for this kind of grouping. If a component can be written once then reused across multiple applications, the development process is accelerated, maintenance eased as bug fixes can be centralized, etc. A component can be defined at any level: macro definitions in assembly code, a collection of procedural language methods compiled in a library, an object oriented class or namespace, Component Object Model (COM) objects and their recent derivatives (ActiveX, COM+), or even multi-host services such as Web Services, CORBA, etc. The modu-

larization also created a new market niche for intellectual property providers. They specialize in component development, having the entire business model based on providing the best designs and implementations in their expertise area. Why spend years researching and implementing specialized algorithms in house, when you can purchase software components from specialized vendors? For example, a company that builds handheld music players can easily purchase the encoding/decoding software the same way it would buy the memory modules from a dedicated manufacturer. This modular assembly process usually assumes a black-box view of the imported modules. This is most of the time a constraint motivated by the protection of intellectual properties from the vendor's side. The provider will only provide a compiled (sometimes obfuscated) library for the targeted technology and keep the source code as a trading secret. Following sections illustrate critical hot-spots for introducing software related security problems (Figure 3-1).

3.1 Source code

There are other factors that motivate the black-box approach. Even if the entire source code is available for a component (ex. from an open-source project), the complexity of analyzing a large code base - often without documentation - is disconcerting at best. For a complex software product the code base is created from a multitude of sources. A short analysis of the source code for the open source Firefox browser [11] revealed some unlikely code authors. Firefox is, as expected, a collection of code aggregated under a common project management from many open source libraries. A simple analysis of the source code (based on the Copyright information in the files) gave away a few examples:

- **jpeg** image manipulation code comes from : Independent JPEG Group

- **network** primitives from : Japan Network Information Center
- **parsing** primitives from : Thai Open Source Software Center Ltd.
- **garbage collection** Bohem (conservative garbage collector for C and C++): Xerox Corporation
- **cairo** - a vector graphics library with display and print output : University of Southern California
- **png** library code developed by an International group of programmers (Glenn Randers-Pehrson, Andreas Dilger, Guy Eric Schalnat Group 42 Inc.)
- the list can go on...

The above mentioned code comes as it is, other code (not mentioned here) comes from existing sources, but was adapted to the application specific needs. After compilation, the resulting binaries come as an integrated part of the popular browser. They share the memory space of the application, having access to the whole process space. A simple example: if the jpeg library misbehaves or has a vulnerability it can reveal or alter any information from other parts of the application (keys, passwords etc that may reside in the memory at the time the vulnerability is triggered)

Static analysis tools together with testing procedures can help, but are known to leave out false negatives. Even for software that is created entirely by a trusted development team, there is no guarantee that the result is bug-free, invulnerable to all known and future attacks.

3.2 Plug-ins

Many complex software tools like web browsers, graphic design utilities, CAD suites, Web browsers, etc allow another type of extension that can be added to the base functionality of the program after deployment on the destination system. They are usually divided into plug-ins and add-ons. If we take the Firefox example, it allows both type of extensions(This excludes customizations that require changes to the source code). "Add-on"s rely on a interpreter for XUL (xml and javascript based language) and provides an API to graphical interface and DOM model. Although they are a considerable threat to the security of a Firefox user, the security aspects of Add-ons are not considered by the Container model (the interpreter should be able to asses and enforce the security mechanisms - similar to the java VM). The plug-ins need a higher level of integration and access because they typically require direct access to the display buffers of the browser. They are used to allow content types that are not natively supported by Firefox to be integrated in the application. The most popular examples of such plug-ins are:

- **Acrobat Reader** For viewing and printing Adobe Portable Document Format (PDF)
- **Adobe Flash Player** Adobe Flash Player is the universal rich client for delivering effective Adobe Flash experiences across desktops and devices.
- **Java** The Java Runtime Environment enables your computer to run applications and applets that use Java technology.
- **RealPlayer** RealPlayer enables your computer to play streaming RealVideo and RealAudio.
- **Windows Media Player** Windows Media Player lets you play streaming audio, video, animations, and multimedia presentations on the web.

Any vendor may create custom plug-ins for new content types. Unlike "Add-on"s plugins come as native binaries for a particular OS/hardware. Some of the above mentioned do not come with support for all Operating systems (ex. Windows Media Player does not have a Linux implementation). The source code for this applications is typically proprietary to the vendor and it is not available for neither the user or the Firefox development team.

3.3 Device Drivers

While application specific protection is important, specifically if it carries confidential, security critical information, vulnerable code or plug-ins will not manage to affect other processes or corrupt kernel space(OS protection mechanisms, file permissions, process isolation, should be sufficient to contain the affected application from doing extensive damages to the system). System drivers on the other hand are designed to run in the kernel space. A corrupted/malition driver would be able, without proper protection and memory isolation, to create extensive damage to the system, infiltrate and deploy root level malware or tracking applications. Because of their nature, device drivers are a ideal candidate for containerized protection. They are invoked frequently, are usually resilient in the system memory, and most require fast response time. Most of those requirements makes them unsuited for software based run-time checking. Their functionality can be strictly enforced with our proposed container mechanisms with minimal performance penalties, but with great security reassurance.

3.4 Buffer Overflow

Programs running on a computing system interact with the environment through a wide array of communications channels: keyboard inputs, persistent and volatile storage devices, networking devices, media scanners(audio, video, etc). Device drivers and operating system primitives abstract the low level protocol and implementation issues and typically present a unified view to the top level applications in the form of data streams. An application sees network, keyboard or file information as a sequence of words of various length that it can interpret based on some specific algorithm. This is where mistakes in the implementations can facilitate attacks. By creating streams of data that do not comply with the original algorithm (either in length or meaning) the attacker can "trick" the program into writing the values at arbitrary locations in memory. Depending on where the data is written and how the attack vector affects the vulnerable system, buffer overflow attacks can be categorized into [71]: stack smashing, arc injection, pointer subterfuge and heap smashing. The original buffer overflow, as well as the majority of vulnerabilities reported, rely directly on stack smashing attacks [44].

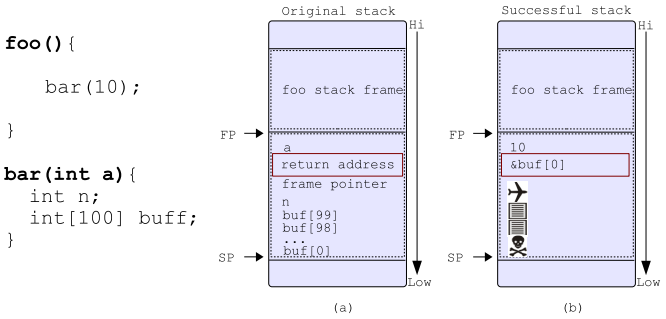

Figure 3-2: Stack smashing attack: (a) stack layout; (b) attack vector implanted

The attacker relies on the placement of the return address from a function at an address in memory in close proximity to local function variables.

For each function instance, typical C compilers allocate a memory region on the program stack. Call parameters, frame pointer of the caller and the return address are first copied on the stack. The next space is reserved for the local variables. While operations with fixed length data structures (basic types, structures) are less prone to cause vulnerabilities, statically allocated buffers on the stack are more susceptible to misuse. If the attacker succeeds in feeding a data stream longer than the allocated length, the program will overwrite memory locations adjacent to the buffer. If the stream of data is sufficiently long, one of the location overwritten is the location of the return address. Since the return address value will be used for an implicit jump at the function return, the attacker now has a possibility for making arbitrary control flow redirections. This ability can be exploited directly, by immediately returning into the recently overwritten buffer, or through other set of indirections. Figure 3-2(a) shows the stack layout during the execution of a function with vulnerabilities. Overriding the buffer buf with executable code, and exceeding the original length(100) allocated for the buffer, the attacker is able to divert the control flow such that the program starts executing the injected code (Fig.3-2(b)).

One common attack mechanism to overcome a non-execute protection of the stack, called return-to-libc, is to force the function return into glibc library functions (such as exec(), or system()) with properly engineered parameters (ex. "/bin/sh"). The most powerful buffer-overrun outcomes is to allow the attacker to spawn a shell with the same privilege as the targeted application. Depending on the local system security policies, a successful attack can potentially give the attacker the ability to run any commands on the vulnerable system with root privilege. Runtime protections against buffer overflow attacks try to protect the location of the return address by either making it read-only, moving it to a safe location or challenging the attacker to solve a

hard problem.

3.5 Conclusion

Component based software modules are typically characterized by a high level behavioral description, in the form of an API (Application Programming Interface) and functional specifications. Equipped with these, the user can test for conformance and compatibility according but usually can not explore the full state space of the imported modules nor can he test for all bugs. Past the comprehensive tests, only a matter of trust in the provider motivate the decision whether to incorporate the component into the final product. Once the module makes its way into the final product, there are several possibilities of failure : back doors intentionally inserted by the provider, software bugs that were difficult to catch during testing phase, undetected vulnerabilities that allow attackers to trigger more sophisticated attacks (buffer overflows described in Section 3.4). The outcome of a successful attack can include for example: code tampering, code injection, intellectual property loss, confidential data theft. It is this scenario that motivates us to view each component as a potential malicious entity. We keep our black-box approach when designing the security requirements for each software component, presuming each might have a vulnerability that can be exploited into inflicting maximum damage to the target system. That is why we do not categorize the threat level of each component and always assume the worst. Our preventive measures divide any code base and imported components into the smallest functional units and create strict bounds for each. Detection of misbehavior is maximized and strictly enforced this way.

Chapter 4 – Solution Details

A primary goal of this approach is to make both the hardware and software enhancements as transparent as possible. In the case of hardware, we provide the bulk of the needed hardware functionality in a module called the Container Manager that sits between the CPU and cache as shown in Figure 4-2. This requirement is needed for security context detection, as caches would obfuscate the relation between the executing principal and the memory accesses, allowing for some classes of attacks. Similarly, our goal in software is to have most of the work done by the compiler and loader, with minimal work required for the programmer (Figure 4-1).

We begin by describing what proper execution means and what is the information we need at run-time in order to detect incorrect behavior. The second part are the architectural enhancements we create for a typical processor architecture needed to implement the reference monitor.

4.1 Permission tables

Most modern programming languages use functions (or subroutines) as structural elements for creating applications. This code separation is motivated by several factors: increased program readability, separation of independent functionality, elimination of duplicate code, and code reuse. Linking with external libraries or imported code is typically done using functions as entry points. Our architecture makes the most out of these characteristics of functions. We assume that the control flow follows an activation tree structure, meaning that the execution of the program consists of sequentially executing program steps (functions), and that the execution of a function starts at the beginning of its body (the entry point) and returns control to the address immediately after

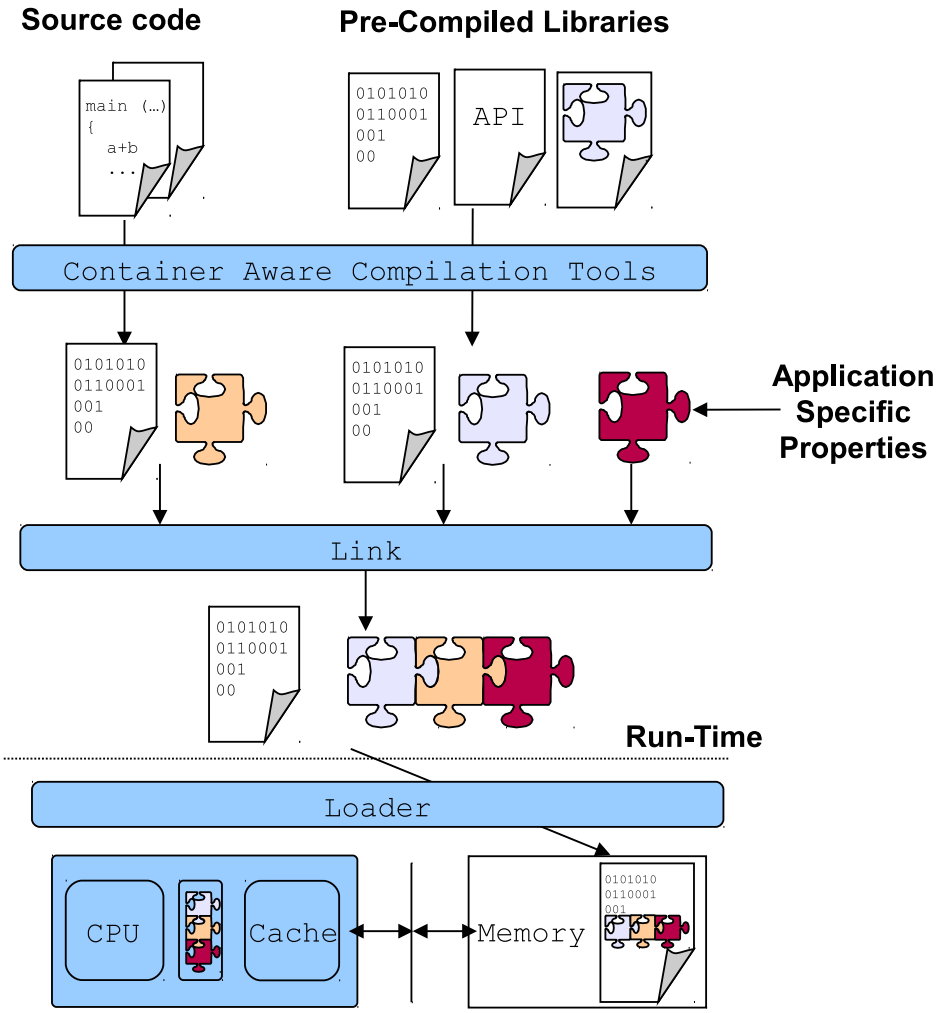

Figure 4-1: Overview

the point that it was invoked [104].

We define a ***Container*** as a collection of executable code (typically one or more functions) together with an enforceable list of permissions and metadata that restricts the code's access to memory and processor time. The reference monitor manages all container contexts, permissions and state and is thus named the Container Manager (CM). Fig 4-2 shows the CM's internal components: the Container Identification Table which keeps the definitions of all containers in a program; the permission list for the currently running principal stored in the Container Runtime Record (CRR); and a buffer for storing dynamic permission in between container switches, the Dynamic Permission

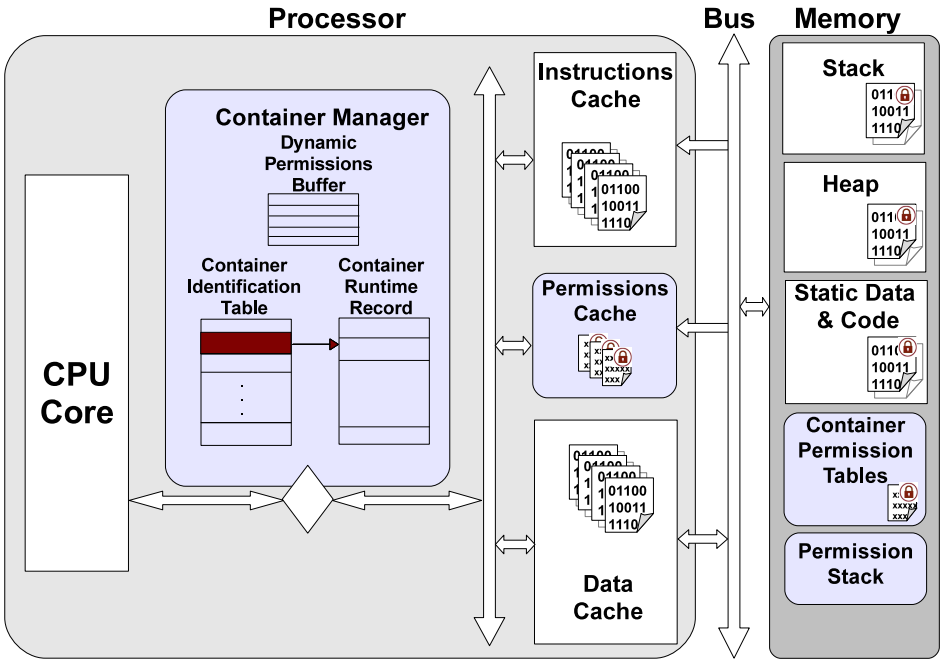

Figure 4-2: Architecture Overview

Buffer. The CRR contains: the list of all permissible memory ranges from within a container, the container code range, the full list of other functions or jumps outside its own bounds, and timing constraints.

The Container Manager checks each memory access (instruction fetches and memory loads and stores), identifies containers, and validates accesses. Since we do not rely on a naming scheme for container identification, the effective PC address (either virtual or physical) is used to identify a container. Even though relocating and library code may have multiple virtual addresses for the same physical address, we assume the code cannot use supervisor services to relocate itself without strong checks from the supervisor. A secure kernel can manage multiple sets of permissions that are context and location dependent and restructure the permission lists according to the context. Another assumption is that the security policy expressed by the access control lists prevents containers from overwriting any other container's code and then directly or indirectly switching execution to that code. (Although we assume

the container can have self-modifying behavior, a write access to another container's code region allows containers to migrate across the static boundaries and assume the permissions of the modified container.)

Next we will consider the types of execution events that need to be handled in the monitoring system:

Code entry When a jump occurs into a container from outside, the reference monitor checks that the jump is permitted and then loads the CRR for the destination container. An instruction counter is also initialized to begin counting the number of instructions that will be executed in the current context. The hardware also places the source container's ID on a special internal stack intended for tracking the chain of container-to-container calls.

Legal behavior inside a container Code executing inside a container is permitted to read and write memory in accordance with the permissions in the CRR. A branch or jump inside the container is permitted while the instruction-count is incremented to reflect within-container time.

Dynamic memory allocation In our approach, memory allocation (malloc) needs to be augmented so that dynamic memory is itself allocated in containers and that the permissions are set accordingly. Thus, a container that requests dynamically-allocated memory will be granted access for it by the memory allocator, while access to the allocation metadata is kept private to the allocator.

Return to the parent container Returns are validated by the Container Manager. The hardware checks the return address against the internally stored return address and loads the parent CRR. (Similar to any hardware stack, this enforces the expected call tree structure. Improper

returns or setjmp/longjmp violate this assumption. This behavior can be supported, but with extra storage and operations for the Container Manager—the software needs to instruct the CM that it will long-jump so that execution state can be kept in synch)

4.2 Container Manager

A summary of the important events/type of actions that occur in the Container Manager:

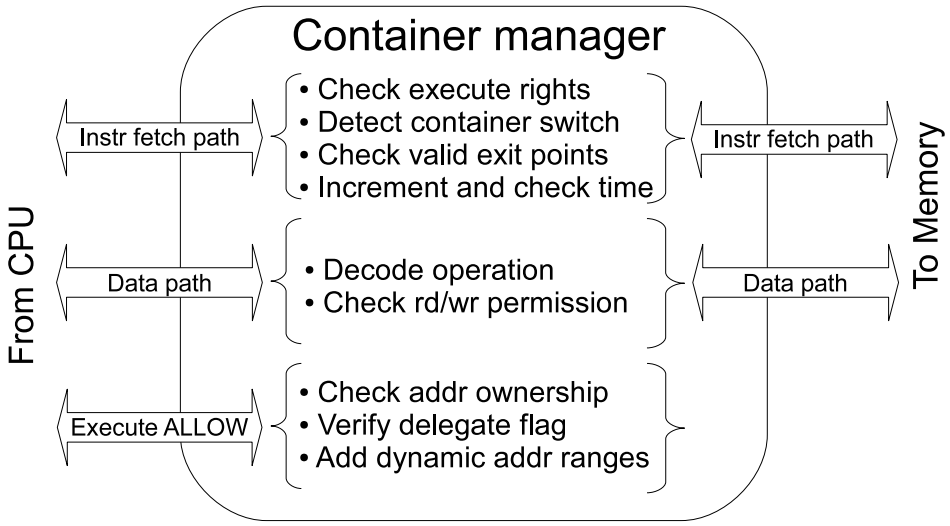

Figure 4-3: Container Manager Operations

- **Program Load** The loader initializes the system memory, loads the program code and data into memory. The container identification descriptor is loaded into the Container Identification Table. On context switches the OS is responsible for loading and saving this table.

- The program starts the execution. On every **instruction fetch** the Container manager first scans the CIT. If the fetch address is in the list, and the ContainerId of the identified record is different than the currently loaded container, then a security context switch needs to occur.

- A **security context switch** is allowed when the current container has the right to make a call into the new context (as listed in the permissible call list from the CRR) or the context is a return operation (the return address is checked to be valid). For a **return** operation, the Container Manager needs to load the static permission list of the caller, pop the permission stack for any dynamic permissions saved there and merge the content of the Dynamic Buffer into the CRR. On a **call** instruction the CRR dynamic entries for the current context are saved to the Permission Stack (they will be needed later on a return), the static permissions for the Callee are fetched from memory and merged with the Dynamic Buffer into the CRR.

- An **ALLOW** instruction is executed when programmatically the current container needs to delegate permissions to the callee or to the caller (ex. when a new heap buffer is allocated). It is the container manager that executes this operation. The first step is to check whether the current container has sufficient permissions and delegate rights on the memory range(s). It then adds all ranges to the Dynamic Buffer. From here the ranges are merged into the CRR on a security context switch.

- **Memory reads and writes** are always checked against the CRR. The CRR allows parallel search in all entries. If the address is matched in any of the ranges or the CRR, and the permissions match the requested memory operation the request to memory is allowed. Otherwise a security exception occurs, the request blocked, and the recovery procedure is triggered.

- **Permission Cache miss** Loading permission lists from memory may generate caches misses. While the permission cache is busy with a miss, the CM delays the retirement of the load/store/fetch until the permis-

sion can be checked. The memory access will still be sent to the proper cache, to allow misses to be worked on in parallel by the respective caches, but the memory operation is considered resolved only when after the permission check is valid. The processor pipeline is not halted and the instruction level parallelism can be still exploited. The delayed action acts like a slower response from the cache serving the request and having a dedicated cache memory helps alleviate conflicts with data and instruction fetches. The CM stays in WaitOnCache state (Fig. 4-4) until the permission cache miss is resolved.

- A **security violation** occurs in any of the cases: the container does not have permissions for the memory address it accesses, an instruction fetch from a memory location without execute permission, a return to a different location than the original caller, a call to a container not listed in the permissible calls list, an ALLOW operation that tries to escalate the permissions or a time-out exceeded for the current container execution. In any of the cases, the Container Manager records the details of the exception (type of exception, memory address and operation, source and destination container) and then invokes the recovery routine listed in the CRR.

- **Recovery**. The simplest strategy requires the container manager to treat the recovery procedure as any other entry point into a container. Once a security violation is detected, the control is passed to the recovery code (specified in the Container Runtime Record) together with an security exception data structure that is placed on the stack as a parameter(this requires the error functions to respect a fixed signature). The execution of the recovery routine is timed and permissions checked. The recovery code does not have security access outside its container (as

this may lead to exploits through the recovery code). The only difference from a regular container code is that a security fault in the recovery code itself does not restart the same recovery code (thus causing an infinite loop), instead it continues by invoking the recovery code that belongs to the caller container.

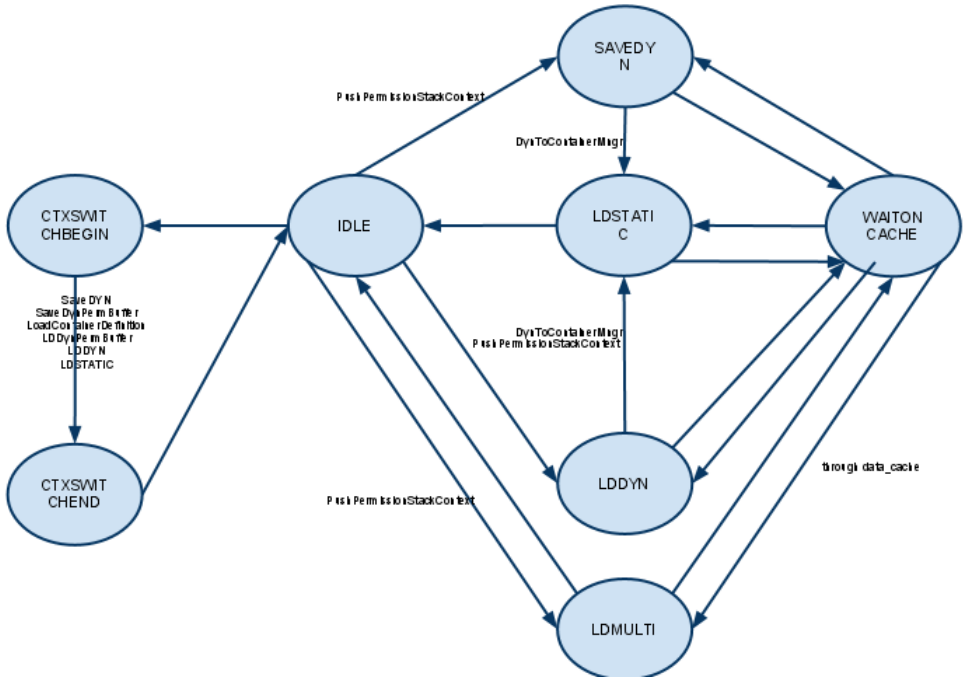

Figure 4-4: Container Manager State Diagram

4.3 Container Identification Table

The Container Identification Table characterizes each execution unit (a process or thread) and is a detailed breakdown of the containers that are part of the execution context. The key elements are the container entry points and the addresses where the permission records for the containers are stored. As Containers can have one or more functions, not necessarily in continuous locations, all entry points must be listed. A unique ContainerId is used for grouping the constituent functions, and for detecting weather the container context needs

to be switched. For each memory fetch the ContainerManager checks against all entry points in the Container Identification Table. Any memory fetch from one of the entry points different from the current loaded container will trigger a security context switch, operation executed by the Container Manager, without supervisor invocation. Fig. 4-5 shows how the CIT for a simple process consisting of three containers is defined. Note that a container may have private, inner functions that are not accessible by other containers. Those are not listed in the CIT, but need to have proper execute permission in the CRR. The compiler tool-chain, with assistance of programmed security annotations, can provide the permission table (run-time metadata and static access lists) according to the application security requirements.

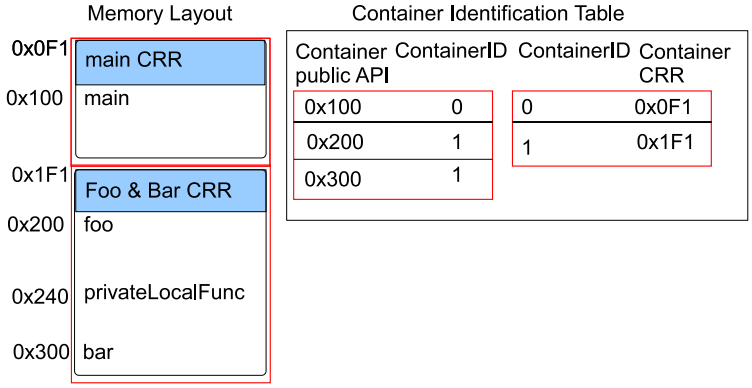

Figure 4-5: Container Identification Table

4.4 Static container permission record

Each container comes along with strictly defined rules on what are the permissible actions and memory regions it can access while it executes. These constraints are derived by the compiler and/or system architect. The compiler identifies all variable allocations, function calls and memory layout for the data structures used in a container. The programmer constraints can be added either as compilation directives in code or as linker configuration files.

Since in some cases the addresses are not known until link/load time, some of the permissions may be annotated by the compiler and later resolved by the linker/loader. The rest of the table has memory access rights in the form: Start Address, Size, Permissions. Internal executable code for the container needs execute permission, all static data permissions (Read, Write, Delegate) derived at compile time will be listed in this section. At run time the content of this table is loaded on a security context switch, and the content of the Dynamic Security Buffer is added to it. When a data memory read/write occurs, the static access list is already loaded into the Container Run-Time Record, and checking incurs a minimum penalty. Other metadata included in Static Permission Record: timing constraints for external container calls, and the location of the recovery code that is automatically invoked by the container manager when permission exceptions and/or time-outs occur.

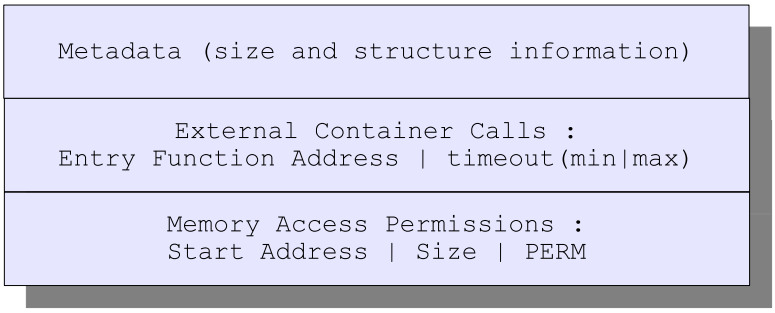

Figure 4-6: Container Runtime Record

4.5 Memory Permission Types

Based on the events above we can further detail the information in the CRR. The Container Runtime Record (CRR) encompasses three broad classes of permission information, each class corresponding to static, instance dependent, and dynamic memory references. Static security metadata is constructed by the compiler and used to build the static memory access lists for objects in

memory. Additionally, system designers (or programmers) can construct static metadata by using security annotations to help resolve compiler ambiguities such as function pointer tables, wild pointers, etc. Instance dependent metadata is used for resolving relative addressing schemes, however by some minor architectural changes we can often resolve instance dependent cases statically. Dynamic metadata is maintained by the container manager for handling memory allocation primitives (malloc, new), calling by reference and returning pointers from function calls.

Static metadata Static memory access lists are constructed by the compiler and include global variables, constants, static values, and instructions (code). Taking into consideration type-unsafe languages (C being the usual suspect), arbitrary memory accesses can be made by either misusing reference based addressing or intentionally including trapdoor functions with vulnerabilities. Most pointer references and simple pointer arithmetic can be characterized by the compiler into safe, forward-only sequential pointers [38]. The memory range for this type of code can be determined precisely, so the access lists constructed using this information could be included in the static access lists.

Unfortunately, complex pointer arithmetic and pointer assignments (wild pointers) limit the compiler. The system designer can help the compiler by using security annotations to specify the expected ranges of memory accessed when dereferencing these types of pointers, or the full list of expected code regions to be accessed in case of function pointers. We note that this is not a limitation for existing code, as program execution should not allow unintended access to arbitrary memory regions, and is instead a characteristic of vulnerable software rather than correct program design. These ranges can be defined through either compiler directives or external configuration files supplied to the linker, and would perhaps motivate work in development of new software tools to facilitate the additions.

Instance dependent metadata

Executing code calls a function by saving the current program counter (either in a register, in memory or a specialized hw unit [68]) and changing the program counter to the address of the called function. Recursive algorithms motivate a stack organization for the memory structure that functions use to store local variables, so that a function can safely call itself. Stacks are implemented by the compiler either using push/pop instructions provided by a CISC processor, or simulating a stack with only load/store operations in RISC and VLIW architectures. A function stack space requirement, as well as access to static memory locations, can be typically determined at compile time. Just like static variables, the location of these values are known at compile time. The only difference is that the location is not absolute, but relative a the frame pointer (FP). If the value of the FP becomes part of the container management hardware and protected from unauthorized changes, then FP relative access lists can be created by the compiler and added to the static access list. Even though access lists can be changed at run-time during execution, the overhead incurred by execution of extra code for permission changes can be reduced by having most of the work done statically at compile/link time. Since not all instance dependent metadata can be resolved statically, particularly if multiple functions share a single container or a single function spans multiple containers, the ambiguities can be solved using the directives for the dynamic memory, described next.

Dynamic metadata Although extensive effort can be spent extracting access lists at compile time, some situations will be impossible to statically resolve. Dynamic memory needs special handling of permissions. Giving access to a private heap for every security context in the program may lead to wasting a lot of memory, especially if the code is not memory intensive, or to increased penalties when a context uses up its allocated heap region. By allowing dy-

namic changes of the access lists, the same heap can be safely shared by multiple entities. Two requirements arise: first is that allocation primitives need to set permissions for the caller code, and second is that currently executing containers must be able to delegate permissions to other containers. Trapping into kernel space for each permission change is one of the solutions preferred by most existing designs [114, 5]. The advantage of such an approach is that no changes to the ISA are needed, with the downside that programs with a highly dynamic behavior will suffer heavy performance penalties. An alternate approach is to create ISA additions that allow direct interaction between the CPU and the container manager. Another challenge for dynamic memory permission allocation is persisting the permission rights.

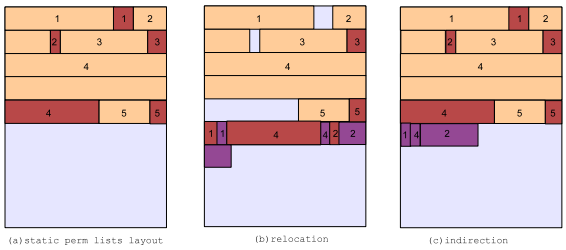

Figure 4-7: dynamicPermissionsIssues

Two approaches can be taken for solving dynamic memory permissions allocation. The first is to associate the dynamic permissions with the container definition, the same way static permissions are defined. The second is to attribute the permissions only to an instance of a container, and automatically revoke the permissions once the container goes out of scope (the entry point function in the container returns). In the first case, once a container is granted access to an allocated memory region, the permission association needs to be added to the static permission list that resides in memory. There are two drawbacks. The first is that a compacted static security data structure design [114] for the permission table is hard to modify. Any new addition to the permissions tables requires an expansion of the used memory. One could either

relocate the full permission table for the container definition where the next available space is or create an indirection list. Relocating requires invoking an expensive call in the supervisor mode but has the advantage of shorter search times. Redirections requires only short write times but long fetch delays as permissions will no longer be kept compact at sequential memory locations. Revocation is another difficult aspect: once deallocation occurs the permissions needs to be revoked for all containers that were granted access, with time consuming memory walks.

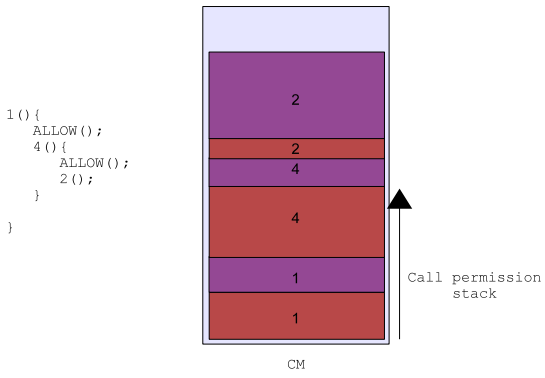

Figure 4-8: With instance dependent permission lists

The second approach is more flexible and avoids the difficult tasks of memory management and revocation issues. We allocate dynamic memory permissions only to an instance of a container rather than to its static definition. That requires permission grants to happen at every container invocation, even if a single function has a recursive behavior, it needs to explicitly delegate its following instance a subset(of complete set) of the permission it received. Using this strategy the container manager needs not handle memory layout problems related to the permission list persistence or revocations. The Container Manager needs a reserved memory region for a permission stack. This serves to persist dynamic permission between container switches.

Containers need to communicate with the Container Manager for permission delegations. For this purpose, we introduce a new primitive: *AL-*

LOW(startAddress, size, permission). We describe how this primitive is implemented later in the section. ALLOW is called by the current container to delegate access to either the caller or to another called container. The parameters needed are the memory range (as start and end address, or start address and size), the permission rights (read, write, execute, and delegate).

Figure 4-9: Using ALLOW: (a)function foo() delegates permission for its buffer to the use() container; (b) foo() is a factory function, control for the created buffer is passed to the caller, delegate permission is needed otherwise the caller can no longer assign permissions to other containers; (c) execute permission can be assigned in the same way

Note that the recipient of the permissions is implicit. There are only two options possible. Either the current context delegates permissions to the next container it will call through a valid entry point, or the permissions are passed to the caller container. A temporary buffer (the Dynamic Permission Buffer)

accumulates permissions that are created with the ALLOW primitive while the current context executes. The next security context switch dictates the recipient of permissions in the Dynamic Permission Buffer. Once a new container is invoked, the Container Manager merges the static permission lists loaded from memory with the content of the Dynamic Permission Buffer. The merged permissions are loaded into the CRR. If the Container Manager detects a return to the caller container, the content of the Dynamic Permission Buffer is merged with the CRR of the caller. Any container that expands its initial memory utilization by producing results in a heap buffer needs to appropriately pass the permissions to the caller, otherwise the allocated buffers will no longer be accessible and results lost.

We note that dynamic memory allocators (malloc, new, etc) need to have the top-most permissions level (ownership) of the full heap space. When a container needs heap space, the allocator will properly manage the space reservation then only give permissions on the buffer requested, keeping the heap control metadata inaccessible to all other containers. The code example in figure 4-9 shows both scenarios for dynamic memory permissions assignments, as well as execute rights on function pointers. Note that permissions for other static or instance dependent memory regions can also be delegated using ALLOW. This gives programmers flexibility to handle scenarios like parameters passed by reference, function pointers, etc.

Before putting any permission ranges into the Dynamic Permission Buffer, the Container Manager has to validate whether the current principal has enough rights on the permission allocation. The algorithm for validation sieves the allocated range through the principal's CRR, eliminating valid ranges. A container can only delegate permissions to a subset of the CRR, thus preventing access rights escalation (see Fig 4-10). If any memory location remains outside the sieving algorithm, an invalid memory access exception is triggered.

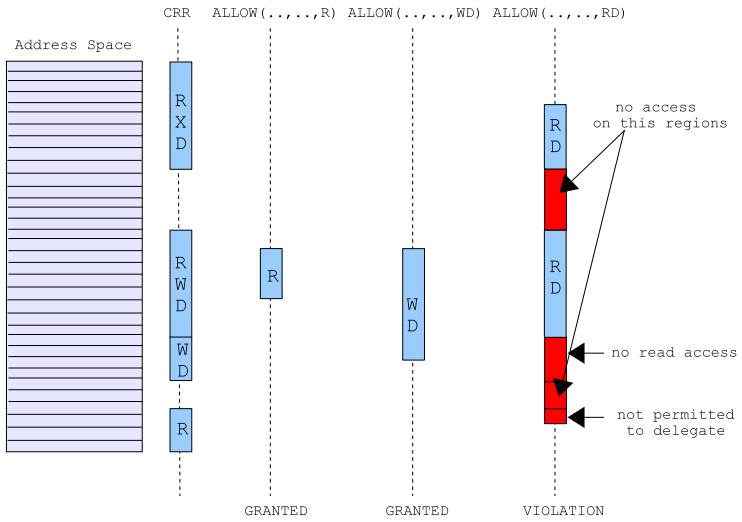

Figure 4-10: Example of correct and violated permission delegation

To implement the allow primitive, we propose a new ISA instruction, called permission control delegate (pcd):

```
pcd Rlow, Rhi, permission
```

Rlow is the register containing the starting address of the memory range, Rhi is the upper limit for the range starting at Rlow, and permission is a 4 bit long encoding of the permission, 1 bit per right: 1/0 read, 1/0 write, 1/0 execute, 1/0 delegate. The example in Fig 4-11 shows how a single invocation of the primitive is translated into executable code. It requires an arithmetic operation for preparing the address and a pcd instruction.

The same implementation can be completed with memory mapped operations, resulting in more operations (on a load/store RISC processor) and higher stress on the registers, but with the advantage of not making any changes to the ISA. Here we leverage the fact that the container manager will check all memory accesses, and an attempt to write to the current container's entry point address is never valid. The container manager can be configured to interpret three consecutive attempts to store to the container's entry point as an ALLOW operation.

```
ALLOW(buff,100,PERM_R|PERM_W);
```
(a)

```
add r2, r1, 100        ;assume R1 == &buff
pcd r1, r2, b'1001     ;add ([r1;r2],RW)
                       ;to the DPB
```
(b)

```
add r2, r1, 100         ;assume R1 == &buff
mov r3, container_addr
str r3, r1              ;push low address
str r3, r2              ;push hi address
str r3, permission      ;push perm
```
(c)

Figure 4-11: ALLOW implementation: (a) The primitive in high level language; (b) implementation using an extra specialized instruction *pcd*; (c) implementation using memory mapped operations.

4.6 Dynamic Memory Challenges

Remember the activation tree assumption: the control flow between functions always flows from the current function to the new function being called, and once the caller finishes, the control returns to the next instruction after the call. Functions are allocated memory starting at the current stack's top and then automatically scratched once the function returns. The space can thus be used again by the next function call. The heap, on the other hand, can be used arbitrarily by the program logic, no restriction imposed (other than resource limitations) on how and when memory chunks are allocated and released. While this is in itself a cause of memory related bugs and leaks, the program has to manage correctly the resource usage either using a strict policy of memory utilization or some helper heap managers, like garbage collectors. It is however the responsibility of the Container Manager to ensure that memory permissions are correctly used in any correct or incorrect memory utilization scenario. For example, dangling pointers to a dead memory area must not be used to gain access to unauthorized memory locations. In our container model (non persistent memory permissions), we already saw that malloc needs to be

part of a special container with full permissions on the heap space. (Note that several heaps may co-exist with different heap managers.) Other containers can then call the heap containers to request services to allocate and deallocate memory. Along with the heap book-keeping the heap container will grant (using ALLOW) full permissions on the allocated memory zone. The newly allocated memory can be either passed to new containers to use, sent back to the caller or de-allocated after use.

Memory de-allocation presents a subtle issue. Once a container calls free or other clean-up mechanism for heap space, the permissions associated with that space should be automatically revoked. Since no global persistent permission tables are supported by the system, we only need to be concerned with the current container call stack. There are two essential elements needed to ensure permission consistency on de-allocations. Fist, *free* has to signal the Container Manager that a revocation is needed. The second is the action performed by the container manager that voids the permission in the Container Run-Time Record, matching the deallocation request for each container in the hierarchy. In order for *free* to be able to signal the container, a *DISALLOW* primitive (authorized only to the heap manager) is needed. Invoking DISALLOW creates an *inverse* permission in the Dynamic Buffer that gets propagated to the caller container, voiding existing dynamic permission.

Implementation wise, we have two policies that can be supported: first, allow child containers to deallocate memory, even if there are other containers in the stack that have access to the memory range; second, only allow the last container that has permissions to the heap buffer to deallocate it. One assumption is needed: the same chunk of allocated memory is also deallocated. (For example, buff = malloc(); free(buff+1); is prohibited—this will also break the allocation/deallocation correctness, so it is not a limiting assumption.) This allows permissions to memory ranges to be matched exactly and also

ensures consistency.

For the first policy, allowing any child container to call free(), the Container Manager has to propagate (by removing the permission from the CRR) the *inverse* permission generated by DISALLOW to the other containers in the stack that hold any permission on that range. The negative permission can be then discarded. The dynamic permission buffer can be used to hold these deallocation ranges.

The second policy, goes a step beyond into the semantics of allocation/deallocation policy to enforce consistency in the way heap memory is used. Since a child container can render the caller inoperable after the return if deallocation was not expected, the implementation allows again for two options: either add *deallocate* as a permission that can be granted to child containers, or enforce the right to call free on a memory range only when the caller is the last in the run-time stack to hold dynamic permission on that range. The first option adds an additional permission encoding to a memory range and requires the Container Manager to automatically propagate *deallocate* to the caller, when the ALLOW primitive is used for that purpose. The second requires verifying the ownership of permissions on a range when free is invoked. Since the latter is a more restrictive and inflexible policy, we only mention it as an option.

4.7 Sparse Data Structures

Complex data structures such as linked lists, trees or graphs pose a challenge for right assignments. As data cells are distributed over a large memory space, in order to pass the full structure to another principal for processing, the same processing as traversing the data structure is needed for security attributes assignment (Fig 4-12). In order to optimize such scenario, we offer a variant ALLOWM(n, addressOfSecRange) (allow multi) that receives as parameters

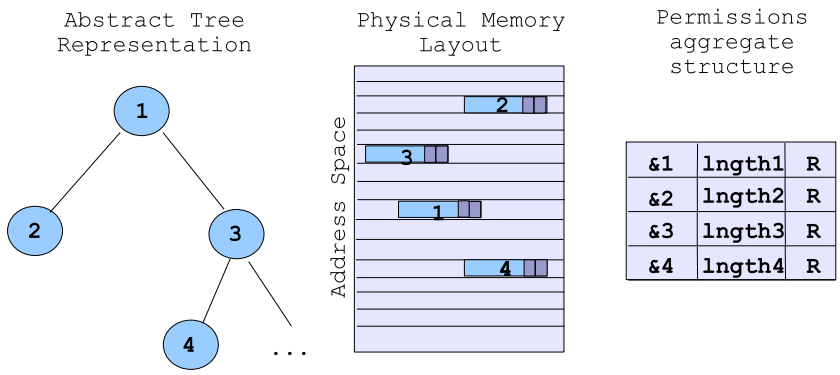

Figure 4-12: Sparse data structure (tree): logical structure, memory layout, security data structure

the number of ranges and the location in memory of security permission data structure that holds multiple delegation attributes. Such security attributes can be made part of the data structure itself and maintained by the base operations (such as add, remove, etc) and reused without re-parsing the structure. Other language primitives can be used (such as security keywords: private, protected, public) and the compiler can automate many of the programming tasks. Figure 4-13 shows an example usage for a C-like language, once the tree is traversed and the permForT assigned, the Container Manager will execute ALLOWM by grabbing all sequential N security attributes found at the addressOfSecRange and puts them in the Dynamic Permission Buffer.

4.8 Architectural challenges

The processor architecture has to be changed to accommodate the reference monitor in the memory path. We refer the reader to [86] for a detailed analysis on the implications of access control checks on the security and performance of the processor pipeline. There are three options to consider.

The first option is to prevent any instruction or memory that was not validated from being fetched into the processor. This is a conservative approach

```
int nperm = 10;
struct permission{
  addr_t start;
  int    length;
  sec_t  perm;
} permForT[10];

foo(tree t){
   for (int i = 0; i<10 ;i++) {
     permForT[i].start  = &t.node[i];
     permForT[i].length = sizeof(node);
     permForT[i].perm   = PERM_R|PERM_D;
   }
   ALLOWM(nperm,&permForT);
   use(t);
}
```

Figure 4-13: Example using permission assignments for a sparse data structure

and requires minimal changes to the pipeline. However, the downside of this approach is that the validation mechanism becomes part of the critical path, thus the performance impact of access control checking becomes a bottleneck, rendering this solution impractical.

The second option is to perform the access control just before the commit stage of the pipeline. This approach applies to both in-order, out-of-order and speculative executions. The commit (or for an in-order pipeline, the write-back) stage stores instructions until all exceptions and misspeculations are solved, helping with precise exception handling and keeping a consistent processor state. Any instruction in the commit buffers cannot retire until both the instruction and the data accesses are validated. Since instruction fetches are performed in the earliest stage of the pipeline, the information about the PC (that identifies containers) is located with the instruction in the commit queue. The commit based access control needs changes to the pipeline for forwarding the PC for each instruction to the commit queue. This way principal identification and control flow can be kept consistent along with the processor state.

A third middle ground solution is to allow speculative execution of instruc-

tions only within containers. When the container manager detects a fetch that does not belong to the current container, it stalls the pipeline until all instructions that belong to the previous container have cleared.

4.8.1 Pipeline Integration

A modern out-of-order processor contains a complex pipeline that is tuned for maximum execution throughput and minimum stall cycles. Our design accordingly strives to minimize the delays caused by memory validation and maximize code parallelism while balancing the need for security checks and updates to the CM state. All memory accesses—loads, stores, and instruction fetches—must be validated by the CM, and we modify the pipeline to handle each kind of memory access. Figure 4-14 shows a diagram of a simplified pipeline with the integration of the CM. We modify two (conceptual) stages of the pipeline: fetch and commit.

The fetch stage modifies the effective program counter (PC) address, which identifies the currently active container. (Using the PC was a conscious design decision that elides namespace problems and potential attacks that would decouple the name of a container from the instructions it executes.) The instruction fetch stage routes through the CM so that the PC is available to infer the current security context. While executing within a container the fetch will belong to the current container and the CM does nothing. When the PC indicates a new container is being entered the CM initiates a security context switch, during which time the fetch stage of the pipeline stalls until all instructions belonging to the active container clear the pipeline and enough of the new security context has been loaded to validate the fetched instruction. Most compiler optimizations and code dependencies do not span function calls, and we found that stalling the pipeline simplifies CM state at an acceptable

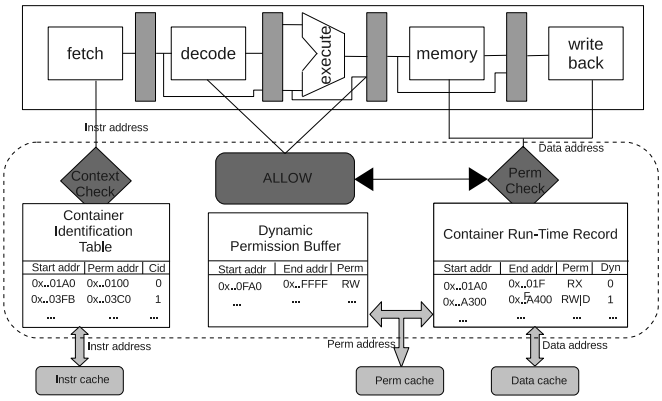

Figure 4-14: A simplified view of a processor pipeline. The CM hooks into the fetch and commit (write-back) stages of the pipeline to validate memory accesses efficiently.

performance vs complexity tradeoff.

Other than instruction fetching, the memory accesses as seen by the pipeline are speculative loads/stores that make visible changes to architecture state during the commit (or similar) stage. The memory address of such operations are known shortly after the execute stage. Validating such memory accesses can be done in two ways: as a parallel memory access permission check with the memory access or as an in-order extension of the commit stage of the pipeline. The second approach is simpler to implement and has no influence on the sensitive tuning of the pipeline timing. Another advantage is that waiting for the speculative state to be resolved results in fewer lookups into the permission tables because the squashed operations will not try to validate permission.

The ALLOW primitive and its variants are implemented as specialized implementation dependent instructions, and executed by the container manager in a co-processor configuration.

4.9 Accelerate Permission List Searches

One of the key elements of the Container Manager is the ability to verify if a memory access is in the authorized list associated with the active container. Since this is a frequent operation that occurs on every memory access, the time spent during the check is a critical factor in the performance of the system. In order for the access control verification to impose little or no impact on the performance of the system, the penalty incurred by the search in the access list needs to incur at most the same delay as the memory access.

Content Addressable Memory (CAM) has been widely used for fast searches in applications like cache indexing using Translation Lookaside Buffers (TLB) and high speed routers. CAMs are special designed memory modules that allow for a value to be located in a storage cell rather then fetching the value based on an address. Although a CAM storage cell is larger than a traditional SRAM (for storing one bit of data a CAM storage can be implemented using 9 transistors compared to 6 transistors for a bit of SRAM [69]), one lookup can be finished in only one cycle. One shortcoming of typical CAMs is that the lookup can only be done for a fixed value, and that checking for a range of values needs several accesses.

Ternary CAM (TCAM) allows for don't care bits (*) as match values. This design was motivated by application of CAM in routing tables. A TCAM storage unit can be implemented using 16 transistors [62] but can be used for matching a wider range of value patterns (ex. a TCAM entry of 1,*,0,1 matches both 1,0,0,1 and 1,1,0,1). A typical way to handle range checking, is to replace each range with several entries in a TCAM, each covering a portion of the desired range[98]. For example, the range 2-10 can be partitioned into the set of patterns 001*, 01**, 100* and 1010. In general, any sub-range of a k bit field can be partitioned into 2(k-1) such patterns. This motivated the

design of specialized CAM with extended comparator circuitry, specialized in applications that require range checking(port ranges in IP lookups) [90].

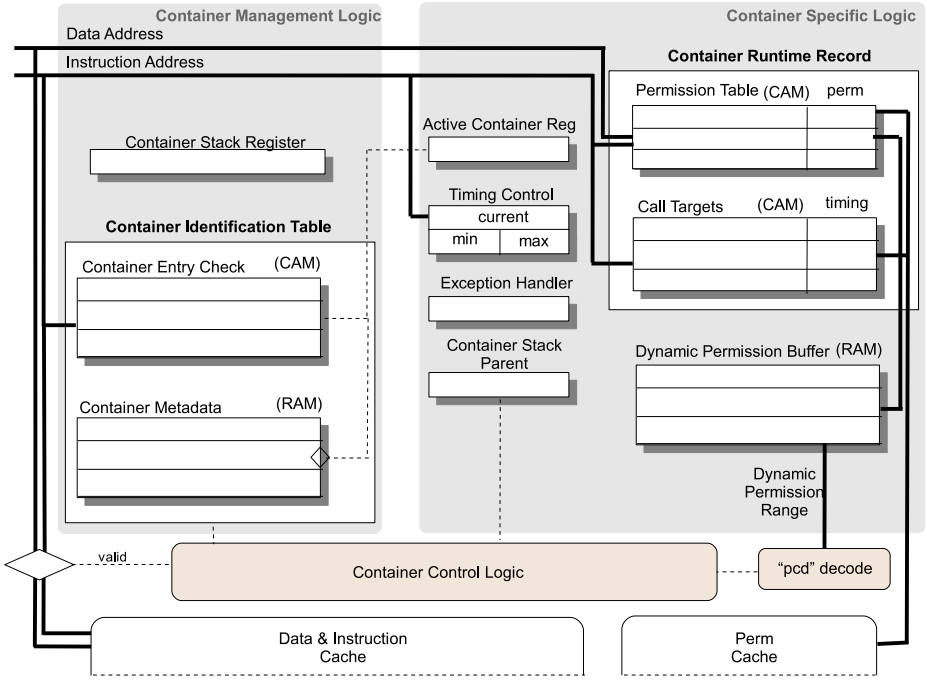

Figure 4-15: Next Level Detail

We propose using a similar design for the Container Run-Time Record table that checks if a memory access is in a range of permissible locations. Figure 4-15 shows all search tables in the Container Manager. The Permission Table in the CRR is the only range checking CAM as it needs to validate memory ranges. The Container Entry Check in the Container Identification Table, uses regular CAM memory, as it holds all entry points into each container, thus detecting container context switches. The Call Targets CAM validates the calls to other containers, thus containing only a list of instruction memory addresses.

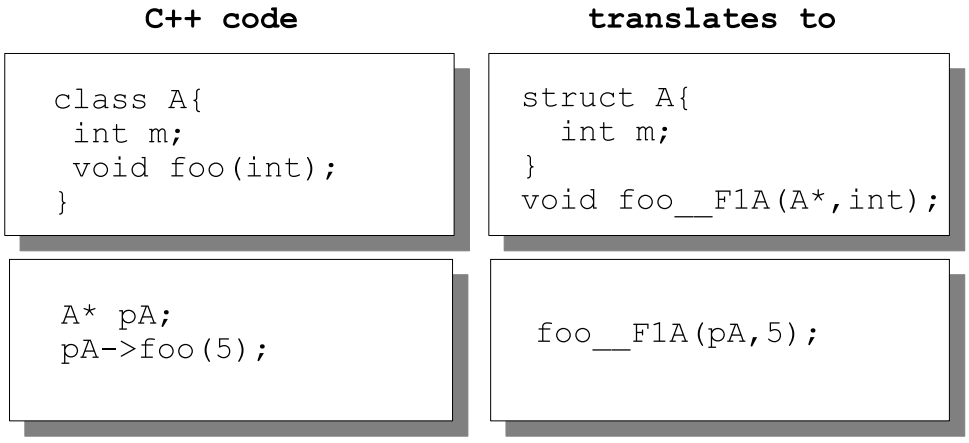

Figure 4-16: C++ implementation details

4.10 Object Oriented Design

Object Oriented Programming (OOP) brings powerful organizational benefits over typical procedural programming. The program written in such a language is closer to the abstract concepts that the author wishes to code in the program, rather than describing the sequence of machine steps that will be executed. The objects are the building block of OOP and writing a OOP program has two steps: first describing the objects in terms of structure and behavior and second orchestrating the interaction between objects. Key concepts such as *encapsulation*, *inheritance* and *polymorphism* make the program design more robust, easier to understand, efficient and reusable. In this section we analyze the compatibility of key OOP concepts with the Container model. We take *C++* as the representative language for the OOP class, as it is the most permissible and insecure, yet very popular.

In order to evaluate if the Container model is suited for Object Oriented programming model, we need to review a few basic techniques that the C++ compiler uses for few key OO features [92]. An object of a C++ class is instantiated by creating a contiguous region of memory, each member variable sequentially allocated like a C struct. In fact there is a one-to-one corre-

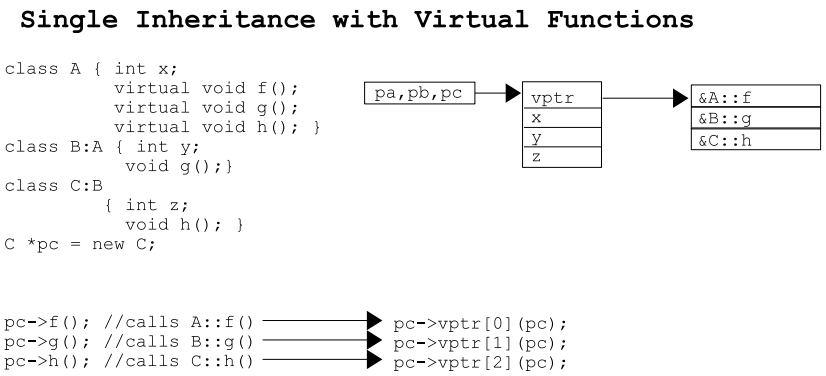

Figure 4-17: Single Inheritance With Virtual Functions

spondence of C++ code and C struct in terms of member functions. Class methods are implemented as regular functions calls that have a hidden parameter, a pointer to the first byte of that region of memory, also known as *this* pointer. So far this is perfectly compatible with our container model. But there are some complications.

One complication is the use of virtual functions. The address of the executing function is not known at compile time and it is determined at runtime from a table of function pointers, called a *virtual table* (details in Fig. 4-16,4-17,4-18, 4-19). Fortunately, the ALLOW primitive does not depend on the location of the calling function, and will delegate the requested permissions for the next function call, as long as that function is listed with EXECUTE permissions in the caller's permission manifest.

The second problem is exception handling. Since exceptions are an abstraction of set-jump/long-jump, the exceptions do not match the activation tree structure. So unless the code that generates and catches exceptions belongs to the same container, exceptions need some additional support (left as future work).

The third issue is a mismatch between the concept of class accessors (*private, public, protected*, etc) and the run-time enforcement of their intended

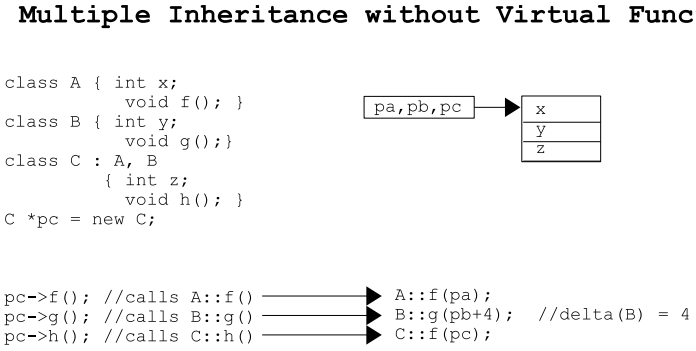

Figure 4-18: Multiple Inheritance Without Virtual Functions

purpose. In current architectures, it is almost impossible to impose run-time restrictions on the accessors, as a programmer understands it. With no memory access restriction mechanisms it is trivial to forge pointers into *private* members of base classes. Thus the accessors are only compile-time restrictions that are not checked at run-time. With a memory protection mechanism we would like to enforce the accessors and define the memory permissions such that at run-time it is not possible to violate them. Unfortunately, this is impossible to achieve in our protection mechanism. Because the permissions are associated with code regions, our model requires that a container that requests memory be given full permissions to it. This way a parent container needs access to the allocated memory regions so it could further delegate access, thus allowing the parent container to access "private" members. Access to "private" functions can easily be restricted. However, enforcing accessor permissions on objects by encapsulating each object in a container requires a persistent model of permissions to associate memory accesses with the static definitions of functions rather than to function instances, which is not possible in our model.

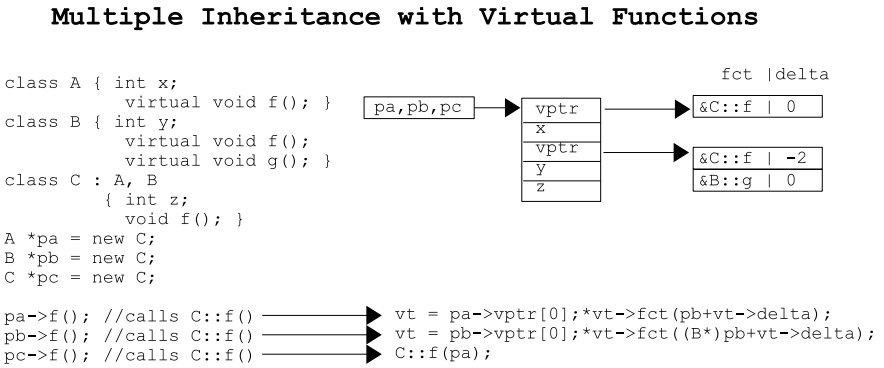

Figure 4-19: Multiple Inheritance With Virtual Functions

4.10.1 Container private data

From the perspective of the class designer, *private* means fields marked as such should not be directly accessible except by member functions of the class itself, *public* means accessible by anyone else. Our goal is to create the run-time system that enforces this principle even in an adversarial scenario. A user of class should NOT be able to access any private marked data, or a memory access violation should occur. In this model *protected* (accessible by class members or derived classes) has can be translated into either public if the principal is part of a derived class or private for anyone else. From a security perspective, an adversarial programmer can write a derived class to access protected members, thus it is still up to the compiler to properly derive the access lists accordingly. A run-time system that enforces "private" accessors needs to support the ability to dynamically create and manage principal private data segments that persist across principal activations. In other words a container should be able to allocate memory and explicitly control which other container is allowed to access it, without granting the immediate caller or callee any access to the memory segment. As objects can be instantiated on the stack, on the heap or on arbitrarily pre-allocated object pools (for example using the placement new operator), statically derived permissions are not a sufficient mechanism

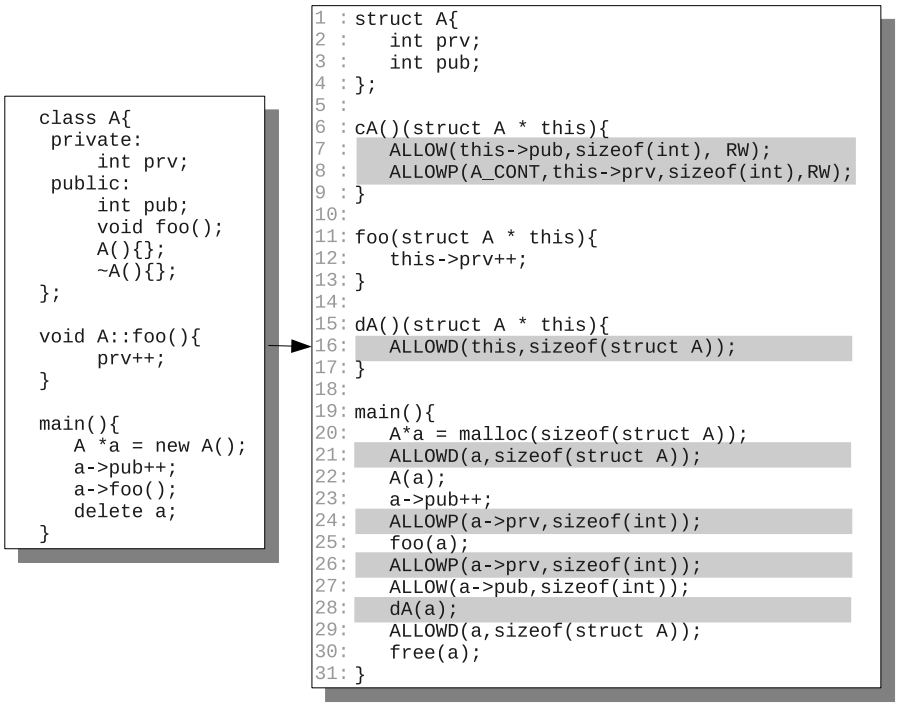

Figure 4-20: Cpp code.

to support class private data. In the traditional container model a container is automatically granted access to the top of the stack when invoked, and heap permissions are passed from the heap manager to the caller by using ALLOW. If allocated on the stack permissions the calling principal owns that region thus we would not be able to enforce access to private object data. On the heap, the typical implementation for object instantiation is: the caller allocates a heap buffer the size of the object then calls the object constructor. Upon return from the constructor, the caller receives permission to the allocated buffer (by the ALLOW executed in malloc) then passes it along to the object constructor, but still maintaining permissions to the buffer.

In order to implement container private data, we introduce the following rules and primitives to the reference monitor(the container manager) : In order for a container to be able to claim ownership of a memory region, the calling container(and any other container on the calling hierarchy) need to

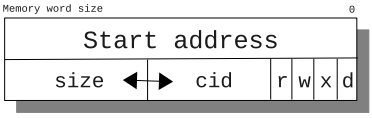

Figure 4-21: Memory Permission record. Each permission record is 2 words(16/32/64) long. More allocated bits for the size allows for larger memory segments, while longer cid allows for finer grained containers. **cid** 0 is used for non-private memory segments

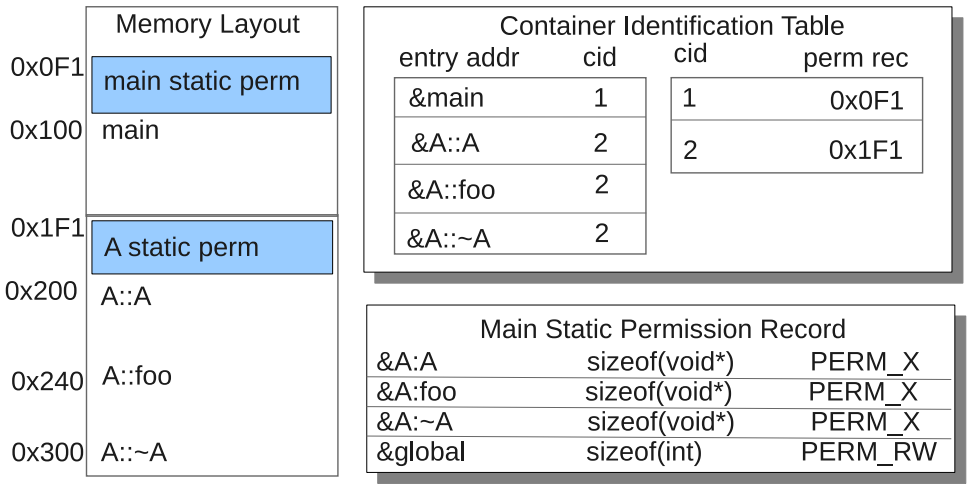

Figure 4-22: Static permission layout

relinquish the rights for that region. ALLOWD (D - Delete/Delegate), directs the container manager to revoke access from the current container (splitting permission ranges if needed) and delegate access to the called container. A class constructor has now full ownership (no other container has permissions to it) of the private memory segment needed for the object The class constructor has to persist the permissions to the private segment. It does it using ALLOWP (P - Private) by specifying a container id(its own container id typically). The CM creates a dynamic permission entry that will be picked up automatically by any other container with the correct id. Since the mechanism need not be specific to OOP and C++ in particular, ALLOWP does not enforces the permission entry to have the same ID as the creator. A container id field equal to 0 is used for default implicit container recipient (any recipient can pick up this permission). Passing the private permissions along the

call stack is explicitly performed in untrusted containers. Using ALLOWP, a container function is responsible for passing along the permissions so they can reach their intended targets.

Aside for the stack managed permissions, we considered a) maintain a permission list associated with the static definition of the class. Just like static permissions are stored for each container, dynamic permissions could be similarly placed. The advantages of this method would be that the calling code would not have to delegate/manage explicitly permissions for class private permissions. The container manager would find the dynamic permission lists at a known location whenever a security context switch occurs. The downside is that due to its dynamic nature, this list is hard to manage and does not scale well. b) maintain an instance list with the class, a concept used in hardware accelerated garbage collection[39]. Similar to a), if we keep a pointer list of all class instances would reduce the size of the dynamic permission lists. The problem with the dynamic lists still exist and this would make tie the CM implementation to the particularities of some programming language implementation. c) associate permissions with the data. Interleaving permissions with data segments eliminates the unmanageable list problem, but introduces 2 other complications. First, without a separate mechanism for differentiating a memory permission token from regular data (such as memory tagging [5], [87]), malicious code can forge permissions. The second, is that functions that do have private data, need to be marked as such and managed separately

4.10.2 Exceptions

Managing code that handles errors in traditional imperative programming is a difficult and tedious task. Most techniques (return error codes, global error states, error handling functions) clutter the code and are prone to program-

ming mistakes. Because we seldom have enough information where the error occurs about a recovery policy, we pass the error up the call stack until a resolution can be formulated. Exception handling offers a better way to manage error states. Using an elegant and minimalistic syntax: try/catch/throw, programmers delegate most of the stack management tasks to the compiler. Besides, compared to C-style error reporting, C++ exceptions bring other benefits: error handling code is identified using typed objects instead of error codes, intermediate functions have no extra code for passing along transiting errors, stack cleanup and object destruction is properly handled and exception handling code clearly separated from "normal" processing.

The exception mechanism, depending on implementation, most often does not follow the call/return semantics, thus making it potentially incompatible with the stack-based permission management proposed so far. Due to many implementation variants, many architecture dependent, we conceptually analyze a few major strategies and how they impact our fine grained permission management. The two main strategies currently used are: managing the exception state with inserted code along the execution path, and the second, building an action table to be used by the C++ runtime only when exceptions occur. And of course any hybrid in between. Exception handling does not come without run-time overhead in both terms of space and time and extensive efforts from compiler writers go into tuning implementation details. The first strategy has the main disadvantages of time delays on the normal execution path, the second uses large lookup tables raising the program image by as much as 10-15%. The second is widely preferred in modern implementations because the consensus is: it is preferable to minimize the execution path that does not deal with exceptions at the expense of considerable delays when exceptions do occur.

The traditional code managed exception handling uses setjmp to create a

list of jmp_buf for each level of a nested try block. On a throw, the program fist locates the landing pad (the location where the exception will be handled) from the jmp_buf list and then uses a longjmp to hop back to the appropriate catch block. Automatic objects that need to be properly destroyed are appended to a separate list associated to each jmp_buf. Following this list objects destructed as the stack is unwound.

Table-driven exception handling allows the C++ runtime library to perform the necessary cleanup when an exception is thrown without adding maintenance code on the normal execution path. A first table maps the PC at the point where an exception is thrown to a series of actions to be performed. The action table lists the destructors to be called, the stack adjustment and how to lookup for the exception handling code. This approach limits the amount of code that executes on the non-exception path at the expense of slower exceptions and larger memory footprint.

Due to their use in legacy implementation of exceptions, setjmp/longjmp can be adapted to work with our fine grained permission mechanisms. On setjmp the program context: stack pointer, frame pointer and program counter are saved in a local structure to be restored by a subsequent longjmp. As our execution context now includes the security context maintained by the CM on a separate stack, we need to save it as well. Since the security context (the permission stack and frame pointers) can not be stored on the program stack (exposed to manipulation by adversarial code), the CM has to store it as instructed by setjmp on the permission stack. Longjmp instructs the CM to restore an alternate context than the top of the stack.(with the appropriate checks to validate that the container where setjmp was called did not return already and that the stacks are always synchronized).

Unfortunately support for generic setjmp/longjmp is not sufficient for exception handling. All exception implementations rely on global tables whose

unruly modification can compromise the security of a program. Several global structures direct stack unwinding and define landing pad selection: an exception stack lists unhandled exceptions, action lists and unwind tables define cleanup instructions and compensation code. The code that manages stack cleanup and control flow transfer resides in the language supported run-time(a combination of low level architecture primitives - ex. for stack unwinding and language specific "personality" functions that understand the structure of the exception objects). Although the exception handling code(the body of catch blocks) do not need special privileges for execution - they operate under the same principal as the parent function - the exception run-time should operate with elevated privileges to: all exception related tables and the special ability to acquire all dynamic permissions on the CM managed permission stack as it unwinds. As each stack frame unwinds, it allows the exception handling runtime to pass along appropriate permission to the destructors thus enabling orderly reclaiming of memory. We currently investigate a mechanism that allows for container private exception handling, a system that will eliminate trust in global handlers and that will allow for orderly recovery in the presence of adversarial recovery code. Conceptually three distinct changes to the exception handling mechanism are needed to allow a container-private exception treatment. The first requires each container to provide its own methods for stack unwinding and compensation code execution. The second step is to divide the exception lookup tables for each individual container. The third is to place the global exception lists into containers of their own and enforce the operations (add, remove, search) to be performed on he lists with the existing fine-grained protection mechanism.

Other C++ considerations

- **Encapsulation & Inheritance** Inheritance and encapsulation are pow-

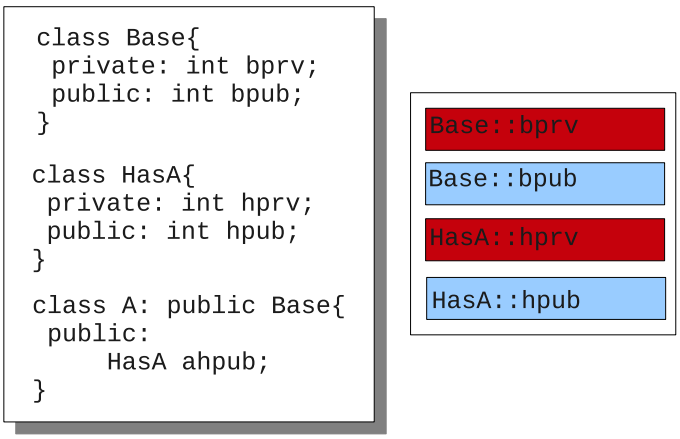

Figure 4-23: Permission Fragmentation. Data layout when using encapsulation and inheritance leads to interleaved private and public access to object memory

erful paradigms of object oriented languages in general. Even when intricate multi-level class hierarchies are used, the compiler generates one compact data structure with all members stacked based on the hierarchy. This approach optimizes locality and for static casts simple and fast relative offsets locate the start address of various members of different classes. In case of enforcing private permissions, both inheritance and encapsulation leads to a fragmentation of permission maps for the object (Figure 4-23). Although the C++ specification does not enforce or recommends any strict memory layout for objects, thus allowing us to potentially regroup object members to alleviate this problem, doing so would break C++ ABIs and would break interoperability with traditional systems.

- **Virtual Functions/Polymorphism** Virtual functions are one of the most important languages features for OO programming. It allows us to implement true polymorphism. Using the *virtual* keyword in a base class allows derived classes to replace (override) the default implementation. In our code, with a reference to the base class we call the virtual function and the code in the derived class is called. C++ implements this

functionality through dynamic binding. The address of the code called is not determined at compile time, but it is computed just before the function call by using pointer to a virtual table(*vtable*) (a static table of pointers created for each class that has virtual functions). In connection to the permission management mechanism, there are two aspects to be considered with respect to polymorphism: how is control flow handled (permission to execute) and is the proper access to class fields given to the virtual function? The first question is solved by using ALLOW with execute permission flags. Since the compiler can not determine it at runtime, the address of the override is not part of the static permission list. The compiler uses the virtual tables to calculate the address of the function to be called and inserts ALLOW execute just before the function call. To protect virtual tables from being maliciously altered, they should only be given read-only permissions to any code. The second problem is harder to solve. Normally the code invoking the virtual method should only statically be aware of the base class and does not need to be aware of the structure of the derived classes. But the called function in the derived class may access fields in the extended object not present in the base class. Permissions to those fields need to be passed to this functions in order for it to access them. Our solution is to extend the virtual tables with object permission layout information(vtables already similar extensions for run-time type identification(RTTI)) so the compiler can generate the appropriate ALLOWs at runtime.

- **Friends** Giving external functions(or entire classes) access to private fields is possible in C++ by declaring the external entities as "friends" of the class. The "friends" are then allowed access to private fields. With container protection mechanism we support this language feature as well.

The only needed instrumentation is that constructors need to also create permission to all friend classes in addition to the internal class methods.

- **Placement new** In addition to the typical "new Obj()" operator used for allocating objects on the heap, C++ uses a variant "new(addr) Obj()" for what it calls "placement new" operator. It allows a programmer to place an object at a predetermined memory location, used primarily for preallocated object stores. Implementation wise, the placement new is a subcomponent of the new operator (bypasses the allocation). Permission delegation happens is the same for both operators, before the constructor is called ALLOWD is used to relinquish permission to the new object memory.

- **const** The "const" keyword allows the compiler to enforce read-only access to certain variables. With fine-grained memory access control we can extend the "const" static validation to run-time enforcement, by using only "read" access to memory.

- **limiting protection** Suppose we would like to restrict access to certain object fields in the way described so far, without enabling full run-time protection for all private fields. This can be easily accomplished by either extending the language with a "secret" keyword or a pre-processor directive.

- **serialization** The memory protection does not extend to secondary storage or span across memory spaces(between processes, file storage, sockets). A full chain of trust including the serialization routine, the kernel and encryption/signature are needed for extending protection beyond a linear memory space.

4.11 Containers in multiprocessing context

Although specifically targeted towards embedded systems, which have limited support for a general purpose operating system, it is important to acknowledge the migration of general purpose computing to the embedded world. We notice that consumer products, from portable media players, secure storage devices and communications terminals to automotive control software, need more and more features from personal computers such as multitasking, threading and hardware abstraction. Two important aspects are briefly covered in this section related to simultaneous execution support for containers: multi-threading and multi-tasking on one end and multi-processing on the other.

Due to the multitude of possible architecture designs and hardware layouts, both aspects need specific architecture customizations depending on the memory and processor communication layouts. A couple of abstractions will allow us to define the issues. A *security context* represents a currently executing state of a set of principles defined by a Container Identification Table. That includes the execution state and the set of permissions it has on the memory space (either physical or virtual). A process or a thread has its own security context, therefore they may have a unique view of the memory space. These are important requirements both for the security aspect of applications and for the implementation requirements. Multi-tasking usually requires the intervention of a supervisor process(OS) for loading, scheduling and switching the execution contexts. Since each task (process) is confined by a complete set of container definitions and a security context, we rely on primitives of the operating systems for managing the security context. The hardware (Container Manager) needs to provide the primitives that allow the supervisor to: load, save and restore the security context with the corresponding process. The critical operations required by a security context switch are: *save*:

saving to memory the dynamic permissions tables, the time-counters for each timed inter-container call, currently executing container and the location of the overflow stack; *load*: loading the Container Identification Table for the process that will start execution, restoring the dynamic permissions table, reloading the static permissions for the current executing container with the overflow stack. Note that the CIT and the static permission lists do not need to be saved in memory as their content does not change and can be reloaded from the original location when needed.

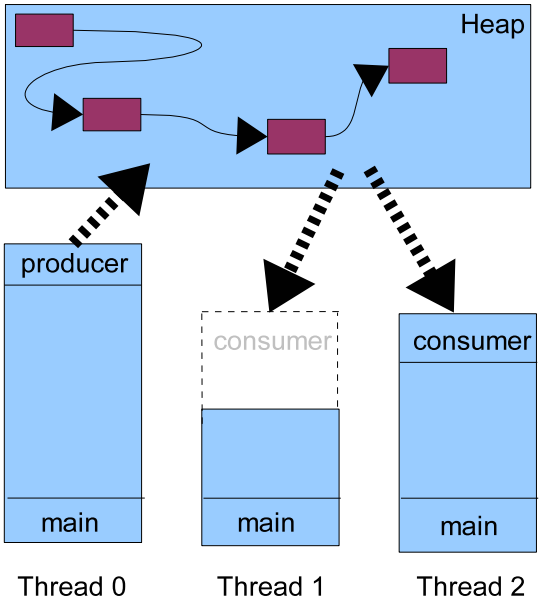

Figure 4-24: Multithreading

Figure 4-24 pictures another challenge of the container manager design. The scenario is a typical producer-consumer implementation of a parallel algorithm. The producer thread(s) creates tasks and adds them in a queue, the consumer(s) verify the availability of tasks and processes them, removing them from the queue. Without explicit permission passing from one permission context to another, the consumer threads are not allowed to access the shared memory region. Three solutions are suitable for this problem. The first is to rewrite the program to use a typical mechanism for inter-process com-

munication: message passing. Unfortunately this solution involves creating unnecessary copies of the memory regions that represent the tasks and defeat the purpose of the shared memory model that threads provide. The second solution is to use a statically shared heap space that threads can have full permissions on. This solution simplifies the sharing of permission but breaks the least privilege principle of our protection mechanism. The third solution to controlled sharing in a non-activation tree execution scenario is to rely on the services of the kernel(supervisor) for distributing permissions to the other principals sharing the memory space. The kernel maintains a list of shareable memory regions between threads and updates the permission stacks accordingly. Each container is required to specifically request a kernel service to pass permissions across threads. Since this functionality is not automated by the hardware, minimal changes to the container manager are required. The kernel calls allow for a great flexibility of sharing. A memory region can be shared between all threads in an application, only a specific thread, or across processes(with more changes to the OS). The OS call pushes the permission to the existing shared memory region of the specified threads and adds the range to all new threads that should be allowed to access the shared memory. Revoking is accomplished by scanning all permission stacks and subtracting the shared range from the existing lists. Although revocation and sharing are costly operations (since they involve software implemented memory searches and changes), they are necessary in order to maintain a strict access control and preserving the least privilege principle.

4.11.1 Concurrency Support

The traditional model of processes and threads strongly relates to protection boundaries: Within a process one or more execution *threads* allows for multiple

sequences of instructions to execute concurrently sharing access permissions to process resources. Many modern programming paradigms require concurrency, and they often ignore the notion of protection among the various execution threads because the process model does not give them any choice. With some systems software support containers easily supports concurrent programming with a twist in the protection model: instead of having multiple execution sequences (threads) inside a single protection domain, containers supports multiple protection domains inside a single execution sequence.

We refer to the continuous execution sequence containing one more more containers as a task. (Some readers may be familiar with Ada or real-time tasks, which correspond well with our use of the term). Multiple tasks have isolated stacks and heap space, but they can directly share memory if their containers have appropriate permissions to the shared data structures. Tasks share access to static data by setting the appropriate static permissions at load time. As opposed to traditional lightweight processes (threads) the heap memory is not shared between tasks. In order to share data between separate tasks the sharer needs to allow access explicitly to the data by invoking a new system call, `share`. `share` adds a new entry in the appropriate destination container's dynamic permission list.

When the OS does a context switch—swapping from one execution context (task) to another—it saves/restores the execution context, such as registers and the PC, to/from a data structure called the task (thread or process) control block or TCB. When the OS initiates a task switch, it notifies the container manager by a dedicated instruction *CTXSWITCH* that handles the container context switch before allowing the new task to continue execution.

CTXSWITCH saves/restores the Dynamic Permissions Buffer and dynamic permissions from the CRR to the top of the permission stack of the previous/next task. The instruction also saves/restores the Container Identification

Table as part of the TCB.

4.12 Performance Considerations

Our approach for fine-grained memory protection involves many low-level operations in the processor that can cause performance loss when using our security solution: security context switches may happen as often as each function call/return, and additional code executes to delegate dynamic memory permissions. We have given careful consideration to each of these potential sources of performance loss in order to minimize the negative impact of our approach, and we present some optimizations to reduce overhead.

Security context switches When a security context switch happens and the new context is not loaded into the CM the processor is stalled and no instruction is allowed to retire. We explore three strategies for optimizing context switches: partial container checking, container windowing, and bus widening.

Partial checks allow loads and stores to be validated against a partially loaded permissions list. If a hit occurs the memory operation is valid, otherwise the CM continues to stall the pipeline and retries the memory access when the permission list finishes loading.

Windowing mimics the register window behavior of the SPARC architecture.The basic idea is to have storage in the CM for multiple CRRs that can be rotated on a security context switch. Writing out the older contexts to memory could be deferred or done in the background while execution continues with the new context. Such a mechanism would reduce many of the memory transfers during a security context switch, but would increase the chip space used by the CM.

Bus widening reduces the bandwidth bottleneck between the CM and the permissions cache. For register to memory operations a word-size bus (16/32/64 bits) is typical: matching the size of the register to the memory load/store makes sense, since a larger bus will have higher latency (and power cost) and provide little throughput benefit if the data size is less than the bus width. For moving data between the CM and the permissions cache, however, the data sizes are larger than one register—especially during a security context switch, for which the CM transfers large continuous sequences of permission lists. Increasing the bus width allows the CM to move data at greater throughput because plenty of data are available.

4.13 Other Considerations

Since registers are not directly part of an addressable memory region, our Container Manager will not enforce rights on the processor registers. If security sensitive data is left in the registers after a function returns, it can be potentially be accessed by self-modifying code attacks against code structure (such as buffer overflows). We consider the responsibility of each security context to clear any security sensitive content left in registers before any context switch.

We also assume that aggressive compiler optimization and prediction does not occur across containers. Otherwise the activation tree structure is not preserved and false container switches can occur. We do not address security context prediction in this work. Any inter-container prediction or optimization is correctly handled by our architecture design. Non activation tree behavior (goto's and set-jump/long-jump) are only permitted inside a container, but it is not allowed cross-containers.

Chapter 5 – Experimental Results

5.1 Initial Experiments

In order to evaluate a system enhanced with our container enforcement mechanism, we conducted a series of simulations that provide an accurate comparison between an unprotected system and a security enhanced device. As expected, the extra validations, code instrumentations and run-time metadata cause a performance penalty. For our test bench we used a cycle accurate simulator, the Simplescalar [7] simulation suite, adapted for an ARM processor architecture [10]. We chose a representative set of benchmarks targeted to embedded systems and we used for compilation a gcc(v3.3) cross compiler for the ARM instruction set. In order to investigate the performance impact of our approach, we measured the overall performance penalty in terms of additional processor cycles taken to complete the execution when using our security mechanisms versus original uninstrumented code .

The benchmarks chosen for the simulations were computationally-intensive applications from MiBench [32]: `bitcount` – tests the bit manipulation abilities of a processor by counting the number of bits in an array of integers; `crc` – checksum calculation for a file; `dijkstra` – an implementation of the graph algorithm for calculating the shortest paths between nodes; `fft` – Fast Fourier Transforms used in digital signal processing; `sha` – the standard secure hashing algorithm used in most security transactions; `stringsearch` – search algorithm for given words in phrases using a case insensitive comparison algorithm; `susan` – an image processing suite with three variants. In addition, we selected three data intensive benchmarks from the Data Intensive Systems (DIS) benchmark suite [60], and a heap intensive benchmark - known as the Richards benchmark. Figure 5-1 shows the frequency of function

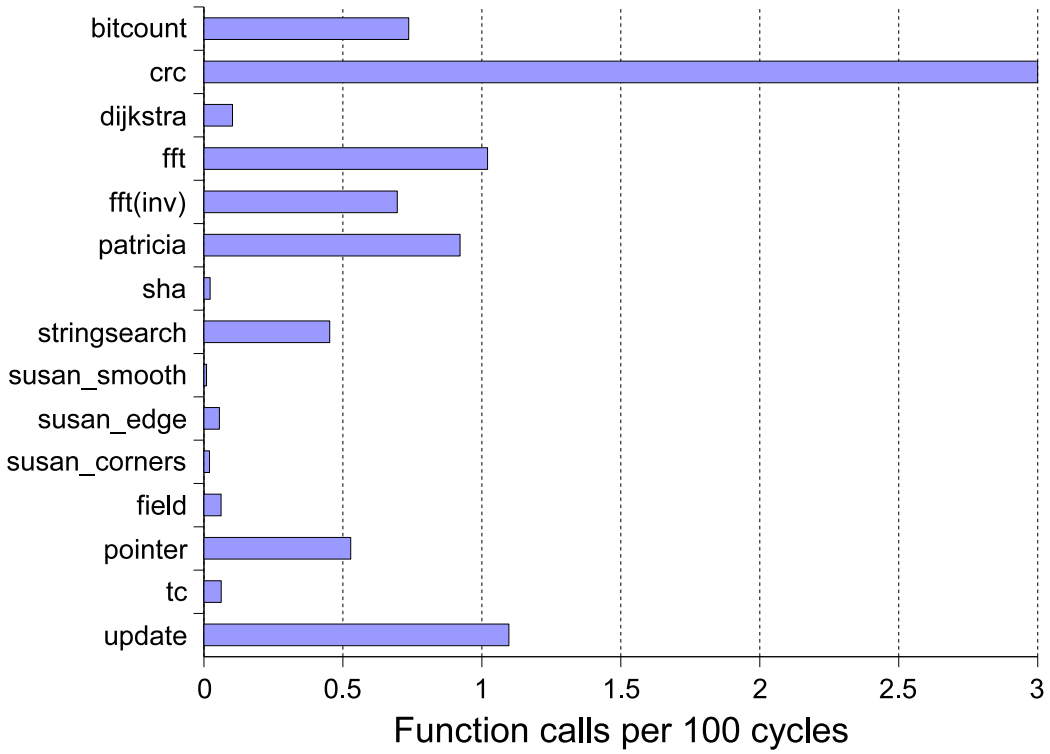

Figure 5-1: Frequency of function calls.

calls for each of the benchmarks as this characteristic significantly influences the overall performance of our system (since containers are defined at function level, or grouping of more functions, the permission metadata for a container is loaded/saved at function calls/returns).

In order to gather the memory requirements of every container partition of the original code, we have augmented the simulator code with additional profiling capabilities. The first action was to create a mechanism for defining containers, called the container manifest - that contains the exact partition of the program into modules, offering the possibility to group multiple functions into a single container. Next we traced the memory accesses of each container to construct the list of static permissions for each container and in the same time catalog each type of memory access. With this information the code was recompiled and enhanced with code instrumentation for dynamic permission

allocation for heap accesses and a static permission list for code, stack and static data accesses.

Both library code (glibc) and benchmark code was instrumented this way. Since SimpleScalar is not a full system simulator, the benchmarks are statically compiled and system calls are not part of the simulations. A second round of simulations provides an accurate measurement of the performance impact caused by the execution of extra instructions and memory accesses for loading and manipulating permission metadata.

The overall performance penalty depends on three factors: (i) extra memory accesses for fetching memory permission information, (ii) execution of dynamic permission delegation and saving the generated permission lists between function calls, (iii) instruction cache locality because of the slightly larger code size. Since the permissions lookup is not on the critical path so does not directly impacts the memory access time.

Since using either the data or instruction cache for permission lists would create another cause of performance degradation and locality problems, we used a separate cache for memory permissions and evaluated the permission caching size impact on the overall performance. This approach allows us to analyze the optimal caching requirements for the permission loading, without affecting the benchmarks caching behavior extensively (the code cache locality is affected because of the dynamic memory instrumentation). For the simulation of the hardware platform we used the characteristics of an ARM processor running at 400Mhz. The external bus and main memory are assumed to run at 100 MHz. The performance of our architecture was observed for a memory hierarchy that contains one level separate instruction and data caches. For the baseline system the instruction cache has 32Kb of 32-way associative memory and the data cache is a 32Kb, 64-way associative cache. The data and instruction cache sizes for the baseline results were matched to the cache sizes of the

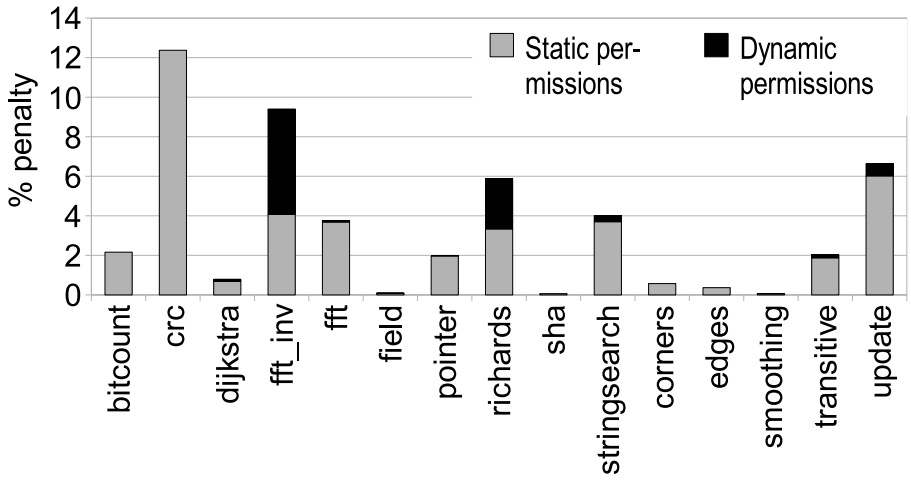

Figure 5-2: Performance Overhead with 16 KB Permission Cache.

experiments. We chose the embedded environment because of the ubiquity of downloaded apps in mobile devices. To the baseline, we added our hardware along the memory hierarchy for checking and managing permissions, including extra cache space for permissions. We also varied the cache sizes in our experiments, to observe the effects of such architectural parameters.

Figure 5-2 shows the percentage overhead for all benchmarks, with a 16 KB permissions cache. The chart also shows a breakdown of the overhead into static and dynamic components. The results are not surprising, since some benchmarks make greater use of dynamic memory than others. With a maximum of 12.37% and an average of only 3.5%, the overhead is reasonable for a large class of applications.

Figure 5-3 shows how performance depends on the size of the permission cache, as it is sized from 16 KB down to 2 KB. Most benchmarks fare well with a permission cache size above 2 KB, although some benchmarks are a little more sensitive than others. Overhead decreases rapidly as the permission cache size is increased relative to the data/instruction cache size. However, the value of added permission cache tapers off quickly after 4 KB, suggesting that permissions exhibit considerable locality and so only a modest-sized cache

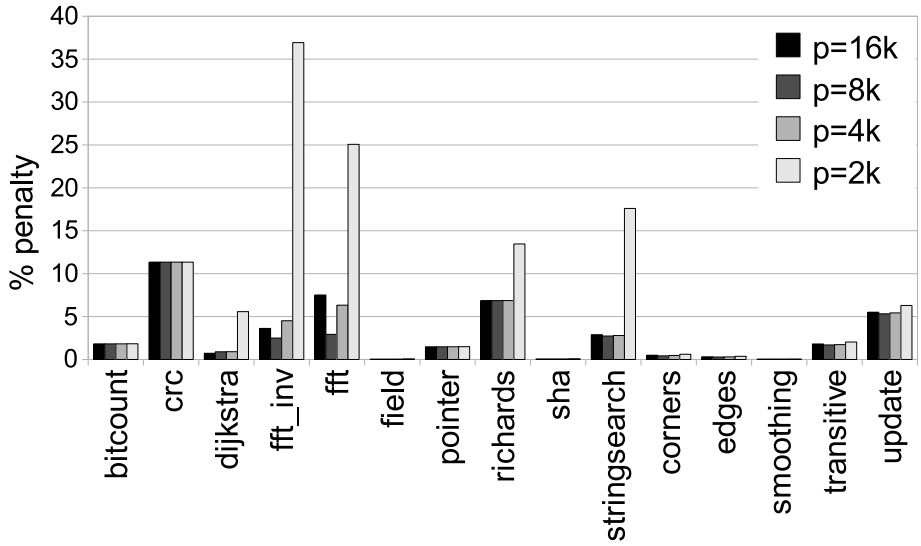

Figure 5-3: Performance Overhead with Varying Permission Cache Size, Fixed 16 KB Instruction And Data Cache Sizes.

is needed.

Figure 5-4 shows the performance results for a set of cache configurations that shows the 2 to 1 ratio of data and instruction cache to permission cache. Note that, for each set of results, the baseline is the unprotected system with the same data and instruction cache as in the result set. These results show that in resource constrained devices, where caching is limited, a good performance standard can be achieved with a relative small amount of permission cache (only one fifth of the total processor L1 cache space). As the permission cache is lowered to less than 2 KB, the cache pressure from fetching the static permissions heavily degrades performance. Also the performance overhead from executing the extra instructions, thus using relatively more of the shrinking instruction cache space, for dynamic permission delegation becomes more substantial than the pressure of a relatively smaller permission cache. The cost of the added instructions from dynamic permissions is especially visible in the Richards benchmark, and also can be observed in Figure 5-5, where we explore the overhead of a small permission cache of 2 KB with larger instruction and data caches. In this case, the Richards benchmark indicates

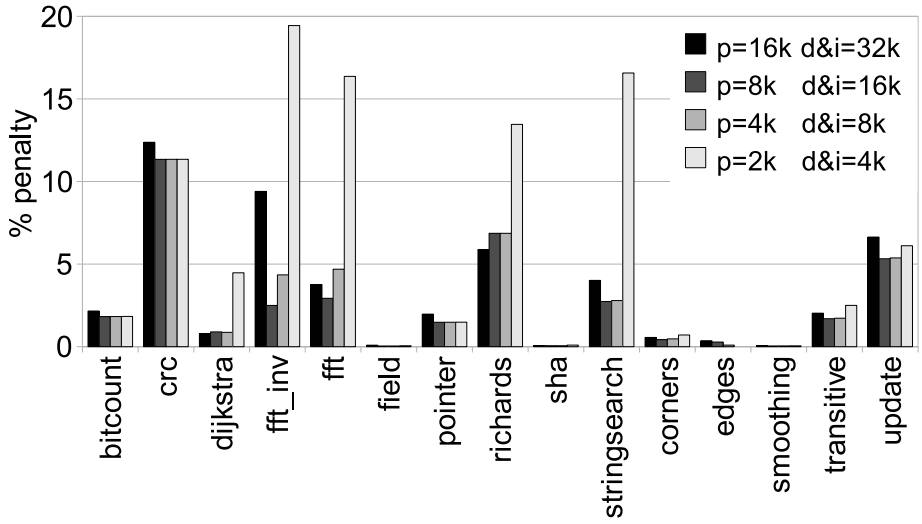

Figure 5-4: Performance Overhead with 2:1 Ratio of Data and Instruction Cache Size to Permission Cache Size

anomalous behavior because dynamic memory allocations dominate the overhead of loading the static permissions.

Figure 5-6 shows how trace driven container grouping can be used to improve performance. For these results, we used a very simple grouping algorithm: The most frequently "connected" functions are included in the same container, and we force each container to hold the same number of functions. The number of functions per container is varied from 1 – the same as in Figure 5-2 – to 5. The average performance penalties for grouping 2, 3, 4, and 5 functions per container were 1.7%, 1.87%, 1.78%, and 1.94% respectively. Note that performance is not always improved by increasing the number of functions per container. As the number of functions per container increases, the cost to load the static permission table also increases. If only a few of the functions in the container are called consecutively, the unused functions' permissions are needlessly prefetched. Thus, the cost of fetching all permissions may outweigh the benefit of avoiding the container switch, in certain cases. This result means there is room for future work in fine-tuning the algorithm that groups functions in to containers.

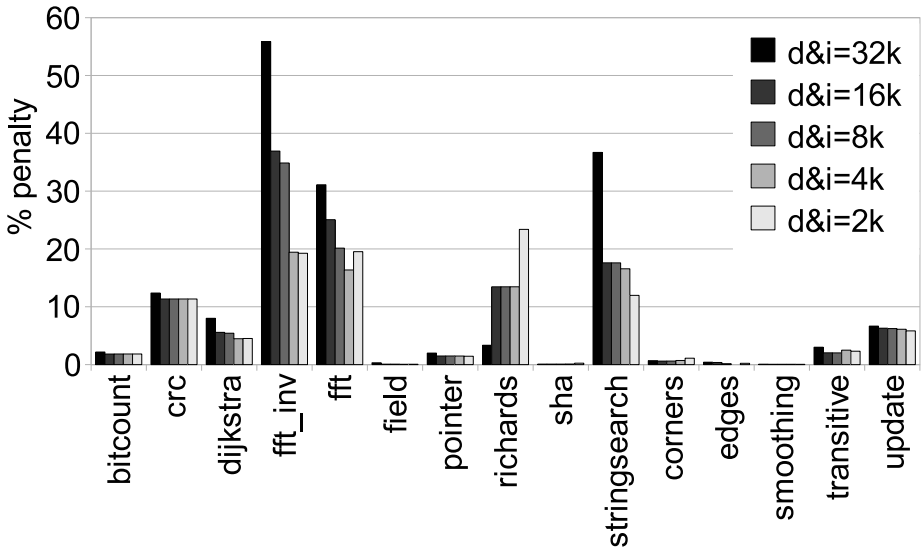

Figure 5-5: Overhead with a Limited (2 KB) Permission Cache

5.2 Advanced Experiments

We implemented the CM on top of a modern processor architecture based on the UltraSPARC III architecture. The UltraSPARC III represents an iteration of a long line of RISC processor designs, and it is equipped with state-of-the-art architectural features. GEMS/Opal [61] provides a cycle accurate micro-architecture simulator and cache model for this architecture. GEMS relies on WindRiver Simics [59] for device models and functional coherency when used for full system simulations. We implemented extensions to GEMS and plugins for Simics to emulate the hardware features of our reference monitor. Table 5-1 summarizes some of the simulated processor parameters, which we chose to match typical system-on-chip and embedded platforms available as commercial products.

To demonstrate the feasibility of the fine-grained memory access control in a complex software environment we chose RTEMS [78] as a suitable OS. We developed a Board Support Package (BSP) for the UltraSPARC III (and OpenSparc Niagara) and contributed it as the first 64-bit target port for RTEMS; it is now part of the upstream RTEMS distribution. RTEMS is

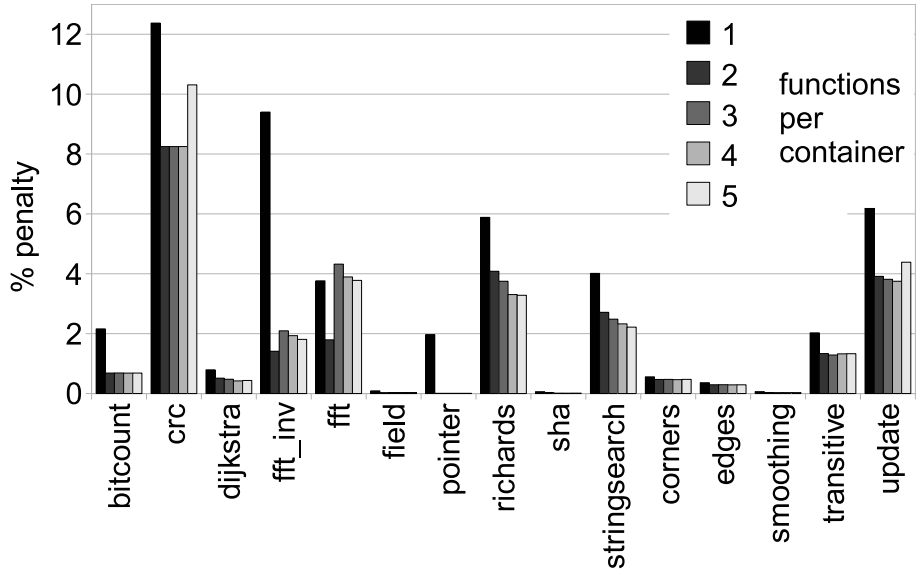

Figure 5-6: Overhead with Multiple Functions per Container, 32 KB Data and Instruction Cache, 16 KB Permission Cache

Name	Value	Name	Value
Cores	1	Inst Window Size	32
		Branch Predictor	YAGS
L1 Dcache	8KB 4xassoc	Reg Window Sets	8
L1 ICache	16KB 4xassoc	L2 Latency	23 cycles
L2 cache	3MB 12xassoc	DRam Latency	80 cycles

Table 5-1: Simulated Architectural Parameters

POSIX-compliant and offers support for custom task extensions including a context switch call-out, which we utilized to implement the container context switch.

We evaluate the performance of our solution with experiments using benchmark applications from MiBench [32], the Data Intensive Systems (DIS) benchmark suite [60], a reduced size Dhrystone test, and the heap-intensive Richards benchmark [115]. We also designed a synthetic microbenchmark that can vary container-related code features such as the rate of function calls, the ratio of dynamic to static memory accesses, the ratio of CPU-bound to memory-bound code, the layout of function bodies, and the size of permission lists.

Each benchmark was executed on the baseline platform, and then with the modified architecture with varying architectural parameters and optimizations. We compute performance overhead as the change in the number of pipeline cycles when benchmarks execute with and without fine-grained memory protection. Graphical results are presented as the percent overhead compared with the baseline execution time, so a lower percentage represents better performance.

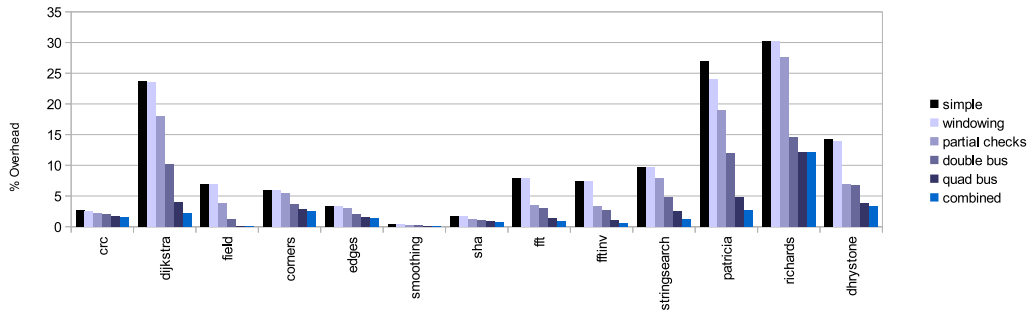

Figure 5-7: Single-threaded performance. Performance overhead for each benchmark using containers without optimization (simple) and with each optimization applied: ideal container windowing (windowing), partial container checking (partial checks), doubling the CM-cache bus (double bus), quadrupling the CM-cache bus (quad bus), and the combination of windowing, partial checks, and quad bus (combined).

Single-Threaded Performance. Our first set of experiments test the performance impact of using containers to protect single-threaded applications and the effectiveness of the optimizations described in section 4. For these experiments we modified the benchmarks, RTEMS system calls, and support libraries to use fine-grained memory protection with containers at the granularity of one function per container. Interrupts were configured to run in the security context of the application, so that interrupt handlers can execute at any time without causing a protection violation.

Figure 5-7 shows the performance as the percent difference in execution time measured in processor cycles between our design and the unmodified architecture. The Richards benchmark, which has a high function call frequency and uses a lot of dynamically-allocated sparse data structures, exhibits the worst performance overhead at 12.17%.

Multithreaded Performance. For our next set of experiments we concentrate on the added task context switch overhead due to containers. In these experiments we constructed a multithreaded program by placing individual benchmarks from crc, susan (corners and edges), and dhrystone in their own RTEMS task. Figure 5-8 shows the performance overhead of this program with preemptive round-robin scheduling with preemption frequency increasing from 0 (no preemption; tasks run sequentially) to 1 KHz. On context switches we execute a custom RTEMS task extension. We save the security context with the task context using a single CTXSWITCH instruction. Not accounting for the overhead of executing the task extensions, the overhead for using containers grows slowly in relation to the preemption frequency.

Function Call Frequency. When each container holds a single function every function call triggers a security context switch, so the frequency of function calls affects performance.

Figure 5-9 shows the performance degradation as the frequency of function

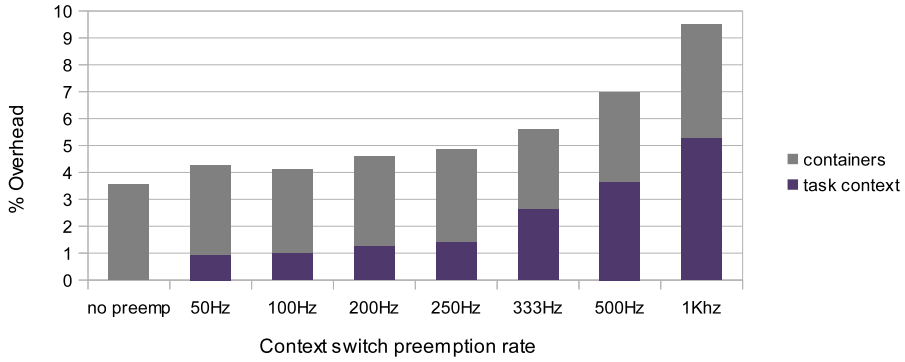

Figure 5-8: Multithreaded performance. Overhead of context switch time respective to whole program execution time when running crc, susan (corners and edges), and dhrystone in separate tasks. We include the overhead from the usual task context switching (task context) above which remains the overhead caused by using containers for fine-grained memory protection (containers).

calls/returns increases. Each function references both dynamic- and static-allocated memory. We used our synthetic microbenchmark for this set of benchmarks, and tuned the dynamic memory (execution of ALLOW instructions) to account for roughly 10% of the total overhead.

Sparse Data Structures. When data cells are not adjacent in memory, each cell needs a separate memory permission entry. In the worst case, passing a reference to such a data structure to another principal for processing requires traversing the entire data structure for permissions assignment. To support such behavior a program can maintain a permission table associated with a data structure, and update the table when adding/removing elements to the data structure. The ALLOWM instruction will pass permissions by taking as argument the number of memory ranges and the location of the permission table.

Figure 5-10 illustrates how performance degrades as permission lists for each function invocation increase in length. While maintaining a constant arithmetic and memory workload for each function call, we increase the size

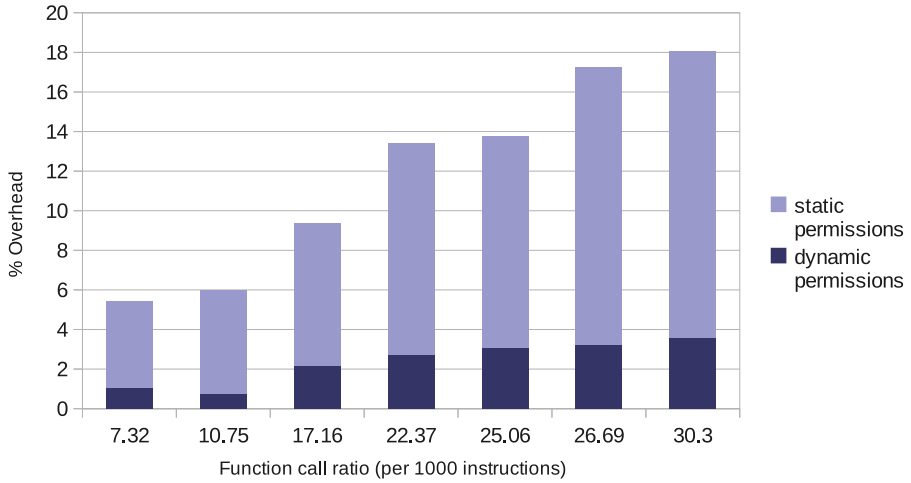

Figure 5-9: Function Call Frequency. Performance overhead as the number of function calls (per 1000 instructions) increases.

of the access lists for both a static and dynamic sparse data structure. Using individual ALLOW calls for each data cell has a significant performance overhead as the processor is busy executing the extra instructions for each function call. Replacing the ALLOW statements with a single ALLOWM to the precomputed permission table reduces the overhead for dynamic memory closer to that of statically derived permissions.

Summary. We have demonstrated through a series of experiments that fine-grained memory protection is practical and obtainable. Architectural optimizations and careful design and integration of the CM with the processor pipeline gives single-threaded performance a modest penalty for the improved security that containers can provide. Multithreaded benchmarks do not exhibit penalties much larger than those of single-threaded benchmarks because the extra work of managing the CM during a task context switch is small; the dominant overhead becomes the task context switch time as the frequency of context switching increases. Containers are an effective solution for modern hardware and concurrent applications.

Our C++ benchmark behavior mimic processor and memory workloads

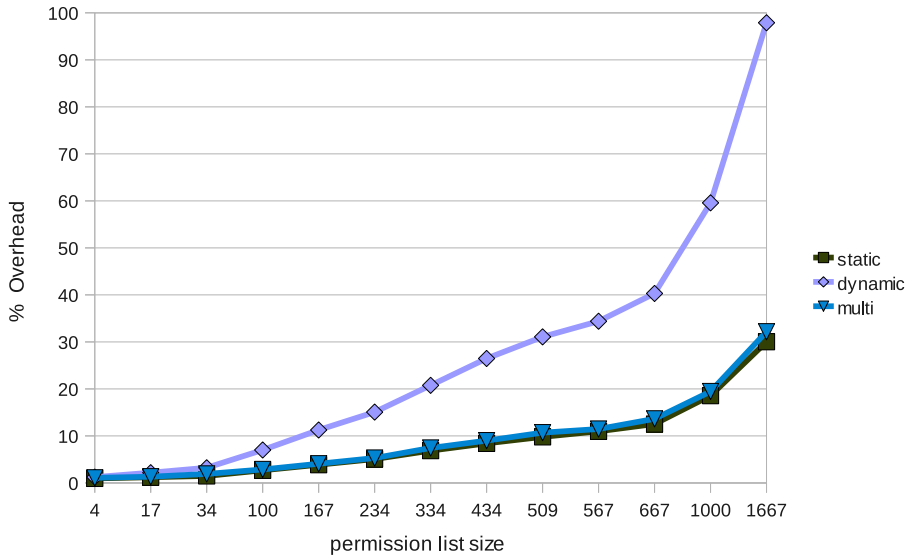

Figure 5-10: Sparse Memory Access and Allow Multiple (ALLOWM). The overhead of our solution increases with the number of memory ranges that get passed between containers. Two extremes are when every permission is determined at runtime with the ALLOW statement (dynamic) and when permissions are determined at compile-time and put in a list (static). ALLOWM uses a table storing permissions for memory ranges that can be passed to a function dynamically. We show the performance when the ALLOWM permissions table is created at compile-time (multi), which shows that ALLOWM can be nearly as efficient as using a static permissions list.

exhibited by C tests and in addition focus on object creation, access of private/public members and destruction, the subject of this research. For each test we first ran the benchmarks with no memory protection and no code instrumentation. We use this results as our baseline. The second run has the CM enabled in the hardware simulator and the static permissions are loaded with the application. This allows us to separately evaluate the impact of static permissions vs the overhead created by the extra instructions and CM workload from the dynamic permissions. The third run has full protection enabled. Graphical results are presented as the percent overhead compared with the baseline execution time, so a lower percentage represents better performance.

The first two sets of experiments Figure 5-12 and Figure **??** exercise the creation and destruction of C++ objects on the heap vs on the stack. All tests

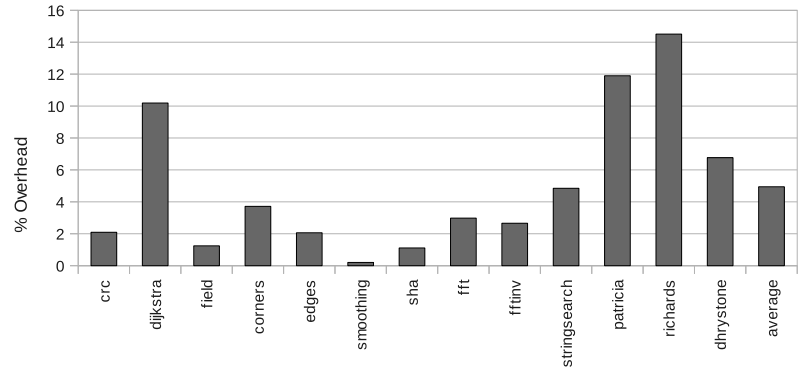

Figure 5-11: Embedded systems benchmarks: Performance Overhead

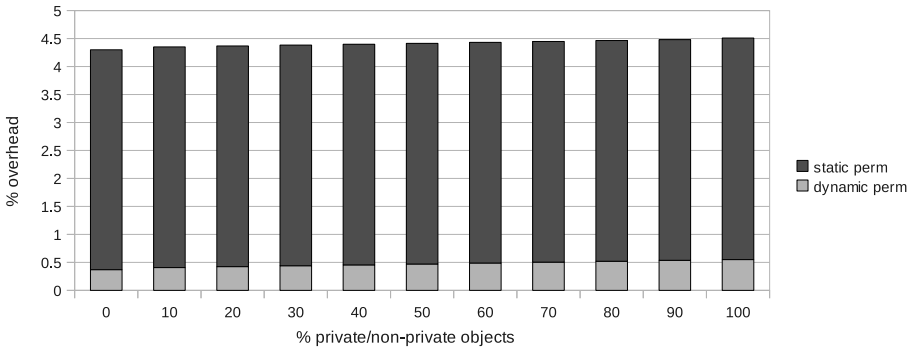

Figure 5-12: Objects creation: heap

have only two types of classes: the first type has only public accessible methods and fields, the second type has half public, half private fields, thus requiring the special handling of permissions between the parent function, the constructor and destructor. We vary the ratio of one type of objects from all public to all private type. Due to the higher overhead of dynamic memory allocation, the heap only tests show little difference between the types of objects being used the difference between all public and public/private extremes being only 0.2%. The difference becomes clear with stack objects. While the static permissions stay constant as expected in both cases, on the stack tests, the results show a much higher penalty when using fine grained memory protection reaching as high as 11.68% when all objects have private members.

Inheritance and encapsulation are powerful paradigms of object oriented

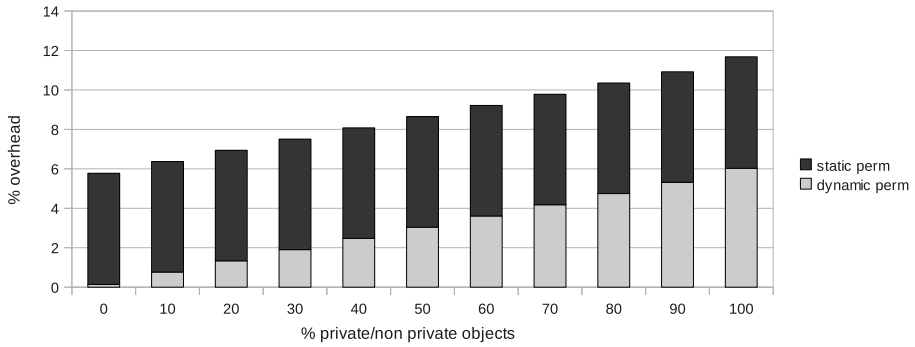

Figure 5-13: Objects creation: stack

languages in general. Even when intricate multi-level class hierarchies are used, the compiler generates one compact data structure with all members stacked based on the hierarchy. This approach optimizes locality and for static casts simple and fast relative offsets locate the start address of various members of different classes. In case of enforcing private permissions, both inheritance and encapsulation create a fragmentation of permission maps for the object. Although the C++ specification does not enforce or recommends any strict memory layout for objects, thus allowing us to potentially regroup object members to alleviate this problem, doing so would break C++ ABIs and would break interoperability with traditional systems. Figure 5-14 shows the performance overhead when we increase the fragmentation level of permissions for private members within objects either through encapsulation or inheritance.

The last set of tests in Figure 5-15, depicts how access functions to private members affect overall memory protection overhead. With no member function called, the overhead is mostly dominated by static permissions. As the number of calls to member function increases for each object instance, the dynamic permission overhead starts to dominate. When using heap allocated objects, using either member functions to process a private or public field would have the same performance impact as the same number of calls to pass along permissions between containers is needed.

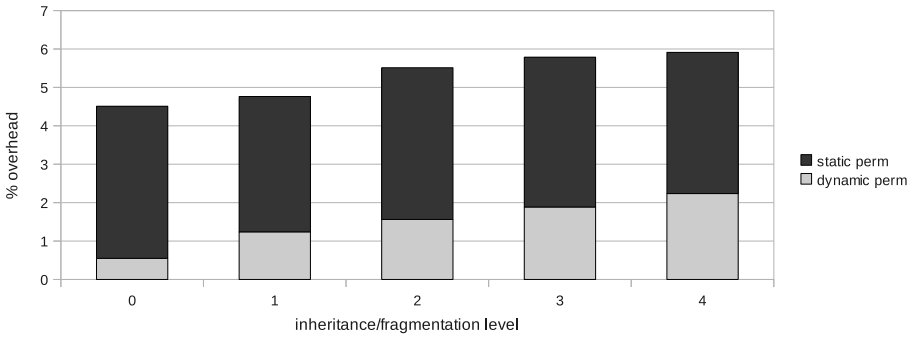

Figure 5-14: Fragmentation/Inheritance/Encapsulation

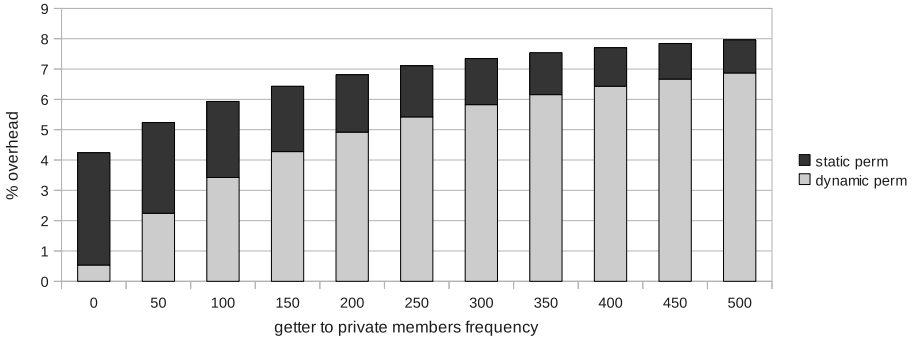

Figure 5-15: Frequency of unoptimized getters to private fields

Chapter 6 – Conclusions

This work describes a solution for isolation of untrusted code at a function level, combining compiler techniques with additional hardware, to provide seamless integration of trusted and untrusted code. By using the stack-like nature of function calls, we can protect execution at a finer execution granularity than existing solutions that rely on the OS to manage permission tables at a process or thread level. We protect memory at a finer spacial granularity than page-level, allowing permissions to be specified for individual words. The fine granularity of protection allows for low overhead protection of code without requiring programmers to hand-code page alignments or to separate trusted and untrusted code into separate processes. Our solution approach includes mechanisms for simultaneously handling dynamic memory, control-flow monitoring, and detecting denial-of-service, all with modest overhead.

The fine granularity of protection allows for memory access control without requiring programmers to hand-code page alignments or use inter-process communication when changing principals or protection domains. Our solution includes protection mechanisms for handling dynamic memory and concurrent applications with modest overhead.

We have also demonstrated that hardware support for fine-grained memory access control can be applied to OOP languages like C++ and provides security benefits to the language such as enforcing private access specifiers and reducing the attack surface of the exception handling code. We described how features of C++ can be handled by our simple containers hardware, and we evaluated the overhead of our protection mechanism for C++ object creation/destruction and accessor functions. Containers can support arbitrary permissions on class fields that need not be tied to access specifiers, or that can be selectively enforced on a per-class basis.

Published research

- Eugen Leontie, Gedare Bloom, Bhagirath Narahari and Rahul Simha Enforcing Encapsulation for C++. In preparation

- Eugen Leontie, Gedare Bloom, Bhagirath Narahari and Rahul Simha. No Principal Too Small: Memory Access Control for Fine-Grained Protection Domains. 15th Euromicro Conference on Digital System Design(DSD), Sept 2012, Ismir, Turkey [48]

- Eugen Leontie, Gedare Bloom, and Rahul Simha. Automation for Creating and Configuring Security Manifests for Hardware Containers. 4th Symposium on Configuration Analytics and Automation (SafeConfig), November 2011, Arlington, VA [51]

- Stefan Popoveniuc, John Kelsey, Eugen Leontie. On the privacy threats of electronic poll books. International Conference on Security and Cryptography SECRYPT 2011, Seville, Spain 2011 [72]

- Gedare Bloom, Eugen Leontie, Bhagirath Narahari, Rahul Simha. Hardware and Security: Vulnerabilities and Solutions, in Handbook on Securing Cyber-Physical Critical Infrastructure Foundations and Challenges, Elsevier, ISBN-13:978-0124158153 [52]

- Eugen Leontie, Olga Gelbart, Bhagirath Narahari, Rahul Simha. Detecting Memory Spoofing in Secure Embedded Systems using Cache-Aware FPGA Guards. Sixth International Conference on Information Assurance and Security, IAS2010 [54]

- Stefan Popoveniuc, Eugen Leontie. SAFE RPC - Auditing mixnets safely using Randomized Partial Checking, International Conference on Security and Cryptography SECRYPT 2010, Athens, Greece 2010 [73]

- Eugen Leontie, Gedare Bloom, Bhagirath Narahari, Rahul Simha, and Joseph Zambreno. Hardware containers for software components: A trusted platform for COTS-based systems. In Proceedings of The 2009 International Symposium on Trusted Computing, TrustCom09. IEEE, August 2009. [49]

- Eugen Leontie, Gedare Bloom, Bhagirath Narahari, Rahul Simha, and Joseph Zambreno. Hardware-enforced fine-grained isolation of untrusted code. In Proceedings of the 2009 ACM workshop on Secure Execution of Untrusted Code, SecuCode09. ACM, Nov 2009.[50]

- Eugen Leontie, Gedare Bloom, Olga Gelbart, Bhagirath Narahari, and Rahul Simha. A compiler-hardware technique for protecting against buffer overflow attacks. Journal of Information Assurance and Security, 2009.[46]

- Eugen Leontie, Olga Gelbart, Bhagirath Narahari, and Rahul Simha. Compiler-FPGA technique to detect memory spoofing in encrypted-execution platforms. 6th Annual Security Conference, April 2007.[53]

- Olga Gelbart, Eugen Leontie, Bhagirath Narahari, and Rahul Simha. A compiler-hardware approach to software protection for embedded systems. Cir- cuits and Systems for Real-Time Security and Copyright Protection of Multi- media, pages 315 - 328, March 2009.[29]

- Olga Gelbart, Eugen Leontie, Bhagirath Narahari, and Rahul Simha. Architectural support for securing application data in embedded systems. IEEE International Conference on Electro/Information Technology, p19-24, May 2008.[28]

- A Simple Compiler-FPGA Technique to Detect Memory Spoofing in

Encrypted-Execution Platforms, Eugen Leontie, O. Gelbart, B. Narahari, R. Simha, 6th Annual Security Conference, April 2007. [55]

Bibliography

[1] Apple itouch / iphone tiff image handling privilege escalation. *http://osvdb.org/displayvuln.php?osvdb_id=38527*.

[2] Sony psp photo viewer tiff file overflow. *http://osvdb.org/displayvuln.php?osvdb_id=19665*.

[3] Martn Abadi, Mihai Budiu, lfar Erlingsson, and Jay Ligatti. Control-flow integrity. *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340 – 353, 2005.

[4] Mehdi-Laurent Akkar and Christophe Giraud. An implementation of DES and AES secure against some attacks. *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 309–318, 2001.

[5] Divya Arora, Anand Raghunathan, Srivaths Ravi, and Niraj K. Jha. Architectural support for run-time validation of program data properties. *IEEE Transactions on very large scale integration(VLSI) systems, VOL. 15, NO. 5*, May 2007.

[6] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Hardware-assisted run-time monitoring for secure program execution on embedded processors. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems, Vol. 14, No.12*, Dec 2006.

[7] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, Feb 2002.

[8] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 115–126, Washington, DC, USA, 2008. IEEE Computer Society.

[9] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems mathematical foundations. *MITRE Corporation*, 1973.

[10] David Brash. The ARM architecture version 6. *ARM Whitepaper available at www.arm.com*, Jan 2002.

[11] The Mozilla Firefox browser. http://www.mozilla.org/firefox.

[12] Mihai Budiu, Úlfar Erlingsson, and Martín Abadi. Architectural support for software-based protection. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 42–51, New York, NY, USA, 2006. ACM.

[13] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147 – 160, 2006.

[14] Tzi cker Chiueh and Fu hau Hsu. Rad: A compile-time solution to buffer overflow attacks. *21st IEEE International Conference on Distributed Computing Systems (ICDCS'01)*, 2001.

[15] Ellis Cohen and David Jefferson. Protection in the hydra operating system. *ACM SIGOPS Operating Systems Review*, pages 141 – 160, 1975.

[16] Marc L. Corliss, E Christopher Lewis, and Amir Roth. Using DISE to protect return addresses from attack. *Workshop on Architectural Support for Security and Anti-Virus*, 2004.

[17] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, pages 227–237, 2003.

[18] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard tm : Protecting pointers from buffer overflow vulnerabilities. *In Proc. of the 12th Usenix Security Symposium*, 2003.

[19] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. *USENIX Security Symposium*, 1998.

[20] Jedidiah R. Crandall, S. Felix Wu, and Frederic T. Chong. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim.*, 3(4):359–389, 2006.

[21] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A flexible information flow architecture for software security. *International Symposium on Computer Architecture*, pages 482 – 493, June 2007.

[22] Prattana Deeprasertkul, Pattarasinee Bhattarakosol, and Fergus O'Brien. Automatic detection and correction of programming faults for software applications. *J. Syst. Softw.*, 78(2):101–110, 2005.

[23] H. M. Deitel and Michael S. Kogan. *The design of OS/2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1992.

[24] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, pages 143 – 155, 1966.

[25] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 566–577, New York, NY, USA, 2009. ACM.

[26] ProPolice Eto Hiroaki, Yoda Kunikazu. Propolice. *Transactions of Information Processing Society of Japan*, pages 4034–4041, 2002.

[27] Christof Fetzer and Zhen Xiao. Detecting heap smashing attacks through fault containment wrappers. *In Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, Oct 2001.

[28] Olga Gelbart, Eugen Leontie, Bhagirath Narahari, and Rahul Simha. Architectural support for securing application data in embedded systems. *IEEE International Conference on Electro/Information Technology*, pages 19 – 24, May 2008.

[29] Olga Gelbart, Eugen Leontie, Bhagirath Narahari, and Rahul Simha. A compiler-hardware approach to software protection for embedded systems. *Circuits and Systems for Real-Time Security and Copyright Protection of Multimedia*, pages 315 – 328, March 2009.

[30] Olga Gelbart, Paul Ott, Bhagirath Narahari, Raul Simha, Alok Choudhary, and Joseph Zambreno. CODESEAL: Compiler/FPGA approach to secure applications. *Proc. IEEE, ISI*, 2005.

[31] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the OP web browser. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 402–416, 2008.

[32] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[33] Brian Hay and Kara Nance. Forensics examination of volatile system data using virtual introspection. *SIGOPS Oper. Syst. Rev.*, 42(3):74–82, 2008.

[34] Ted Huffmire, Shreyas Prasad, Tim Sherwood, and Ryan Kastner. Policy-driven memory protection for reconfigurable hardware. *Lecture Notes in Computer Science*, pages 461 – 478, 2006.

[35] G. Hunt, C. Hawblitzel, O. Hodson, J. Larus, B. Steensgaard, and T. Wobber. Sealing OS processes to improve dependability and safety. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, March 2007.

[36] G. Hunt and J. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, April 2007.

[37] Koji Inoue. Energy-security tradeoff in a secure cache architecture against buffer overflow attacks. *SIGARCH Comput. Archit. News*, 33(1):81–89, 2005.

[38] Trevor Jim, Greg Morrisett, Dan Grossman, Mike Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. *Usenix Annual Technical Conference, pages 275-288, Monterey, CA*, Jun 2002.

[39] José A. Joao, Onur Mutlu, and Yale N. Patt. Flexible reference-counting-based hardware acceleration for garbage collection. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 418–428, New York, NY, USA, 2009. ACM.

[40] Wen-Fu Kao and S. Felix Wu. Lightweight hardware return address and stack frame tracking to prevent function return address attack. In *CSE '09: Proceedings of the 2009 International Conference on Computational Science and Engineering*, pages 859–866, Washington, DC, USA, 2009. IEEE Computer Society.

[41] Mazen Kharbutli, Xiaowei Jiang, Yan Solihin, Guru Venkataramani, and Milos Prvulovic. Comprehensively and efficiently protecting the heap. *ACM SIGOPS Operating Systems Review, Proceedings of the 2006 ASPLOS Conference*, pages 207 – 218, December 2006.

[42] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327 – 353, 2001.

[43] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. *Proceedings of Advances in Cryptology (Crypto)*, pages 388–397, 1999.

[44] Benjamin A. Kuperman, Carla E. Brodley, Hilmi Ozdoganoglu, T. N. Vijaykumar, and Ankit Jalote. Detection and prevention of stack buffer overflow attacks. *Communications of the ACM*, pages 50–56, Nov 2005.

[45] Ruby B. Lee, David K. Karig, John P. Mcgregor, and Zhijie Shi. Enlisting hardware architecture to thwart malicious code injection. In *In Proceedings of the 2003 International Conference on Security in Pervasive Computing*, pages 237–252. Springer Verlag, 2003.

[46] Eugen Leontie, Gedare Bloom, Olga Gelbart, Bhagirath Narahari, and Rahul Simha. A compiler-hardware technique for protecting against buffer overflow attacks. *Journal of Information Assurance and Security*, 2009.

[47] Eugen Leontie, Gedare Bloom, Bhagirath Narahari, and Rahul Simha. C++: "hide" or "protect" the implementation ? *in preparation*, 2012.

[48] Eugen Leontie, Gedare Bloom, Bhagirath Narahari, and Rahul Simha. No principal too small: Memory access control for fine-grained protection domains. *15th Euromicro Conference on Digital System Design(DSD)*, September 2012.

[49] Eugen Leontie, Gedare Bloom, Bhagirath Narahari, Rahul Simha, and Joseph Zambreno. Hardware containers for software components: A trusted platform for COTS-based systems. In *Proceedings of The 2009 International Symposium on Trusted Computing, TrustCom09*. IEEE, August 2009.

[50] Eugen Leontie, Gedare Bloom, Bhagirath Narahari, Rahul Simha, and Joseph Zambreno. Hardware-enforced fine-grained isolation of untrusted code. In *Proceedings of the 2009 ACM workshop on Secure Execution of Untrusted Code, SecuCode09*. ACM, Nov 2009.

[51] Eugen Leontie, Gedare Bloom, and Rahul Simha. Automation for creating and configuring security manifests for hardware containers. *4th Symposium on Configuration Analytics and Automation (SafeConfig)*, 2011.

[52] Eugen Leontie, Gedare Bloom, and Rahul Simha. Hardware and security: Vulnerabilities and solutions, 2011.

[53] Eugen Leontie, Olga Gelbart, Bhagirath Narahari, and Rahul Simha. Compiler-FPGA technique to detect memory spoofing in encrypted-execution platforms. *6th Annual Security Conference*, April 2007.

[54] Eugen Leontie, Olga Gelbart, Bhagirath Narahari, and Rahul Simha. Detecting memory spoofing in secure embedded systems using cache-aware fpga guards. In *IAS*, pages 125–130, 2010.

[55] Eugen Leontie, Olga Gelbart, Bhagirath Narahari, and Rahul Simha. A simple compiler-fpga technique to detect memory spoofing in encrypted execution platforms. *6th Annual Security Conf, Las Vegas*, March 2007.

[56] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[57] Peter Loscocco, Stephen Smalley, Patrick Muckelbauer, Ruth Taylor, S. Jeff Turner, and John Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, 1998.

[58] Bharat B. Madan, Shashi Phoha, and Kishor S. Trivedi. Stackoffence: A technique for defending against buffer overflow attacks. In *ITCC '05:*

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I, pages 656–661, Washington, DC, USA, 2005. IEEE Computer Society.

[59] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35:50–58, February 2002.

[60] J.W. Manke and J. Wu. Data-intensive system benchmark suite analysis and specification. *Atlantic Aerospace Electronics Corp*, 1999.

[61] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[62] R.K. Montoye. Apparatus for storing dont care in a content addressable memory cell. *United States Patent no5,319,590, 6/94*, 1994.

[63] G. J. Myers and B. R. S. Buckingham. A hardware implementation of capability-based addressing. *ACM SIGOPS Operating Systems Review, Volume 14 , Issue 4*, pages 13 – 25, 1980.

[64] R. M. Needham and R. D.H. Walker. The cambridge cap computer and its protection system. *ACM SIGOPS Operating Systems Review*, pages 1 – 10, 1977.

[65] Designing Secure Systems on Reconfigurable Hardware. Ted huffmire and brett brotherton and nick callegari and jonathan valamehr and jeff white and ryan kastner and tim sherwood. *ACM Transactions on Design Automation of Electronic Systems*, July 2008.

[66] Siddika Berna Ors, Frank Gurkaynak, Elisabeth Oswald, and Bart Preneel. Power-analysis attack on an ASIC AES implementation. *Proceedings of the International Conference on Information Technology (ITCC)*, 2004.

[67] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. *Topics in Cryptology CT-RSA 2006, The cryptographers' track at the RSA conference 2006*, pages 1 – 20, February 2006.

[68] Hilmi Ozdoganoglu, T. N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. SmashGuard: A hardware solution to prevent attacks on the function return address. *IEEE Transactions on Computers*, 2006.

[69] Kostas Pagiamtzis and Ali Sheikholeslami. Content-addressable memory (cam) circuits and architectures: a tutorial and survey. *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, pages 712 – 727, 2006.

[70] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kcman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery oriented computing (roc):motivation, definition, techniques, and case studies. *UC Berkeley Computer Science Technical Report UCB//CSD-02-1175*, March 2002.

[71] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2004.

[72] Stefan Popoveniuc, John Kelsey, and Eugen Leontie. On the privacy threats of electronic poll books. In *SECRYPT*, pages 281–286, 2011.

[73] Stefan Popoveniuc and Eugen Leontie. Safe rpc - auditing mixnets safely using randomized partial checking. *International Conference on Security and Cryptography SECRYPT*, 2010.

[74] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 135 – 148, 2006.

[75] Nguyen Anh Quynh and Yoshiyasu Takefuji. A novel approach for a file-system integrity monitor tool of xen virtual machine. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 194–202, New York, NY, USA, 2007. ACM.

[76] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Multi-aspect profiling of kernel rootkit behavior. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 47–60, New York, NY, USA, 2009. ACM.

[77] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, New York, NY, USA, 2009. ACM.

[78] RTEMS. http://www.rtems.com.

[79] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer system. *Proceedings of the IEEE*, pages 1278–1308, Sept 1975.

[80] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE, Volume: 63, Issue: 9*, pages 1278 – 1308, 1975.

[81] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *In Proceedings of the 10th USENIX Security Symposium*, pages 201–220, 2001.

[82] Zili Shao, Jiannong Cao, Keith C. C. Chan, Chun Xue, and Edwin H.-M. Sha. Hardware/software optimization for array & pointer boundary checking against buffer overflow attacks. *J. Parallel Distrib. Comput.*, 66(9):1129–1136, 2006.

[83] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-vm monitoring using hardware virtualization. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 477–487, New York, NY, USA, 2009. ACM.

[84] Jianli Shen, Guru Venkataramani, and Milos Prvulovic. Tradeoffs in fine-grained heap memory protection. *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 52 – 57, 2006.

[85] Weidong Shi, Joshua B. Fryman, Guofei Gu, Hsien Hsin S. Lee, Youtao Zhang, and Jun Yang. Infoshield: a security architecture for protecting information usage in memory. *The Twelfth International Symposium on High-Performance Computer Architecture*, pages 222–231, 2006.

[86] Weidong Shi and Hsien-Hsin Sean Lee. Authentication control point and its implications for secure processor design. *39th Annual IEEE/ACM International Symposium on Microarchitecture MICRO-39*, pages 103 – 112, 2006.

[87] Weidong Shi, Chenghuai Lu, and Hsien-Hsin S. Lee. Memory-centric security architecture. *High performance embedded architectures and compilers, Barcelona, Spain, November 17-18*, 2005.

[88] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. Bitvisor: a thin hypervisor for enforcing i/o device security. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, New York, NY, USA, 2009. ACM.

[89] Alexey Smirnov and Tzi cker Chiueh. Dira: Automatic detection, identification, and repair of control-hijacking attacks. In *In NDSS*, 2005.

[90] E. Spitznagel, D. Taylor, and J. Turner. Packet classification using extended tcams. *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, 2003.

[91] Francois-Xavier Standaert, S. Berna Ors, Jean-Jacques Quisquater, and Bart Preneel. Power analysis attacks against FPGA implementations of the DES. *Proceedings of the International Conference on Field-Programmable Logic and its Applications (FPL)*, page 8494, 2004.

[92] Bjarne Stroustrup. Multiple inheritance for c++. *The C/C++ Users Journal*, May 1999.

[93] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. *17 International Conference on Supercomputing (ICS)*, 2003.

[94] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM.

[95] G.Edward Suh, Jaewook Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *Architectural support for programming languages and operating systems*, pages 85 – 96, 2004.

[96] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *In OSDI*, pages 1–16, 2004.

[97] Michael M. Swift, Steven Martin, Henry M. Levy, , and Susan J. Eggers. Nooks: an architecture for reliable device drivers. *Proceedings of the Tenth ACM SIGOPS European Workshop*, 2002.

[98] D. E. Taylor. Survey & taxonomy of packet classification techniques. *Tech. Rep. WUCSE-2004-24*, 2004.

[99] Radu Teodorescu and Josep Torrellas. Prototyping architectural support for program rollback using fpgas. *Field-Programmable Custom Computing Machines*, pages 23 – 32, April 2005.

[100] The Apache Software Foundation. Apache httpd modules. http://httpd.apache.org/modules/, 2009.

[101] Kris Tiri and Ingrid Verbauwhede. A digital design flow for secure integrated circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1197–1208, July 2006.

[102] Mohit Tiwari, Hassan Wassel, Bita Mazloom, Shashidhar Mysore, Frederic Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2009.

[103] D. Turner. Symantec internet security threat report: Trends for January - June 2007. *Tech. report, Symantec Inc.*, 2007.

[104] Alfred V.Aho, Ravi Sethi, and Jeffrey D.Ullman. *Compilers - Principles, Techniques, and Tools,Addison-Wesley*. Addison-Wesley, 1988.

[105] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. *International Symposium on High Performance Computer Architecture*, pages 173 – 184, 2008.

[106] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. *High-Performance Computer Architecture, International Symposium on*, 0:273–284, 2007.

[107] J. Viega, J.T. Bloch, T. Kohno, and G. McGraw. Token-based scanning of source code for security problems. *ACM Transactions on Information and System Security*, pages 238 – 261, 2002.

[108] D. Wagner and H. Chen. Mops: An infrastructure for examining security properties of software. *Proceedings of the* 9^{th} *ACM Conference on Computer and Communications Security*, pages 235 – 244, 2002.

[109] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *In Network and Distributed System Security Symposium*, pages 3–17, 2000.

[110] Dan Wallach. Technical perspective native client: a clever alternative. *Commun. ACM*, 53(1):90–90, 2010.

[111] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 545–554, New York, NY, USA, 2009. ACM.

[112] Geoffrey Washburn and Stephanie Weirich. Good advice for type-directed programming aspect-oriented programming and extensible generic functions. *Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*, pages 33 – 44, 2006.

[113] Peter Wilson, Tom Mihm, Alexandre Frey, Danny Kershaw, and Tiago Alves. Implementing embedded security on dual-virtual-cpu systems. *Design and Test of ICs for Secure Embedded Computing*, pages 582–591, 2007.

[114] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian memory protection. *Proceedings of ASPLOS-X*, Oct 2002.

[115] Mario Wolczko. Benchmarking Java with Richards and DeltaBlue. available at http://research.sun.com/people/mario/java_benchmarking, 2006.

[116] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. *Foundations of Intrusion Tolerant Systems*, pages 213 – 226, 2003.

[117] J. Xu, Z. Kalbarczyk, S. Patel, and R.K. Iyer. Architecture support for defending against buffer overflow attacks. *Proceedings of the* 2^{nd} *Workshop Evaluating and Architecting System Dependability (EASY-2002)*, 2002.

[118] Pengyuan Yu and Patrick Schaumont. Secure FPGA circuits using controlled placement and routing. *International Conference on Hardware/Software Codesign and System Synthesis*, Oct 2007.

[119] Joseph Zambreno, Tanathil Anish, and Alok Choudhary. A run-time reconfigurable architecture for embedded program flow verification. *Proceedings of the NATO Advanced Research Workshop (ARW) on Security and Embedded Systems*, 2005.

[120] Joseph Zambreno, Alok Choudhary, Rahul Simha, Bhagi Narahari, and Nasir Memon. Safe-ops: An approach to embedded software security. *ACM Trans. Embed. Comput. Syst.*, 4(1):189–210, 2005.

[121] Kun Zhang, Tao Zhang, and Santosh Pande. Memory protection through dynamic access control. *The 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 123 – 134, 2006.

[122] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iwatcher: Efficient architectural support for software debugging. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 224, Washington, DC, USA, 2004. IEEE Computer Society.

[123] Xiaotong Zhuang, Tao Zhang, Hsien-Hsin S. Lee, and Santosh Pande. Hardware assisted control flow obfuscation for embedded processors. *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Sep 2004-b.

[124] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. HIDE: An infrastructure for efficiently protecting information leakage on the address bus. *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004-a.