No Principal Too Small: Memory Access Control for Fine-Grained Protection Domains

Abstract—**Modern programs comprise multiple threads of execution inside a single principal—the process—with a single protection domain, usually a page table. We propose a hardware-enforced, fine-grained memory protection mechanism to divide the process into smaller principals and multiple protection domains. Our approach supports modern software engineering better than traditional processes by enabling developers to align software components with protection mechanisms. We implemented our architecture using a cycle-accurate simulator of a complex out-of-order pipeline and evaluate our solution using open-source benchmarks and synthetic microbenchmarks designed specifically to stress our system.**

I. Introduction

Modern applications increasingly use untrusted third-party code to decrease development time and improve features. Software reuse has become such a widely accepted software engineering practice that experts estimate over 99% of executed instructions in US Department of Defense software come from commercial off-the-shelf components for which the vendor does not supply the source code [1]. Demand for features puts pressure on developers to allow user customization: Web servers like Apache [2] and browsers like Firefox [3] allow users to download third-party *plugins* that run directly as part of the application. Unfortunately software reuse and extensibility have led to safety and security problems. Symantec reports [4] at least 300 vulnerabilities for web browser plugins during 2010, and 500 in browsers themselves. The Ariane rocket crashed shortly after its first take-off causing the loss of about $500 million USD due to improper reuse of a software component [5]. A flaw in the software that handles tiff files for the iPhone and Sony PSP [6], [7] was exploited to allow users to run unlicensed applications that are not signed by the manufacturer. Despite such problems, economic pressure and user demands push software development toward components and plugins.

Application developers can use software isolation to prevent exploits from propagating beyond a vulnerable software component. Software isolation dates back to the first generation of time-sharing machines, from which Saltzer and Schroeder [8] defined a common terminology for memory protection. We adopt from their terminology the notions of *principal* and *domain*. A *principal* is the entity to which authorizations are granted. A protection *domain* is the set of objects (memory) that currently may be accessed by a principal. Modern hardware and OS mechanisms support the process as a principal and the virtual address space (page table) as a protection domain. Such mechanisms can be used to isolate software components: Chromium [9] is a web browser that renders each web page in a separate process from the main browser. But widespread adoption of process-based component isolation is discouraged by engineering costs, performance loss, and philosophical idealogy. (Open-source projects like GNU [10] encourage tight integration of dynamic plugins and base applications because of the copyleft philosophy of the GNU Public License.)

Fine-grained memory protection can divide an application into its constituent components so each can be given a smaller principal and protection domain that aligns with modular software design. In this paper we describe the design, implementation, and evaluation of a solution for fine-grained memory protection. We designed a hardware *reference monitor* that checks each memory access for compliance to a strict set of access control permissions; previous solutions for memory protection have shown that such checks can be efficient in hardware [11], [12], [13].

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. Although fine-grained memory protection has been proposed in the past (see Section IV), a novel aspect of our solution is that a single thread of execution can be subdivided and internally isolated into multiple, finer-grained principals. We implemented our reference monitor in a full system simulator of a modern out-of-order processor; for multithreaded applications our reference monitor requires systems software support for which we modified RTEMS (Real-Time Executive for Multiprocessor Systems). We evaluated our implementation of architectural and software modifications using a comprehensive set of commercially representative embedded systems benchmarks and a synthetic microbenchmark that exercises the reference monitor. Our experimental results show that the single-threaded performance overhead of applying our solution (to the benchmarks we used) varies from as low as 0.01% to as high as 12.17% with an average increase in execution time of 2.28%. We view these numbers as modest with respect to the security and safety gains made by achieving fine-grained memory protection.

II. Hardware Containers

Fine-grained memory protection can prevent unauthorized accesses between components of a program, which is often an early step in a sophisticated high-level attack. Our solution divides any code base into small units (functions/procedures) and creates strict bounds for each. With this approach we

prevent several classes of attacks: memory scanning, buffer overflows, raw memory writes.

Our solution's isolation primitive is the ***container***, which we define as the collection of executable code (a set of functions) together with an enforceable list of permissions and metadata that restrict the code's access to memory. Our hardware reference monitor mediates interactions between containers by interposing on accesses to instruction memory and changing the reference monitor state when the executing code calls a function belonging to a different container. We term this change in state a *security context switch* meaning that the context for making security-related decisions changes. Because the reference monitor manages container permissions and state we call it the *Container Manager (CM)*. The rest of this section describes the design of our system—including memory permissions, the internal design of the CM, integration of the CM with a complex out-of-order pipeline, and systems software support for multitasking—and practical considerations that arose during implementation.

A. Memory Permissions

The CM grants to executing code the access permissions to memory regions according to preconfigured permission lists and runtime permission changes. Memory access permissions include read, write, execute, and delegate. The first three of these should be familiar; read allows load instructions, write allows store instructions, and execute allows new code to run. Execute permissions accomodate dynamically loaded libraries, plugins and function pointers. The delegate permission propagates permissions between containers; in particular, the lack of the delegate permission prevents a container from passing permissions to any other container.

Usually a compiler can extract memory permissions along with memory allocations for program variables. Sometimes, however, the compiler cannot extract permissions: when source code is unavailable; when a system-wide runtime policy conflicts with the expected compile-time policy; and when dynamic memory references cannot be determined statically. For unavailable source code a developer can assemble permissions externally to the code and link them into the executable. When the compiled permissions conflict with a system-level policy the system will need to replace the program's permissions at load time, which can be solved by using dynamic linking for the permissions. Dynamic memory needs more support, which we describe in the remainder of this section.

A dynamic memory range poses two problems for permissions extraction: the base address and size. Often the base address is unknown until the heap allocator supplies the address. In many cases a static analysis tool can validate if dynamic ranges are accessed safely, but exceptions such as unsafe memory references and pointer arithmetic require runtime support [14], [15]. For each dynamic memory region that program code may access, the program must inform the CM about the base address and size of the region so that the CM can validate memory accesses. Our solution comprises a set of new instructions and CM logic that enables application code to handle dynamic permission creation and delegation. When compile-time tools cannot determine the bounds for a pointer reference, a programmer must intervene to disambiguate it.

We introduce a compiler primitive, ALLOW, that permits both compiler-generated and manual annotations for bounding memory accesses. To simplify permissions management the CM automatically revokes dynamic permissions when a container exits (code returns to a function in a different container). In other words, an ALLOW gives permissions only to an instance of a container and automatically revokes the permissions once the container goes out of scope. A container uses the same primitive to return dynamic data with the access permissions required. Our approach has the benefit that the target of an ALLOW is implicit; the current container gives permissions either to the next callee container, or the permissions are passed to the caller container upon a return. Figure 1 shows some examples of code using ALLOW.

(a)

```
foo(){
   char *buff = (char*)malloc(100);
   bar(buff);
}

void bar(char * buff){
   for(;i<100;i++) buff[i]=i;
}
```

(b)

```
foo(){
   char *buff = (char*)malloc(100);
   ALLOW(buff,100,PERM_R|PERM_W);
   bar(buff);
}

void bar(char * buff){
   for(;i<100;i++) buff[i]=i;
}
```

(c)

```
foo(){
   char * buff = bar();
   for(;i<100;i++) buff[i]=i;
}

char * bar(){
   char *buff = (char*)malloc(100);
   ALLOW(buff,100,PERM_R|PERM_W|PERM_D);
   return buff;
}
```

Fig. 1. Using ALLOW: (a) function foo() does not give permissions on buff before calling bar(), and at runtime the CM will prevent bar() from writing into buff. (b) foo() gives read and write permissions on the whole range of buff before calling bar(), which rewrites the data in buff; (c) bar() is a factory function that passes control for the created buffer to the caller with delegate permission so the caller can give permissions on the buffer to other containers.

Permission delegation enables access permissions to follow the flow of a program's call stack. We bootstrap permissions by initiating the heap allocator (new/delete/malloc/free) with ownership over the whole heap. We also modify the heap allocator so that dynamic memory is allocated in containers and permissions are granted to the callee by using ALLOW.

When allocating a new buffer the allocator will do its normal job to allocate space, and then it will return the buffer with access permissions only for the buffer. Our approach to protecting dynamic memory reduces the attack surface of heap-allocated data to be exactly those buffers that are accessible for normal program use; heap metadata, unallocated heap space, and unshared memory regions are protected implicitly.

Because dynamic permissions follow the call stack, the CM can store dynamic permissions of past containers in a *permissions stack* that provides efficient storage and retrieval in the usual case of a sequential execution stream. During a security context switch induced by a call instruction the CM pushes all dynamic permissions of the current container onto the permissions stack, so that during a return the security context switch can pop the permissions stack to load the dynamic permissions of the caller container into the CM.

Compiler-assisted permission extraction and delegation reduce the effort of writing programs that use fine-grained memory protection. When the compiler cannot resolve ambiguous memory references, the ALLOW primitive can be invoked directly in source code to control memory access permissions. Our solution allows permissions to follow the program control flow naturally, thus allowing local decisions about access rights and an efficient mechanism—the permission stack—for managing dynamic permissions.

B. Container Manager Details

We have so far treated the CM as a black box that tracks containers, executes new instructions for permissions management, loads permission lists, manages dynamic permission changes, validates memory accesses, and handles access control violations. The CM's internal components, shown in the architectural overview in figure 2, include

- the *Container Identification Table*: holds the definitions of all containers in a program comprising mainly their entry points and the location of permission lists.
- the *Container Runtime Record*: holds the currently executing container's permission list, which includes both a static and dynamic part. The static part contains the accessible memory ranges resolved at compile-time, the code range of the container, the full list of permissible jumps to other containers,and a function pointer to a recovery routine for handling security violations. The dynamic part is the set of dynamic permissions on memory ranges that have been granted from other containers by the ALLOW primitive.
- the *Dynamic Permissions Buffer*: a buffer for storing dynamic permission between container switches.
- the *Permissions Cache*: although not part of the CM proper, this cache stores the recently used static and dynamic permissions. All permissions loading goes through the permissions cache.

These components play a role in how the CM handles the following container-related execution events:

Program Load When the loader initializes memory with the program's code and data it also loads the Container

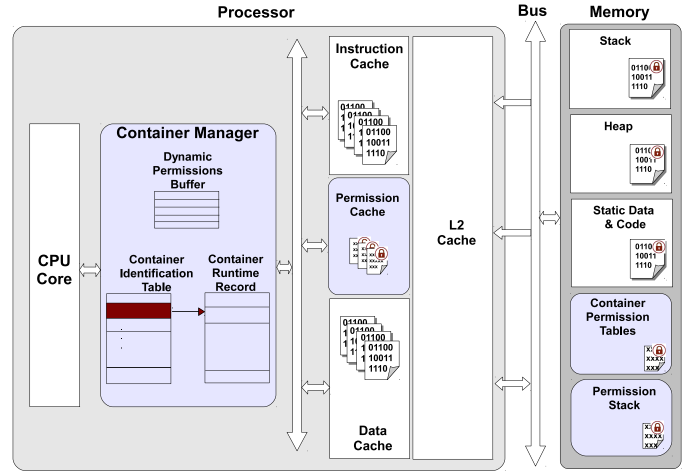

Fig. 2. Overview of how the CM fits into a typical architecture. The current container (shown in red in the Container Identification Table) corresponds to the security context residing in the CRR. The Dynamic Permissions Buffer is a temporary storage area for permissions that will be given to the next container to execute.

Identification Table.

Context switch During a context switch the OS is responsible for saving and loading this table. See section II-D for more container details related to multitasking.

Security context switch A permissible jump—usually a function call or return—from one container to another initiates a security context switch. On a **call** the CM pushes the CRR's dynamic permissions on to the permission stack, discards the caller's static part of the CRR, fetches the static part of the callee's CRR from memory, moves the Dynamic Permission Buffer into the CRR, and stores the return address onto an internal stack. On a **return** the CM verifies the return address against its internal stack, discards the callee's CRR, fetches the static part of the caller's CRR, pops the permissions stack to restore the caller's dynamic part of the CRR, and merges the Dynamic Permissions Buffer into the CRR. Similar to any hardware stack, the return address and permissions stacks enforce an expected call tree structure. Improper returns or setjmp/longjmp violate this assumption. Although we currently do not support sejmp/longjmp functionality, we could add the necessary control mechanisms to the OS and C library to handle such functionality.

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.

Executing ALLOW The CM executes ALLOW to enable containers to propagate permissions dynamically. After verifying that the current container has delegate privilege on the memory range, the CM adds the range to the Dynamic Permissions Buffer. As explained above, the security context switch merges the dynamic permissions into the CRR.

Security violation A security exception is raised whenever the currently executing container attempts to take an impermissible action, for example fetching an instruction from a memory location without execute permission, returning to a

different location than the original caller, calling a container not listed in the permissible calls list, executing ALLOW on a memory range without delegate permissions, or attempting to escalate permissions by passing ALLOW a superset of the container's permissions on a memory range. For any security violation the CM records the details of the violation (type, memory address, operation, source and destination container) and then invokes the recovery routine (exception handler) listed in the CRR.

The CM's most frequent operation is validating memory accesses—instruction fetches, and data loads/stores—against the permission tables in the CRR. Multiple such operations can occur simultaneously every clock cycle, so any delays will degrade execution performance significantly. To accelerate memory access searches we use content addressable memory (CAM), which has been widely used for fast searches such as in cache indexing using translation lookaside buffers (TLBs) and high speed routing.

CAMs are specially-designed memory modules that allow for addressing memory by the value rather than index for a fast hardware search. A shortcoming of typical CAMs is that the lookup can only be done for a fixed value, so checking for a range of values needs several accesses to the CAM. Researchers in routing and networking appliances designed specialized CAM with extended comparator circuitry for applications that require range checking (such as port ranges in IP lookups) [16].

Checking which memory range a particular access belongs to fits in the same usage pattern for both instruction fetches (to which function does it belong) and loads/stores (if the range is in the access list then validate permissions for that range, otherwise trigger an access violation). Delays for such checking is in the range of typical L1 cache hit latency (1-3 processor cycles) and occur in parallel to cache accesses.

C. 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 3 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 performance vs complexity tradeoff.

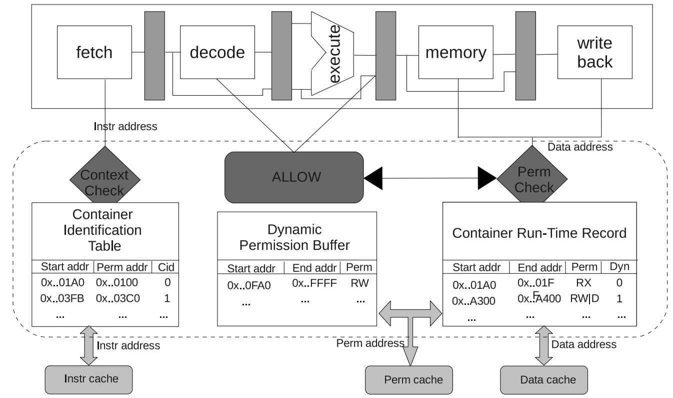

Fig. 3. A simplified view of a processor pipeline. The CM hooks into the fetch and commit (write-back) stages of the pipeline to validate memory accesses efficiently.

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.

D. 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.

E. 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.

a) 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.

b) ALLOWM: ALLOW for multiple ranges: Complex data structures such as linked lists, trees, or graphs pose a challenge for dynamic permission assignments. In order to pass a complete structure from one container to another the same processing as traversing the data structure is necessary to delegate permissions. When data nodes are distributed over a large memory space this traversal can be time consuming. To optimize such a scenario we created a variant of ALLOW called ALLOWM (for allow multi) that takes as parameters the number of ranges and the location in memory of a 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 data structure operations so that permissions can be reused without re-traversing the structure.

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 I
SIMULATED ARCHITECTURAL PARAMETERS

III. EXPERIMENTS AND RESULTS

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 [17] provides a cycle accurate micro-architecture simulator and cache model for this architecture. GEMS relies on WindRiver Simics [18] 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 I 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 [19] 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 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 [20], the Data Intensive Systems (DIS) benchmark suite [21], a reduced size Dhrystone test, and the heap-intensive Richards benchmark [22]. 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.

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 II-E. 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 4 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%.

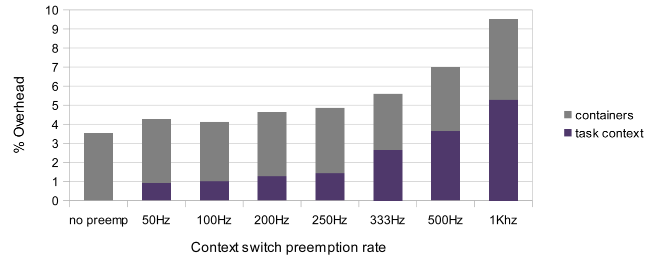

Fig. 5. Multithreaded performance. Overhead of context switch time respective to whole program execution time when running crc, susan (corners and edges), and dhrystone in separate tasks. We include the overhead from the usual task context switching (task context) above which remains the overhead caused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

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 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

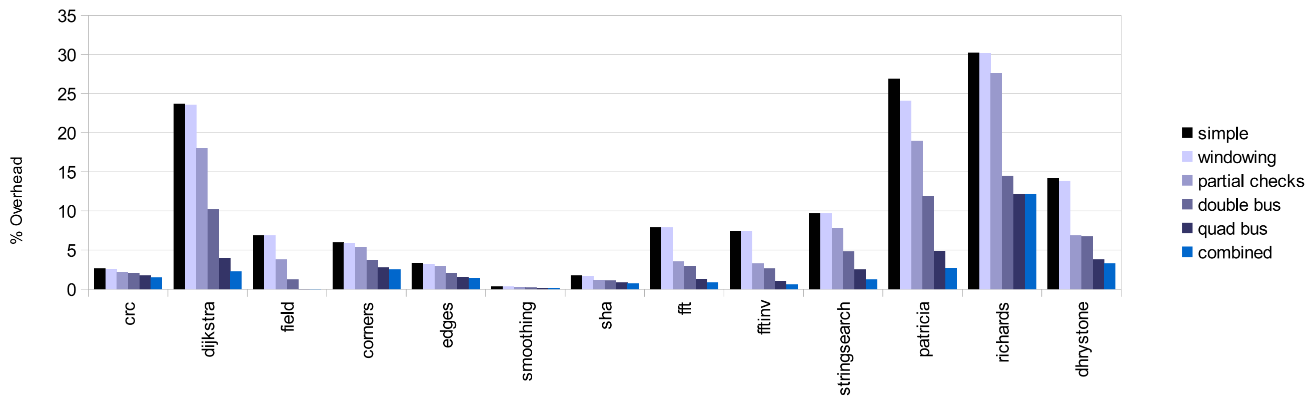

Fig. 4. Single-threaded performance. Performance overhead for each benchmark using containers without optimization (simple) and with each optimization applied: ideal container windowing (windowing), partial container checking (partial checks), doubling the CM-cache bus (double bus), quadrupling the CM-cache bus (quad bus), and the combination of windowing, partial checks, and quad bus (combined).

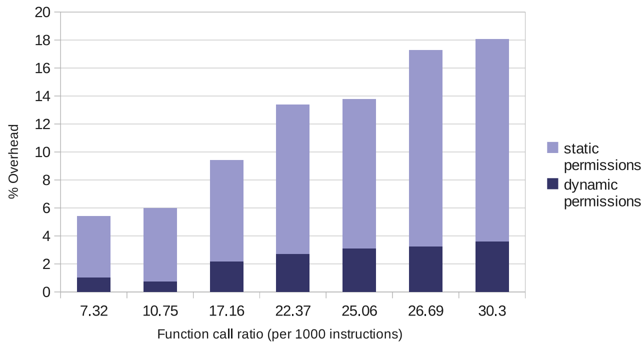

Fig. 6. Function Call Frequency. Performance overhead as the number of function calls (per 1000 instructions) increases.

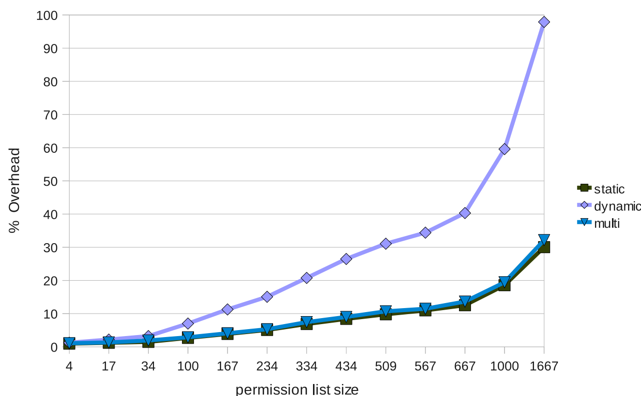

Fig. 7. Sparse Memory Access and Allow Multiple (ALLOWM). The overhead of our solution increases with the number of memory ranges that get passed between containers. Two extremes are when every permission is determined at runtime with the ALLOW statement (dynamic) and when permissions are determined 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.

switch, so the frequency of function calls affects performance.

Figure 6 shows the performance degradation as the frequency of function 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 7 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 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.

IV. Related Work

One approach to fine-grained memory protection is to place each protection domain in a distinct set of pages and then use the OS to modify page tables. Examples of this approach include StackGuard [23] and HeapServer [24]. A related solution is to divide an application into processes that use inter-process communication (IPC) mediated by the OS so that existing process and page-table protection can be used; an example is the Chromium project [9]. Unfortunately these solutions cannot support fine-grained principals because calling into the kernel to change page tables or deliver IPC is prohibitively slow. To circumvent such overhead others have proposed custom hardware solutions designed for fine-grained memory protection.

Mondrian Memory Protection (MMP) [13] uses a hardware-traversed hierarchical trie that provides word-grained read/write permission on data and thread-grained execute permission. A TLB-like caching structure accelerates permission checks based on locality. Permissions are encoded compactly to minimize the amount of memory reserved for permission storage. Unfortunately compaction complicates permission changes, and frequent changes to permissions results in frequent trie node updates that are difficult in hardware and slow in software. MMP aims to protect large code modules whose interaction is mediated by an OS. As such, the OS is invoked every time a security context switch occurs. In MMP, the smallest principal is a thread, whereas in our solution individual function invocations within a thread are protected.

Shen et al. [25] leverage program properties to offer a mixed TLB (for page-grained protection) and array-based (for word-grained protection) permission checking. They achieve low

performance overheads, but do not offer fine-grained code protection as we do.

Similarly to our work, MemTracker [11] tackles integration with a complex out-of-order pipeline. The solution is designed for validating the state of heap allocations to assist in debugging memory errors, and is not intended for security. Also MemTracker only tracks memory reads and writes and is not concerned about instruction fetches, which must be monitored to track the active principal.

InfoShield [26] proposes a hardware reference monitor that holds the access permissions for a limited number of memory locations. The limited number of protection zones—and performance penalties that come with the extra instructions executed—make the scheme impractical for protecting multiple interacting software components.

Arora et al. [27] enforce application-specified data properties at run-time by tagging data addresses with an additional security tag. For example, using a single bit SECTAG can mark read-only memory regions at a fine granularity. The checker (reference monitor) can be set to interpret and enforce the value of the security tag as specified by a program-wide security policy. The solution works for protection domains that are defined statically. When dealing with function-level principals and dynamically allocated memory the solution becomes impractical or infeasible.

Our solution differs from and improves on the related work by enforcing permissions at a finer granularity of principal. Our prior work[1]also supports a fine-grained principal but without consideration for concurrency, out-of-order pipelines, sparse data structures, or the architectural optimizations described in this paper.

V. Conclusion

In this paper we have 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. Our solution includes protection mechanisms for handling dynamic memory and concurrent applications with modest overhead. We improve on prior art by supporting fine-grained principles as small as functions and protection domains as small as a word of memory all at a modest overhead.

References

[1] V. R. Basili and B. Boehm, "COTS-based systems top 10 list," *Computer*, vol. 34, no. 5, pp. 91–95, May 2001.

[2] The Apache Software Foundation, "Apache httpd modules," http://httpd.apache.org/modules/, 2012.

[3] "Add-ons for firefox," https://addons.mozilla.org/en-US/firefox, 2012.

[4] S. Corp., "Internet security threat report," vol. 16, April 2011.

[5] J.-M. Jazequel and B. Meyer, "Design by contract: the lessons of ariane," *Computer*, vol. 30, no. 1, pp. 129 –130, jan 1997.

[6] "Apple itouch / iphone tiff image handling privilege escalation," http://osvdb.org/show/osvdb/38527, 2012.

[1]citations omitted for blind review

[7] "Sony psp photo viewer tiff file overflow," http://osvdb.org/show/osvdb/19665, 2012.

[8] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer system," *Proceedings of the IEEE*, pp. 1278–1308, Sept 1975.

[9] C. Reis, A. Barth, and C. Pizano, "Browser security: lessons from google chrome," *Commun. ACM*, vol. 52, pp. 45–49, August 2009.

[10] "Gnu coding standards: 4.8 dynamic plug-in interfaces," 2012. [Online]. Available: http://www.gnu.org/prep/standards/html_node/Dynamic-Plug_002dIn-Interfaces.html

[11] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, "Memtracker: Efficient and programmable support for memory access monitoring and debugging," *High-Performance Computer Architecture, International Symposium on*, pp. 273–284, 2007.

[12] W. Shi, C. Lu, and H.-H. S. Lee, "Memory-centric security architecture," *High performance embedded architectures and compilers, Barcelona, Spain, November 17-18*, 2005.

[13] E. Witchel, J. Cates, and K. Asanovic, "Mondrian memory protection," *Proceedings of ASPLOS-X*, Oct 2002.

[14] G. C. Necula, J. Condit, and M. Harren, "CCured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2004.

[15] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," *Usenix Annual Technical Conference, pages 275-288, Monterey, CA*, Jun 2002.

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

[17] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[18] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50–58, February 2002.

[19] RTEMS, "http://www.rtems.com."

[20] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

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

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

[23] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," *USENIX Security Symposium*, 1998.

[24] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic, "Comprehensively and efficiently protecting the heap," *ACM SIGOPS Operating Systems Review, Proceedings of the 2006 ASPLOS Conference*, pp. 207 – 218, December 2006.

[25] J. Shen, G. Venkataramani, and M. Prvulovic, "Tradeoffs in fine-grained heap memory protection," *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pp. 52 – 57, 2006.

[26] W. Shi, J. B. Fryman, G. Gu, H. H. S. Lee, Y. Zhang, and J. Yang, "Infoshield: a security architecture for protecting information usage in memory," *The Twelfth International Symposium on High-Performance Computer Architecture*, pp. 222–231, 2006.

[27] D. Arora, A. Raghunathan, S. Ravi, and N. 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.