Hardware-enforced Fine-grained Isolation of Untrusted Code

Eugen Leontie
Dept. of Computer Science
George Washington University
Washington, DC 20052
eugen@gwu.edu

Gedare Bloom
Dept. of Computer Science
George Washington University
Washington, DC 20052
gedare@gwmail.gwu.edu

Bhagirath Narahari
Dept. of Computer Science
George Washington University
Washington, DC 20052
narahari@gwu.edu

Rahul Simha
Dept. of Computer Science
George Washington University
Washington, DC 20052
simha@gwu.edu

Joseph Zambreno
Dept. of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011
zambreno@iastate.edu

ABSTRACT

We present a novel combination of hardware (architecture) and software (compiler) techniques to support the safe execution of untrusted code. While other efforts focus on isolating processes, our approach isolates code and data at a *function* (as in, C function) level, to enable fine-grained protection within a process as needed for downloaded plugins, libraries, and modifications of open-source projects. Our solution also enforces timing restrictions to detect denial of service from untrusted code, and supports protection of dynamically allocated memory. Because bookkeeping data can become substantial (permission tables that at their finest granularity describe which memory words may be accessed by which functions), our solution employs a stack-structured bookkeeping mechanism that tracks the flow of execution and automatically dispenses with bookkeeping data when no longer needed. This approach also enables an architectural optimization to handle permissions for dynamically allocated memory, allowing heap blocks to be appropriately shared across the trust boundary. Tested across a suite of benchmarks, our solution had a worst case 12% overhead and 3.5% average overhead at the finest level of code granularity (every single function in its own unit of isolation). The overhead is easily reduced by using trace-driven analysis to combine functions into coarser-grained groups that share permissions.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]

General Terms

Security, Design, Performance

Keywords

Memory protection, isolation, fine-grained protection, software security, architectural support for security

1. INTRODUCTION

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 [21] – 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 [13]. For example, a report by Symantec [22] shows that in the first six months of 2007, no fewer than 300 security vulnerabilities for web browsers originated from security flaws in popular add-ons (e.g. JVM, QuickTime, Adobe PDF reader, Adobe Flash player) [5].

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 [24], or have proposed carefully managing the interactions between processes [8, 7]. 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.

The main contribution of this paper is a *function-oriented memory protection* approach that, for the most part, requires no rewriting of code but merely identification of untrusted functions or groups of functions – the untrusted code. Such identification can be provided either at compile-time (in a project's `makefile`, for instance) or at load time. Furthermore, our approach does not need an operating system to mediate communication between isolation units or supervise the management of permissions. The handling and checking of permissions is performed entirely in hardware. In addition, our approach has the following useful features:

- *Arbitrary domain and principal granularity*. Our approach allows fine granularity on both sides of an access permission: the permission domain can be as small as a single word and the principal can be as small as the smallest function (a single instruction with an accompanying return). Because fine granularity can result in significant bookkeeping data – for the currently executing function, our hardware needs to know what memory the function can access – we propose a *stack-structured* approach in which permission tables are loaded and discarded automatically as functions are called and return. This approach has the further advantage that all permission-related metadata are already in the hardware when needed and therefore avoids the equivalent of a "page fault" that is characteristic of on-demand approaches.

- *Dynamic memory*. Our stack-structured approach makes it easy to incorporate permissions for dynamically allocated memory. In particular, a function may allocate memory (using, say, `malloc`) and set permissions on it before passing the pointer to untrusted code. Note that this feature needs compiler support (to automatically insert such permission setting) and a one-time re-write of the allocator.

- *Minimally invasive architectural support*. The architectural modifications we devise are designed to sit between cache and CPU in a single unit that can be added to a processor core without modifying either the instruction set or the rest of the architecture. However, some optimizations are possible if one is willing to modify the instruction set (for setting dynamic memory permissions) and CPU (to optimize the instruction pipeline for performance).

- *Protection from denial of service*. One of the simplest attacks is to insert an infinite loop or an especially long computation that slows the processor. Thus, when calling an untrusted function, one would like some assurance that it will return within a reasonable time. Our architectural modifications include execution timing that is checked against predetermined limits, which can be set by the application developer based on profiling.

- *Control flow monitoring*. Our architectural modifications make it easy to check proper control-flow between units of isolation so that only pre-defined entry points are allowed, thus preventing attacks based on unauthorized jumps into code [3].

- *Tunable efficiency*. While it is unlikely that plugins will consist of single functions, our implementation shows a modest 3.5%-average overhead on benchmarks in which every single function is assigned its own unit of isolation. Overhead is incurred when a function call crosses an isolation boundary and will be much reduced with coarser granularity, allowing developers some flexibility in trading off strong isolation with performance.

The remainder of this paper is organized as follows: Section 2 reviews prior work in memory protection; Section 3 introduces our solution and provides the details of our hardware and software modifications; Section 4 evaluates the performance overhead introduced by our hardware enforced fine grained memory protection while Sections 5 and present discussion and concluding remarks.

2. RELATED WORK

The idea of isolation dates back to the first generation of timesharing machines. We adopt the terminology of Saltzer and Schroeder [16] 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.

At the present time, commodity systems exploit hardware support for paging by including additional permission bits for each page that determine whether the currently executing process can access the page. The assignment and revocation of permissions is managed by the OS, with the process as the principal. However, such page-based permissions come with several drawbacks. First, 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. Second, changes to permissions are mediated by the OS and therefore require modifications to the OS. Third, as mentioned earlier, since the OS is process-based, such a mechanism does not work for the plugin and open-source applications considered in this paper. Fourth, the OS can only manage the page tables during a context switch, when a process suspends or resumes execution. 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.

Recently, some research efforts have begun to address the problem of memory protection by proposing architectural modifications that provide memory protection at a finer granularity of domain than pages (words of memory) for smaller principals than processes (threads). One influential work by Witchel et al. is Mondrian Memory Protection (MMP) [24], which improves on the prevailing page-based model by allowing permissions to be specified on domains as small as the word size.

Some researchers have proposed using encryption to protect memory [4, 12, 18]. For example, Shi et al. describe MESA [18], an architecture for a secure memory system with both access control and encryption for tamper proof storage. Although MESA supports separating code and data (memory spheres) into different principals (principles), the encryption increases the cost of sharing between principals. Thus, using MESA to protect memory between functions would impose a high cost on marshaling parameters.

With a different objective and approach, InfoShield [17] protects a fixed set of highly critical data, such as secret keys. However, InfoShield allows access to anything that is not explicitly protected. In contrast, our solution by default denies access to all data and thus can protect more data than InfoShield.

Arora et al. [1] introduce security tags for data ranges to support memory protection policies, including isolation for a large number of small principals. However, the complexity and cost of switching between principals becomes prohibitive as the number of security tags increases. To protect functions, the security tags must be defined statically and a static permission map of the entire application must be known. Such information is not generally available for dynamically allocated memory.

At the other end of the hardware-software spectrum lie software-only solutions, such as static techniques based on information flow [27], executing untrusted code in an interpreter [10], or specialized isolation for operating system drivers [20, 23]. Type safe languages are also used by Microsoft's Singularity project [8, 7]. These approaches all have some drawbacks, either in extra overhead, no support for weakly typed low-level languages, or weak support for dynamic memory.

Our solution combines hardware, in the form of a memory protection hardware module, and software, as a compiler module that inserts protection-assigning instructions for dynamic memory, and differs from previous work in several ways. As mentioned earlier, the main difference is that we are focused on fine-grained *function-oriented memory protection* suited to plugin software architectures, the importing of libraries, and unbridled modification of open-source code. We explored the high level motivation and introduced our solution in a previous paper [11], which discusses using our architecture to support component-based software but omitting architectural details; in this paper, we present those details.

We are able to achieve this protection without prohibitive costs by leveraging call-return semantics to get automatic prefetching and revocation of static permission data, in much the same way that the call stack provides automatic memory management of stack-based data (parameters, local variables). This stack-structured approach, as it turns out, also lends itself to an efficient way for managing permissions on dynamic data. In addition, we provide timing protection for denial of service attacks and control-flow checking to detect unauthorized jumps, and our modifications are minimally invasive in terms of the required architectural redesign.

3. CONTAINER-BASED APPROACH

A primary goal of our approach was to make both our 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 1 (The details are described in Section 3.3.) In this manner, a processor designer would merely insert our module and interface it with both the CPU and the cache. 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. In fact, the only action on the part of the programmer is to define the units of isolation – this requires modification of a software development tool that builds the application (like `make`). Figure 1 also shows the additional bookkeeping data (the permissions) stored in main memory, which are fetched into the Container Manager as needed to enforce isolation boundaries. We next describe details, starting with a definition, followed by software support in Section 3.2 and architectural details in Section 3.3.

3.1 Containers

We use the term *container* to describe our unit of code or data isolation. On the principal (code) side, this unit can be as small as

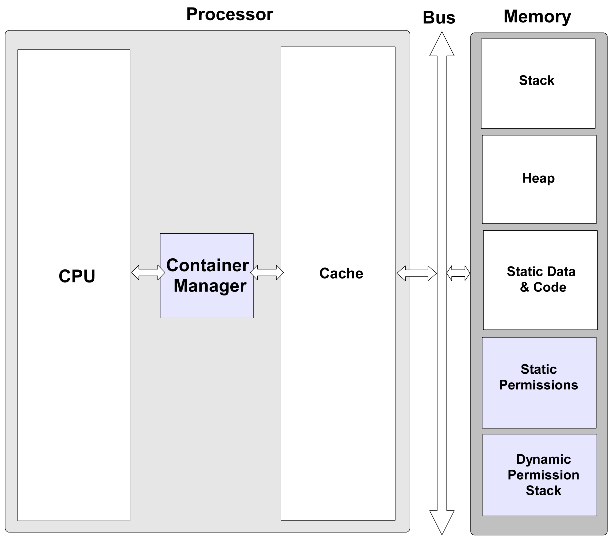

Figure 1: A high level view of our architecture. The container manager acts as a reference monitor for memory access control, with additional cycle counting to detect denial of service.

a single function, or can be as large as a substantial library of functions, or even the whole application. On the domain (data) side, the unit can be as small as a single word or an arbitrarily large buffer. Most often, since plugins and libraries contain both code and data, a container will protect one or more functions and some data as a single unit. Every container has a unique identifier, the *Container-ID*, and has a set of internal bookkeeping data (metadata, permission tables) that are used by the hardware to enforce memory protection. This data includes static permissions that govern accesses to global data ranges and stack allocated variables, as well as code entry/exit points.

3.2 Compiler Support

We augment a standard C compiler to extract container boundaries and permissions. By default, each function is assigned its own container and has full access permissions to static variables. Variables that are passed to a function are given read-only permission if the `const` keyword is used, and otherwise are given read and write permissions. Memory ranges for static variables are automatically calculated, so bounds checking is enforceable by the hardware.

In the traditional software development approach, an application developer creates a project file (for example, `makefile`) to build the application using a collection of configuration, compilation and loading tools. In our approach, the software product goes through an additional step: extracting an application permission *manifest*. The manifest consists of all the information needed by the hardware: a list of containers, their identifiers, memory permissions, permitted call patterns, and approximate run times (in terms of the number of instructions executed) for selected containers of interest. Although most permissions can be extracted automatically at compile-time, some permissions for dynamic memory accesses are only computable at runtime (due to the lack of bounds checking, array index computation, and pointer aliasing in C). In such instances, the compiler is unable to infer the correct permission assignments, and the programmer is instructed to add permission annotations. The tool then generates the manifest, which can be edited manually by the developer. Our point here is that both developers and language designers have the option of using the hardware to appro-

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

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

```
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;
}
```
(b)

```
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;
}
```
(c)

Figure 2: Use of the ALLOW macro. (a) shows an unprotected code snippet with a possible violation since `i` is not initialized in `bar`. (b) shows how `ALLOW` is added in `foo`, giving permission to the next function (`bar`) to read/write the memory located at the address of `buff` and continuing for 100 bytes. (c) shows how `ALLOW` is used to produce dynamic memory and `bar` provides permissions to its caller (`foo`).

priately strengthen robustness. In some languages such as C with much low-level pointer manipulation, more programmer intervention may be needed, whereas the burden on developers is considerably less in higher-level languages or even in C variants such as Cyclone [9]. The entire application now consists of all the executable code and static data along with the manifest.

Note that in the above scenario the developer leaves a number of decisions to the compiler. For example, consider the code example in Figure 2(a) which shows a function `foo()` calling a function `bar()`. If the developer places both in the same container, `bar()` has permission to access the buffer `buf` passed into it. On the other hand, if the two functions are in different containers, explicit permission needs to be granted. In this case, as shown in Figure 2(b), the compiler default is to insert a macro that we call ALLOW to specify the read-write permission on this shared data that crosses container boundaries. In the case, for example, where the type modifier `const` is used in C/C++, the compiler will insert an ALLOW macro with read-only permission. Similarly, for a heap-allocated block that is returned, the compiler inserts an appropriate ALLOW macro as shown in Figure 2(c). Other cases include call-by-reference parameter passing or any situation in which a reference is passed between containers. In this manner, the compiler inserts a number of such ALLOW macros depending on how the compiler infers the desired permission.

During the code generation phase, the compiler translates ALLOW macros into a special instruction called `pcd` (permission control delegate), that enables the hardware (the Container Manager) to manage the permissions at runtime. The `pcd` instruction takes a memory range and permission type (for example, read-only) as operands and associates the permission with the given range of memory. Figure 3 shows how a single ALLOW macro translates into two machine instructions, one of which is `pcd`. (An alternative to defining a new machine instruction is to use memory-mapped I/O, at the cost of requiring more instructions - Figure 3c).

```
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 3: ALLOW implementation: (a) in high level language; (b) implementation using a new instruction, *pcd*; (c) implementation using memory mapped operations.

Although the compiler is able to extract permission information needed for static memory accesses, dynamic memory is another matter. Handling dynamic memory requires both a modification to the memory allocator and a mechanism for creating memory permissions at runtime. Recall that the compiler (or programmer) generates the high-level macro ALLOW, as shown in Figure 3(a), for explicitly assigning permissions that carry across container boundaries. Thus, if dynamic memory is allocated in one container and needs to be passed to another, the ALLOW macro is inserted to assign the default permission. The compiler can be set up to use other defaults (for example, a default of read-only) and directives. Thus, assignment of permissions is straightforward. However, revocation is more complicated. Consider how revocation is handled in process-oriented approaches: Tables are maintained by the OS [24], and permissions must be "undone" when the permission expires or goes out of scope. Most often, this revocation occurs on every return from one domain to another. In contrast, our approach exploits the execution stack by letting dynamic permissions disappear implicitly when no longer needed so that no explicit revocation is required. The Container Manager described in the next section uses stack like structures to save the permission context. Permissions for dynamic memory are kept active only for live functions, and revocation is automatic when functions return. This strategy avoids frequent supervisor calls for execution patterns that follow an activation tree structure. For other execution patterns like exceptions, setjmp/longjmp, and preemptive context switches, the appropriate mechanism is a software-based supervisor in the kernel.

3.3 Hardware Support

Achieving acceptable performance with fine-grained memory protection requires augmenting existing hardware. In our hardware design, we extensively use three types of memory: registers, scratchpad RAM, and content addressable (associative) memory (CAM). Figure 4 shows the architecture of the Container Manager, the mod-

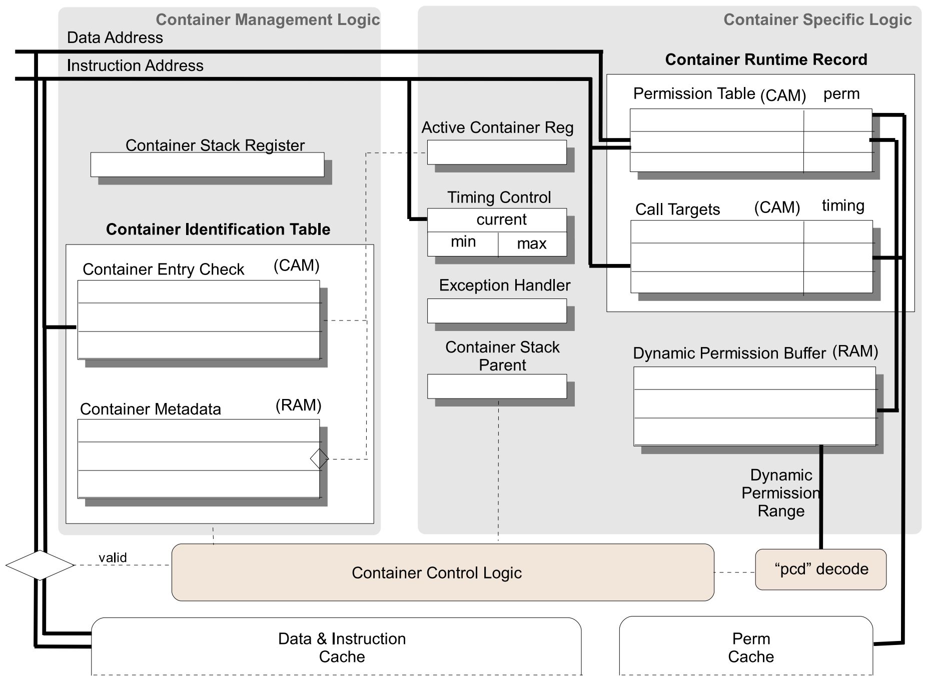

Figure 4: Details of the Container Manager. The Container Manager is modularized into three pieces: the Container Management Logic handles all storage that applies across the set of all containers; the Container Specific Logic handles storage for only the currently active container; the Container Control Logic implements the functionality using the storage.

ule that we introduce between CPU and cache, conceptually divided in three components: Container Management Logic (for handling data related to all containers), Container Specific Logic (for information about the currently executing container), and Container Control Logic (to implement permission checking, dynamic permissions management, timer management, and container switching).

3.3.1 *Container Management Logic*

Some of the Container Manager's hardware tracks state across all of the containers for an application. This hardware resides in the Container Management Logic, which comprises the Container Identification Table and the Container Stack Register. The Container Identification Table has both the Container Entry Check CAM and the Container Metadata RAM. The Container Entry Check monitors every instruction address to identify if a new container is being accessed. The set of all container entry points is prepared by the compiler and pre-loaded with the executable binary, so that checking for entry points involves only a single cycle CAM lookup. The data in the Container Metadata are paired with container entry points to specify where the container's static permissions are located.

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 Container ID is used for grouping the constituent functions, and for detecting whether the container context needs to be switched. For each memory fetch, the Container Manager checks the address 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, which is executed by the Container Manager independent of any supervisor (OS).

3.3.2 *Container Specific Logic*

For each container, its static and dynamic permissions are stored by the Container Manager in the Container Specific Logic, consisting of the Container Runtime Record, Dynamic Permission Buffer, Timing Control, and a handful of bookkeeping registers. Static permissions to memory ranges are stored in the Permission Table, a range checking CAM, and allowable function calls are stored in the Call Targets CAM. The Dynamic Permission Buffer temporarily holds the permissions that the active container provides to the next container to execute. The Timing Control hardware consists of a cycle counting timer and a pointer to a stack of timers. There are also registers that contain the active container identifier, the parent of the active container (caller), and a pointer to an exception handler to assist with memory access or timing exceptions.

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 must be 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 now also available in range-checking variants (with extended comparator circuitry) that store memory ranges, and return the range in which a given data word resides [19]. This is exactly the functionality needed for a permission check: given an address, identify whether it lies in the set of ranges stored for the current container.

3.3.3 Container Control Logic

As instructions are fetched, the address of the instruction is compared with the known entry points of containers in the Container Identification Table, and also with exit points in the Container Runtime Record to see if this instruction indicates a container switch is impending. Similarly, for data accesses, the Container Manager checks the address and access type with the permission tables in the Container Runtime Record. For `pcd` instructions, the Container Control Logic decodes the instructions and loads the Dynamic Permission Buffer as directed by `pcd`.

Timing is tracked by the number of instructions committed, and timing constraints are specified by software developers. If a maximum timeout is specified for a container, a countdown timer is set to the timeout. For a hierarchy of container calls, the timer in the top container must include the timing of its callees. If a callee's timeout is greater than the caller's timeout, then the callee will inherit the more restrictive timeout. A stack of timers is maintained, with only the active container's timer being updated on each committed instruction; when the child container returns, the elapsed time of the child invocation is subtracted from the parent's timer. A timer overflow is checked only for the active container.

When the active container executes a call or return instruction, a container switch occurs. In the case of a call, if the target is found in the Call Targets, the call is allowed to proceed and the Container Entry Check CAM is searched to find the static permissions of the callee, and the dynamic permissions of the active container are pushed to the Dynamic Permissions Stack (the static permissions are discarded). In the case of a return instruction, the Container Stack Parent register indicates where to find the static permissions, and the Dynamic Permissions Stack is popped to retrieve the parent's dynamic permissions, which are written to the Permission Table. In both cases, all valid data in the Dynamic Permission Buffer are merged in to the Container Runtime Record, allowing for dynamic buffers to be respectively passed and returned to callee and caller. The container switch also updates the active, parent, and timing registers.

4. EVALUATION

Our solution imposes performance overhead from added instructions for managing dynamic permissions and from interposing on memory accesses. To evaluate the overhead, we instrumented the SimpleScalar simulator [2] with our modified hardware in an out-of-order processor model with the ARM ISA. For the software support, we modified the `gcc 3.3` cross-compiler for the ARM platform, translating the `ALLOW` macro to two instructions for `pcd`.

Our baseline simulator was SimpleScalar's `sim-outorder` configured as a typical embedded system, with a 400 MHz CPU, 100 MHz bus and memory. The data and instruction cache sizes for the baseline results were matched to the cache sizes of the 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.

To evaluate the performance of our solution, we ran experiments using a range of benchmark applications. We chose computationally intensive benchmarks from MiBench [6], data intensive benchmarks from the Data Intensive Systems (DIS) benchmark suite [14],

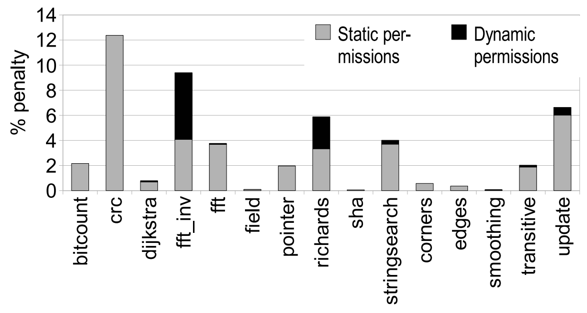

Figure 5: Performance Overhead with 16 KB Permission Cache.

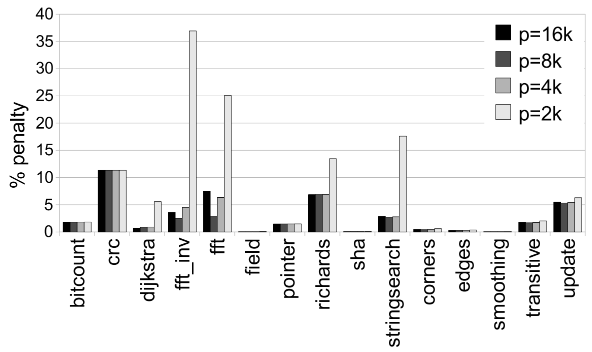

Figure 6: Performance Overhead with Varying Permission Cache Size, Fixed 16 KB Instruction And Data Cache Sizes.

and, to stress the dynamic permissions, the heap intensive Richards benchmark [26]. Each benchmark was executed on the baseline platform, and then with the modified architecture with varying parameters. All of our results are presented as the percent overhead compared with the baseline performance, so lower is always better.

Figure 5 shows the percentage overhead for all benchmarks, with a 16 KB permissions cache, with a breakdown of the overhead into static and dynamic components. The results are not surprising, since some benchmarks use more 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 6 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 more sensitive than others. Overhead decreases rapidly as the permission cache size is increased relative to the data/instruction cache size, but the decrease tapers off after 4 KB, suggesting that permissions exhibit considerable locality and only a modest cache size is needed.

Figure 7 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

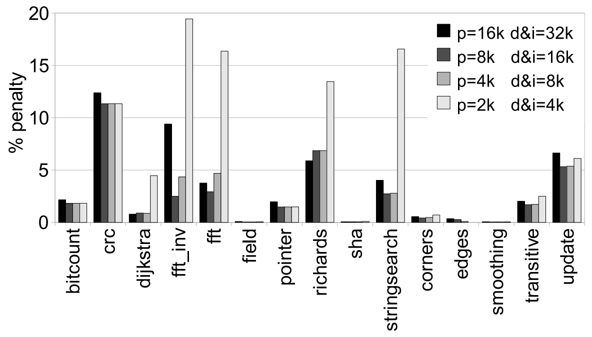

Figure 7: Performance Overhead with 2:1 Ratio of Data and Instruction Cache Size to Permission Cache Size

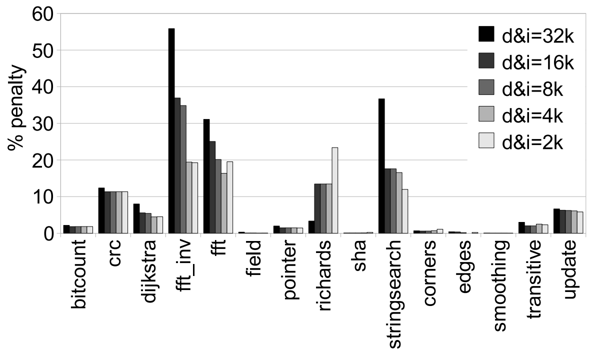

Figure 8: Overhead with a Limited (2 KB) Permission Cache

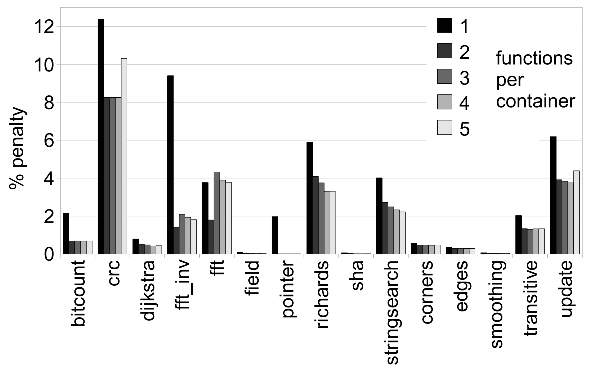

Figure 9: Overhead with Multiple Functions per Container, 32 KB Data and Instruction Cache, 16 KB Permission Cache

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

Figure 9 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 – 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.

5. DISCUSSION

We have shown that fine-grained monitoring of code execution is feasible with modest overhead, minimal software effort and a minimally-invasive architecture. Our solution is successful because we use the behavior of call-return programs to motivate our permission management's stack structure.

5.1 Exploiting Stack Locality

As a program executes, the call stack naturally reflects data and code locality. By using a stack structure for permission tracking, our solution directly exploits this locality. When the code for a container is first loaded, all of the permissions for the container's stack-allocated variables are also loaded. Thus, permissions for static data are automatically retrieved with the instructions that will use that data, hiding the cost of fetching the permissions. When a container calls another container, its dynamic permissions are stored on a permissions stack. When the container resumes, the dynamic permissions are restored from the stack. When a container exits via a return instruction, its dynamic permissions are automatically revoked, because they are no longer stored anywhere. Thus, permissions are efficiently managed in a stack structure, similar to how the call stack achieves automatic memory management.

5.2 Minimally Invasive Architecture

The results we present are based on assuming that precise identification of container boundaries is feasible in hardware. However, very complex pipelines might make extracting such information difficult. We also tested a solution in which the compiler adds instructions to flush the pipeline when a container switch should occur. This flushing added an average of 2% overhead to our performance results, but represents a very simple solution that can minimize the invasiveness of our architectural changes.

5.3 Applications of the Approach

Plugins and extensions for web-based applications routinely link in third-party code to run in the same address space as the browser itself [15], a perfect fit for our model. High performance server applications also use a plugin architecture, such as the Apache httpd Modules [21], that provides feature customization for servers. Finally, commodity operating systems use a modular framework for device drivers, but the drivers are themselves executed within the kernel address space; using our solution, driver functions could be called with restricted privileges. Note that adding such protection to an OS kernel is not trivial, as demonstrated by Mondrix [25], an application of MMP to Linux.

5.4 Secure Loading

Thus far, we have not addressed an obvious attack: what if untrusted code is able to modify the metadata (manifest) to overwrite its privileges? The loading of permission data can be made secure through standard techniques such as encryption and integrity checks. Also, ensuring that the manifest is distributed and installed properly requires a special-purpose trusted installation mechanism.

6. CONCLUSION

We have presented 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 granularity than existing solutions, which rely on the OS to manage permission tables at a process or thread level. We also protect memory at a finer granularity than page-level, allowing permissions to be specified to individual words if necessary. The fine granularity of protection allows for low overhead protection of code without substantial code re-writing. Our solution also includes mechanisms for handling dynamic memory, monitoring control-flow, and detecting denial-of-service, all with modest overhead.

Acknowledgments

This work is partially supported by NSF grants ITR-025207 and CNS-0934725 and AFOSR grant FA9550-09-1-0194.

7. REFERENCES

[1] D. Arora, S. Ravi, A. Raghunathan, and N. Jha. Architectural support for run-time validation of program data properties. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(5):546–559, 2007.

[2] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, Feb 2002.

[3] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

[4] O. Gelbart, P. Ott, B. Narahari, R. Simha, A. Choudhary, and J. Zambreno. CODESSEAL: A compiler/FPGA approach to secure applications. In *Proceedings of the IEEE Conference on Intelligence and Security Informatics (ISI)*, 2005.

[5] C. Grier, S. Tang, and S. 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.

[6] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE Workshop on Workload Characterization*, 2001.

[7] 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)*, Mar. 2007.

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

[9] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the Usenix Annual Technical Conference*, pages 275–288, June 2002.

[10] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the USENIX Security Symposium*, 2002.

[11] E. Leontie, G. Bloom, B. Narahari, R. Simha, and J. Zambreno. Hardware containers for software components: A trusted platform for COTS-based systems. In *Proceedings of the IEEE/IFIP International Symposium on Trusted Computing and Communications (TrustCom)*, Aug. 2009.

[12] D. Lie, C. Thekkath, M. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Nov. 2000.

[13] P. Loscocco, S. Smalley, P. Muckelbauer, R. Taylor, S. J. Turner, and J. 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.

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

[15] Mozilla Corporation. Firefox add-ons. https://addons.mozilla.org/, 2009.

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

[17] W. Shi, J. Fryman, G. Gu, H. Lee, Y. Zhang, and J. Yang. InfoShield: a security architecture for protecting information usage in memory. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 222–231, 2006.

[18] W. Shi, C. Lu, and H. Lee. Memory-Centric security architecture. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, pages 153–168, 2005.

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

[20] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 22(4), 2004.

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

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

[23] D. Wagner. Janus: An approach for confinement of untrusted applications. Technical Report CSD-99-1056, UC Berkeley, 1999.

[24] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 304–316, 2002.

[25] E. Witchel, J. Rhee, and K. Asanović. Mondrix: memory isolation for linux using mondriaan memory protection. *SIGOPS Operating Systems Review*, 39(5):31–44, 2005.

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

[27] S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3), Aug. 2002.