Memory Segmentation to Support Secure Applications

Student: Jonathan Woodruff
Supervisor: Simon W. Moore
Project Lead: Robert N. M. Watson

University of Cambridge, Computer Laboratory

Abstract.

Current CPU architectures provide only weak support for software segmentation, a key underpinning for software security techniques such as sandboxing, managed languages, and static analysis. Because hardware memory segmentation is relevant mainly in the program abstraction its support has been deemphasized in modern operating systems, yet modern hardware requires operating system support to use its segmentation features. This paper argues that by implementing a *capability* model, it is possible to safely support creation, distribution and use of segments purely in user space. Hardware support for user-mode segmentation would enable efficient sandboxing within processes, enforcement of compiler structure, managed languages, and formal verification of machine code. We present a working prototype of such a system implemented on an FPGA and running FreeBSD.

1 Introduction

Sandboxing, managed language runtimes, bounds checking, and static analysis are software vulnerability prevention and mitigation techniques that rely on the idea of memory segmentation, that is, that memory should be addressed as independent objects with sizes and properties rather than as a single contiguous space. However modern software has gone to great lengths to implement segmentation using hardware primitives not intended for the purpose. Examples include sandboxing, which has used process separation [18] using the paged memory translation lookaside buffer, or TLB [20], and managed languages, which use a blend of static analysis and general purpose instructions for bounds checking [15].

A dominant instruction set architecture, IA-32, actually supports segmentation in user space, however it depends on the operating system for protection and manipulation [2]. We observe that fine-grained segmentation features are primarily useful in the program abstraction, yet available hardware segmentation models have required management from the operating system [19]. This mismatch, and the fact that RISC architectures did not arrive at a consensus model for segments, meant that hardware segmentation has not become a useful feature to the modern software stack.

We have developed a RISC segment model which maintains the single-cycle instruction principle and a load store architecture. Our segment model allows creation and distribution of segments along with protection domain crossing entirely in user space. Our model is intended to support a full *capability* model [13] within a RISC processor. This capability segment model can efficiently support sandboxing [10], managed language runtimes, and static analysis. We have implemented our model as an extension to the 64-bit MIPS instruction set and have a working prototype on an FPGA which runs FreeBSD. We are beginning to evaluate our approach through a set of application use cases that include sandboxing and language runtimes and this will require the development of software models and an evaluation methodology.

2 Background and Motivation

2.1 History of Segmentation

In the 1950s and 1960s, the expanding size of computer programs made obvious the need for memory protection and virtualization [5, 17]. Programmers of the time identified segments as a direct solution for their require-

ments, allowing them to protect and relocate program modules [6]. However a more forward-looking proposal in 1961 suggested that if programs were willing to sacrifice precise control over protection and virtualization to use regularly sized pages, an operating system could efficiently allow multiple programs to run on the same machine [8]. Large-scale computers generally moved to paging to support operating systems with multiprogramming [16]. Nevertheless, computers that are expected to run a single program often support segmentation, for example the Intel 286 [2] and the modern ARM Cortex R cores [9]. As computer designers preferred paging to segmentation, programmers eventually found that strict high-level languages could enforce memory protection to some degree [4] and that managed languages could actually verify memory references at run-time [1]. Thus hardware segmentation fell out of use for large computer programs.

Intel's IA-32 instruction set demonstrates this historical pattern. The Intel 286 was the first Intel x86 processor to support memory protection [2], but because it was not expected to be used primarily for multiprogramming, a segmentation model was devised that might support protection for program objects. However segments were found to be awkward to use since segment descriptors were manipulated in kernel mode, a higher privilege level than that of the program which was creating and using the objects. The next generation of x86 processor, the 386, supported paging in addition to segmentation to enable standard multiprocess operating systems. Eventually the segmentation system was rarely used and was mostly dropped in the transition to x86-64 [3].

2.2 Motivation for Renewed Segment Support: Security

While strict programming languages and managed runtimes have been sufficient to ensure a level of working correctness, both have shown cracks when facing intentional exploitation [15]. These flaws were often due to compilers and runtimes being programmed in unsafe languages themselves. Recurrent and escalating problems with security due to the ever more connected global network of computers have shown that a working correctness is not sufficient for security conscious programs.

Maintaining security in the face of a global collection of capable adversaries requires very strong correctness of security mechanisms. While securing individual client computers does not automatically make a network secure, many network intrusions begin with a violation of a client's memory protection. A comprehensive hardware segmentation system would make direct, automatic and continual protection of objects in memory available to compilers and runtimes to improve performance and simplify enforcement.

2.3 The Capability System Model: A Guide for a Comprehensive Segment Model

In 1966 Computer Science began to clearly discuss a cohesive, decentralized system for program correctness and security [7]. The result was the development of *capability* theory. A capability is an unforgeable token of authority which can be used by a program component and which can be manipulated according to rules and delegated to another program component. Computer scientists spent much effort toward proving that capability systems could be correct and safe and even constructed several hardware capability systems [13] [14], including commercial systems from IBM [12] and Intel [11]. Many of these hardware capability systems were the pinnacle of program-centered, segment-based computers as their memory protection systems often did not depend on a central authority for inter-domain transactions, but on a set of hardware enforced rules that allowed safe, direct sharing between program components. These hardware capability systems were mostly overlooked by industry because they sacrificed performance for memory safety in a time when RISC processors were demonstrating how much performance was available to simplified integrated circuit computers [11]. However, we can learn from these validated capability machines to build a comprehensive segment system for modern RISC processors.

3 A Capability Segment Model for RISC Processors

A segment mechanism that implements the capability model of safe, programmatic memory protection should have three properties:

- All memory accesses must be via segments.
- The program must be able to restrict its segments but not expand them.
- The program must be able to pass control between domains possessing different segments without granting additional segments to either domain.

The first property, the non-bypassable property, implies an additional layer of indirection before the page table that all memory requests must pass through.

The second property, the guarded manipulation property, requires that segment descriptors be distinguished from general purpose data by the processor hardware, not only in registers, but also in memory. This segment manipulation property also implies new instructions for manipulating segment descriptors, or at least special restrictions on general purpose instructions when manipulating segment descriptors.

The third property, the protected call gate property, suggests a special flow control instruction with the ability to safely leave a domain, as defined by a set of segments, and enter a new domain.

3.1 Our Instruction Set Architecture

We have chosen to implement our RISC capability model by extending the 64-bit MIPS instructions set. We selected MIPS because it has a well established 64-bit instruction set and adheres to a prototypical RISC philosophy.

We chose to implement the universal enforcement property by adding a segment table through which all virtual addresses are offset before reaching the page table. General purpose loads and stores are implicitly offset via a fixed segment in the table and instruction fetches are offset via another fixed segment. This fixed implicit offset model is very similar to the i286 segment model. We have also added a complete complement of new load and store instructions which allow explicit use of other segments in the table.

By storing segment descriptors in a distinct register file, separate from general purpose registers, we partially fulfill the guarded manipulation property by not allowing general purpose manipulation. We also added a dedicated set of segment descriptor manipulation instructions which only allow reducing the privilege of descriptors. We decided to protect segment descriptors in memory using tags on general purpose memory locations rather than using dedicated memory so that compilers could treat descriptors as they do pointers, including storing them on the stack and passing them as arguments.

We are able to fulfill the protected call gate property with a single-cycle instruction. We observe that a domain can be trusted to manually store its own state and to "unpack" its own state when it is called. Since any number of segment descriptors can be stored in a segment, it is only necessary to make a single segment available in a new domain for the entire domain to be "unsealed". To facilitate this, we allow segments to be sealed by a domain and passed to other domains to be used in protected calls. This simple primitive is sufficient to support protected procedure calls and is easily implemented as a standard single cycle instruction.

Possibly the most radical feature mentioned above is tagged memory, which implies an extra bit for each word in memory. We were able to implement this with a simple scheme which is practical in a traditional computer system without modifying the external memory hierarchy. The tags for DRAM locations are stored together in the top of DRAM and a small tag controller sits at the mouth of the memory controller on the processor and provides a tag for every memory request made by the core. Our segment descriptors are 256 bits wide and must be aligned in memory, thus we have an overhead of 1-bit for every 256-bits of data (i.e. 0.4%).

4 Proposed Evaluation of the Capability Segment Model

We have only begun to evaluate our segment implementation, but we hope to demonstrate that several key applications will benefit from our RISC capability segment model.

4.1 Protected Procedure Calls

We would like to thoroughly study the low-level performance of crossing between protection domains using our novel RISC style protected procedure calls. This will include various trust relationships: mutually untrusting domains and calls with asymmetric trust. Our methodology here will likely focus on performance and compare against x86 segmentation domain crossings as well as domain crossings using seperate address spaces.

4.2 Sandboxing

We would like to demonstrate that our capability segment model can conveniently and efficiently support application sandboxing. State of the art application sandboxing depends on process separation using TLB enforcement. Thus protected interactions which are logically procedure calls must be implemented as inter-process communication involving the operating system and two independent memory spaces. Allowing program components to run in a hardware enforced segment of the program's address space should allow a simpler communication model and more efficient use of the TLB.

We have experience with the Capsicum [20] sandboxing API in FreeBSD and have a version of gzip which uses Capsicum sandboxes. We plan to implement the same sandbox architecture using user space segments. We will note the complexity of the code modifications required for each approach and will then measure how performance scales with the number of sandboxes. We expect that a large number of sandboxes will place undue pressure on the TLB and cause performance collapse in the Capsicum case but that the segment model will scale more closely to the unprotected model. We have done an initial experiment which sandboxes libpng using both techniques. While we have a methodology for measuring performance and at least some methodology for usability, we are still working on a methodology for measuring relative security which will involve formal verification.

4.3 C/C++ Annotations

Segments in our model can be used as general purpose pointers with a limited range of modifications available. David Chisnall has extended Clang and LLVM to accept annotations that implement pointers as segments to ensure they are used according to programmer intent, including bounds checking and read, write, and execute property enforcement. This approach is designed to allow simple security enhancement of existing code bases in C which often have poor security properties. We plan to evaluate usability of the annotations by measuring the number of lines that needed modification for added security benefit compared to pervasive bounds checking, and by attempting to compare the clarity of resulting code.

5 Future Work/Collaboration Potential

Enforcing Program Structure in the Linker

Modern ELF files which are consumed by a linker to incorporate binaries into a running program specify segments of the program along with properties for each segment. It should be possible to build a custom linker which constructs a segment descriptor for each ELF segment and implements memory references as segment loads and stores.

Managed Language Runtimes

Flexible segment hardware should allow managed language runtimes to be simpler, faster, and more secure. A straight forward use of segment registers as pointers and of protected calls on invocations would provide strong hardware protection of managed language objects.

Static Analysis Assistance

Static analysis of computer programs for correctness has proven very promising. If segment manipulation was part of the user space code stream, static analysis may be able to make a breakthrough in reducing proof complexity.

6 Conclusion

We hope to demonstrate that a user space segment model is useful in the context of the modern software stack to enhance the security and correctness of computer programs. We also hope to demonstrate that a user space capability segment model is implementable in a modern RISC processor without breaking the principles of RISC design.

An FPGA implementation of the 64-bit MIPS processor will be available for download this year and we will welcome collaboration with researchers who would like to apply our capability segment model to their own challenges in engineering secure software and systems.

6.1 Acknowledgments

We would like to thank our colleagues - especially Peter Neumann, Jonathan Anderson, Ross Anderson, Nirav Dave, Ben Laurie, Steven J. Murdoch, Philip Paeps, Michael Roe, David Chisnall, Robert Norton and Khilan Gudka.

This PhD work is part of the CTSRD Project which is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

Bibliography

[1] Chess, B.: Improving Computer Security using Extended Static Checking. In: Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on. pp. 160–173. IEEE (2002)

[2] Childs Jr, R., Crawford, J., House, D., Noyce, R.: A Processor Family for Personal Computers. Proceedings of the IEEE 72(3), 363–376 (1984)

[3] Cleveland, S.: x86-64 Technology White Paper. Tech. rep., Advanced Micro Devices (02 2002)

[4] Cowan, C., Wagle, F., Pu, C., Beattie, S., Walpole, J.: Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In: DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings. vol. 2, pp. 119–129. IEEE (2000)

[5] Denning, P.: Virtual Memory. ACM Computing Surveys (CSUR) 2(3), 153–189 (1970)

[6] Dennis, J.B., Van Horn, E.C.: Programming semantics for multiprogrammed computations. Commun. ACM 9(3), 143–155 (1966)

[7] Feiertag, R., Neumann, P.: The Foundations of a Provably Secure Operating System (PSOS). In: Proceedings of the National Computer Conference. vol. 48, pp. 329–334 (1979)

[8] Fotheringham, J.: Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store. Communications of the ACM 4(10), 435–436 (1961)

[9] Frame, A., Turner, C.: Introducing New ARM Cortex-R Technology for Safe and Reliable Systems. Tech. rep., ARM (03 2011)

[10] Gudka, K., Watson, R., Hand, S., Laurie, B., Madhavapeddy, A.: Exploring Compartmentalisation Hypotheses with SOAAP. In: Workshop paper, Adaptive Host and Network Security (AHANS 2012) (2012)

[11] Hansen, P., Linton, M., Mayo, R., Murphy, M., Patterson, D.: A Performance Evaluation of the Intel iAPX 432. ACM SIGARCH Computer Architecture News 10(4), 17–26 (1982)

[12] Houdek, M., Soltis, F., Hoffman, R.: Ibm System/38 Support for Capability-based Addressing. In: Proceedings of the 8th Annual Symposium on Computer Architecture. pp. 341–348. IEEE Computer Society Press (1981)

[13] Levy, H.M.: Capability-Based Computer Systems. Butterworth-Heinemann, Newton, MA, USA (1984)

[14] Needham, R., Walker, R.: The Cambridge CAP Computer and its Protection System. Operating Systems Review pp. 1–10 (1977)

[15] Parrend, P., Frénot, S.: Classification of Component Vulnerabilities in Java Service Oriented Programming (SOP) Platforms. Component-Based Software Engineering pp. 80–96 (2008)

[16] Randell, B., Kuehner, C.: Dynamic Storage Allocation Systems. Communications of the ACM 11(5), 297–306 (1968)

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

[18] Schroeder, M., Saltzer, J.: A hardware architecture for implementing protection rings. Communications of the ACM 15(3) (March 1972)

[19] Watson, R., Neumann, P., Woodruff, J., Anderson, J., Anderson, R., Dave, N., Laurie, B., Moore, S., Murdoch, S., Paeps, P., et al.: CHERI: A Research Platform Deconflating Hardware Virtualization and Protection. In: Workshop paper, Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012) (2012)

[20] Watson, R.N.M., Anderson, J., Laurie, B., Kennaway, K.: Capsicum: Practical capabilities for Unix. In: Proceedings of the 19th USENIX Security Symposium. USENIX (August 2010)