Automation for Creating and Configuring Security Manifests for Hardware Containers

Eugen Leontie, Gedare Bloom, and Rahul Simha
Department of Computer Science, George Washington University
E-mail: {eugen,gedare,simha}@gwu.edu

***Abstract*—Hardware containers provide fine-grained memory access control to isolate memory regions and sandbox memory references between components of an application. A hardware reference monitor enforces a security manifest of memory access permissions for the currently executing component. In this paper we discuss how automation tools help software developers to create the security manifest that configures hardware containers.**

I. Introduction

Modern software developers build systems by combining commercial off-the-shelf (COTS) components with glue code; Basili and Boehm [1] estimate over 99% of instructions executed are from pre-packaged COTS software. Application developers can no longer ensure the security and reliability of their own software without relying directly on the reputation and warranties of COTS software vendors. In addition, applications often allow third-party plugins and extensions that introduce new features to the application at runtime. COTS components and third party plugins are untrusted code that, without any security isolation primitives, can compromise system security.

Commodity hardware and operating system (OS) isolation techniques build on virtual memory primitives (pages or segments) and can isolate untrusted code by wrapping each untrusted component in a different virtual address space, e.g. a process. The application then relies on the OS to mediate communication between components. An example of such an application is Chromium [2], a web browser that isolates web page rendering from the main browser process. A drawback of this approach is the overhead of going through the OS when the active component changes.

Reducing the overhead of OS-mediated isolation is the goal of research in hardware support for fine-grained memory access control. Our prior work introduced hardware containers [3], [4] for fine-grained isolation of code and data that targets code written with a hierarchical control flow, which is normal in imperative, procedural, and object-oriented languages. Containers isolate code at the granularity of functions as a natural boundary for structured components and programs. Fine-grained access control on word-sized data blocks further restrict components from accessing data without permission. A hardware reference monitor enforces a security manifest that authorizes each component with permissions to read and write data in memory and to transfer control flow to other components. In this paper we discuss the software tools that assist application developers in creating and configuring the security manifest for hardware containers.

II. Security Manifest

Each container wraps a set of functions and has strictly-defined rules about the permissible actions and accessible memory regions of its functions. Compile-time tools generate the rules and store them in a security manifest that the hardware reference monitor loads and enforces at runtime. Application developers use these tools to generate memory access permissions (read, write, execute) for three kinds of memory references: static, instance-dependent, and dynamic.

Static Permissions. When processing a function, the compiler identifies and adds permissions to the security manifest for memory accesses to global variables, constants, static values, and code. These permissions are obtained easily during normal compilation and are required for correct execution of the code. In languages without type safety, such as C, arbitrary memory accesses can be made by misusing pointers. Many pointer references and simple pointer arithmetic can be characterized by the compiler into safe, forward-only sequential pointers [5] with precise memory ranges. When the compiler can generate this range it can include permissions in a static access list. Our tools need help to handle complex pointer arithmetic and pointer assignments to arbitrary locations. Software developers can use security annotations to denote the expected ranges and permissions for memory that such pointers should access. Note that requiring annotations is not a limitation of our approach but instead a characteristic of vulnerable software design. Security annotations might be defined through either compiler directives or external configuration files that the linker reads. We are interested in mature software tools to help create security annotations.

Instance-dependent Permissions. Each time a function executes it creates new copies of its local variables. In hierarchical or recursive control flows these variables are stored on a stack. Function stack space requirements and accesses to local variables are known at compile-time, but unlike static memory accesses these instance-dependent accesses are relative to a base address—the frame pointer. The compiler can create access lists relative to the frame pointer that the hardware reference monitor can enforce because it protects the frame

pointer. These access lists are part of the static permissions of a program. Not all instance-dependent memory references can be resolved at compile-time, especially if the mapping between containers and functions is not one-to-one. Such references are instead treated as dynamic memory references and are resolved with programmatic directives.

Dynamic Permissions. Compile-time tools can extract access lists for memory references as described above, but dynamic memory references need special handling of permissions at runtime. A design choice of hardware containers is to bind dynamic memory permissions with each container's invocation rather than its static definition. Dynamic permissions are handled by the hardware reference monitor as a stack in parallel to the program's call stack. New instructions support delegating and revoking permissions on dynamic memory ranges that are passed to a callee or returned to a caller. We are interested in improving our tools to help programmers configure dynamic permissions.

Today's complex software makes it difficult to annotate source code manually and to extract memory access permissions even with compiler help, especially with pre-compiled libraries and object code. For such cases, we propose that execution traces be used to derive permissions. By giving permissions for only tested and approved access patterns, the security manifest avoids unauthorized access.

Another concern is that enforcing fine-grained access control for time-critical code may not be desirable, especially if security can be achieved though other means. In such cases relaxing component granularity by including several functions in one component (container) decreases the overhead related to permission checking.

III. Related Work

Because of the demand for C, compiler assistance for memory protection is an established area for practice and research. A well-known solution is StackGuard [6], which relies on the compiler to add canary values near the return address on the stack and to verify those canary values before returning from a function. Instead of detecting when memory is overwritten, compiler analysis and code generation can enable writing to a memory region only during operations on the memory object in that region [7], but at a high cost of extra instructions every time a memory object is written.

An alternative to protecting memory regions is to ensure that code accesses data safely. Two approaches to support safety in C include Cyclone [5] and CCured [8], both of which change how pointers are used. Both approaches instrument pointers with dynamic bounds checking and garbage collection. Whereas Cyclone limits pointer arithmetic, CCured adds dynamic checking so that pointer dereferences can be verified at runtime in situations that the compiler is unable to resolve safely. Both approaches rely on compiler and programmer assistance to achieve maximum security and performance, but dynamic checking causes performance loss. The compiler techniques that resolve at compile-time many pointer-based references can help to create static permissions for our security manifest; dynamic checks can be converted to dynamic permissions.

An important aspect of security automation is mapping a security policy to an enforcement mechanism. Huffmire et al. [9] demonstrate compiler translation of an access policy into a hardware execution monitor for enforcing policies such as compartmentalization, access control, secure hand-off, Chinese wall, and redaction. This execution monitor is meant for hardware designs with isolated circuitry that communicate through the monitor and is not applicable directly to software applications that share hardware resources, but compilation tools for security policies may benefit our approach.

IV. Future Work

The effectiveness of hardware containers is limited by the ability of automation tools. Compilers can easily generate the permissions for static and stack-local memory ranges, but non-local (pointer) references and dynamic memory currently require manual code annotations. In addition to generating permissions, we are also interested in tools that can map security policies to sets of permissions, validate permissions against a chosen security policy, and translate existing code annotations such as mandatory access control labels (secret/public) or language visibility labels (public/private/protected) into container-enforceable memory access and control-flow permissions.

Acknowledgments

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

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] C. Reis, A. Barth, and C. Pizano, "Browser security: lessons from google chrome," *Commun. ACM*, vol. 52, pp. 45–49, August 2009. [Online]. Available: http://doi.acm.org/10.1145/1536616.1536634

[3] 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 2009 International Symposium on Trusted Computing, TrustCom09*. IEEE, August 2009.

[4] ——, "Hardware-enforced fine-grained isolation of untrusted code," in *Proceedings of the 2009 ACM workshop on Secure Execution of Untrusted Code, SecuCode09*. ACM, Nov 2009.

[5] 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.

[6] 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.

[7] K. Zhang, T. Zhang, and S. Pande, "Memory protection through dynamic access control," *The 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 123 – 134, 2006.

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

[9] T. Huffmire, S. Prasad, T. Sherwood, and R. Kastner, "Policy-driven memory protection for reconfigurable hardware," *Lecture Notes in Computer Science*, pp. 461 – 478, 2006.