Hardware Containers for Software Components: A Trusted Platform for COTS-Based Systems

Eugen Leontie,* Gedare Bloom,* Bhagirath Narahari,* Rahul Simha,* and Joseph Zambreno †
*Department of Computer Science, George Washington University
†Iowa State University, Department of Electrical and Computer Engineering
E-mail: {eugen,gedare,narahari,simha}@gwu.edu, zambreno@iastate.edu

***Abstract*—Much of modern software development consists of assembling together existing software components and writing the glue code that integrates them into a unified application. The term COTS-Based System (CBS) is often used to describe such applications, for which the components assembled are understood to be Commercial-Off-The-Shelf (COTS) components written by a multitude of independent third parties. The manner of assembly in CBS includes full-source components that are integrated at compile-time, pure-binary libraries incorporated at load-time, and plugins that are loaded into the application at execution time by the user. Because components have access to system resources, applications may crash due to faulty components or may be compromised by malicious components. In this paper, we ask the question: can hardware support the development and deployment of CBS by providing applications with a trusted platform for managing components and their interactions? We present an architecture that places each CBS component in a hardware-enforced *container*. The hardware then detects improper usage of system resources (unauthorized memory accesses or denial-of-service) and enables applications to undertake a hardware-supervised *recovery* procedure. Furthermore, the hardware also maintains a violation record to enable developers to recreate the violation for the purpose of debugging and further development. Taken together, the purpose of the architecture we propose is to enable executing untrusted CBS code on trusted hardware.**

I. Introduction

By some estimates [4], over 90% of instructions executed are from *Commercial-Off-The-Shelf (COTS) components*, pre-packaged software components developed independently and meant for use in multiple applications [16]. Such a statistic underscores a sea change in software development over the past few decades: Instead of building applications by writing code from scratch, developers combine existing components together with glue code that manages the interactions between components. Furthermore, many applications allow users to incorporate optional third-party plug-ins or extensions long after the application is deployed. Thus, an application's security and stability is often beyond the control of the application developer since malicious or faulty components may not only crash an application but may actively compromise the security and privacy of other components.

In this paper, we propose a trusted platform supporting COTS-Based Software (CBS) applications. Such a trusted platform enables executing untrusted components in a safe manner by detecting and responding to malicious behavior. To see why such hardware support is desirable, consider the fact that, not so long ago, it was possible for any application to walk all over memory – indeed, it wasn't until the hardware was able to check memory accesses that kernel integrity and process separation were enforceable. Unfortunately, the operating system (OS) *process*, today's unit of isolation, is both too reliant on intervention from the OS, which itself may contain vulnerabilities, and too coarse a unit to help in overseeing the interactions inside a component-built application. For a CBS application, one needs to detect the tampering activity of a malicious component, to isolate a component that has crashed, and to recover and continue operation in systems that must remain responsive.

While our architectural features are intended for general computing platforms, embedded systems can benefit by having automatic recovery without human intervention and can use the framework to constrain the I/O capabilities of software components. The need to restrict a component's I/O will grow in importance as increasing device complexity makes the component style of development more appealing [11]. Also the security of such devices is becoming more important, as hackers have created elaborate software-based techniques to evade elementary security mechanisms. For example, a flaw in the component used by the iPhone and Sony PSP to handle Tiff files has been manipulated by hackers to allow them to run code that was not permitted by the manufacturer [19]. Thus, a trusted platform that supports component-based applications by providing strong isolation and recovery features will be useful in embedded systems.

The main contribution of this paper is a trusted hardware-support framework for component-based development. The hardware features limit a component's

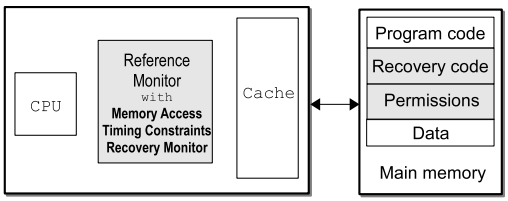

Fig. 1. High-Level Architecture. We add a hardware reference monitor for memory access control that also manages recovery after detecting a violation. Static and dynamic permissions are managed and stored in regular memory by our added hardware.

memory accesses, control data sharing between components, continually monitor each component's execution, and enable hardware-supervised recovery when something goes wrong. In particular, the hardware places each component in a conceptual *container* – think of each component as having its own virtual processor – that provides isolation and memory protection, call-return monitoring, denial-of-service detection, container recovery, and gathering of forensic information. For some of these features, we refine and integrate some ideas in the architecture literature (for example, the memory-protection mechanism in [33] and the notion of recursive recovery in [6]), and for others (denial-of-service, memory-protection of dynamic memory), we devise entirely new solutions. The architectural details are explored further in a separate paper [14].

Our approach differs from related work in that we directly address the need to protect dynamic memory at a fine granularity and we seek to minimize the hardware design necessary to realize our solution. In particular, we place logic outside of the processor pipeline, as shown in Figure 1, and use compiler and language techniques to produce a permission stack that follows the natural activation record of modern programs. Together, the software-generated and hardware-enforced containers provide a robust execution platform for CBS applications. Our evaluation of the overhead incurred by the close monitoring performed in hardware shows a modest 6% average overhead across a suite of benchmarks even with the finest container granularity. The performance impact can only be accurately measured through simulation because the modified code affects caching behavior. The overhead is also caused by looking up access permissions in hardware, assigning dynamic permissions, and conservatively flushing the pipeline to prevent invalid violations triggered by misspeculations.

In addition to hardware support, there are two accompanying software tools implied in our architectural framework: a compiler module that produces the runtime metadata needed by the hardware, and a runtime loader that securely translates symbolic linked modules into pre-specified relative addresses. Neither tool entails any significant modification to existing compilation, but instead constitute modest added steps.

Note that although a Trusted Platform Module (TPM) [29] can be used for integrity checking, its use requires a trusted OS and it is limited to verifying that the code that executes is untampered. The model of verifying code by checking its hash simply places trust in the code's provider or some third party reviewer. In the case of CBS development, this means that every component's provider must be trusted by the application developer. Our solution instead validates the runtime behavior of components by providing strong isolation coupled with hardware-based monitoring. By treating components as black boxes, the developer of a CBS application can focus on component functionality without devoting resources to test the security and reliability of each component used.

The rest of the paper is organized as follows. We begin by defining and describing containers in the next section. Following this, we describe an overview of the architecture in Section III. Section IV describes simulation results that evaluate our platform's performance. We present related work in Section V before concluding in Section VI.

II. How Containers Work: Practicality, Detection, and Recovery

What is a container? A *container* is a unit of isolation supported by the hardware. The unit may be as small as a single byte of addressable memory or as large as an entire application. Thus, a container may encapsulate just data, just code, or most typically both code and data as one may expect with components. The hardware monitors all memory accesses in containers, all jumps into and out of containers, and each container's execution time. In this manner, containers may be used to protect data, to protect code, and to ensure some degree of proper execution. For a language like C++, we envision containers used to encapsulate all dynamically allocated memory, every object, and even individual methods if desired. Of course, a fine granularity carries greater overhead, so developers can choose instead to only "containerize" large components such as libraries.

A developer's perspective. An important consideration for our platform is its practicality, in particular how containers affect application developers. Much of the development process remains the same – a developer assembles components and writes the glue code as before. However, just prior to deployment, the developer will use our modified compiler tools to generate an application *manifest* that identifies components and specifies constraints on their behavior. To the extent recovery is desired, the developer will need to write or identify

recovery code. Also, some code annotation by the developer may be helpful to manage (more efficiently than the compiler) the fine-grained access control mechanisms, which allow data sharing between components at any level of granularity.

A hardware and OS perspective. From a hardware perspective, much of the hardware functionality is performed by a module inserted between the standard CPU and cache[1]. Thus the processor is mostly unmodified. A standard OS can be used with only one modification – for a context switch, metadata related to containers needs to be swapped, similar to switching page tables. Naturally, the hardware can also be used for isolating parts of the operating system such as drivers, which requires recompiling those parts of the operating system. Note that the handling of container-related events is almost entirely done in hardware and is not OS specific, and can even be used in embedded platforms without an OS.

Detection of violations. Whether malicious or not, we identify the following violations: illegal inter-component read, illegal write to read-only data, buffer overflow, illegal code entry or return point, denial-of-service, and faulty recovery. An illegal inter-component read occurs when one component reads memory owned by another, for example a bad pointer, which is identified as an out-of-container access by the hardware and is immediately trapped. Similarly, an illegal write to read-only (or permissionless) data is easily detected in hardware if software has appropriately set permissions for read-only objects. Buffer overflow, a common bug and popular exploit [1], is prevented by enforcing memory bounds checking in the hardware. A jump or return to invalid code, as determined by the metadata stating which code a container can reach, is detected by the hardware. To detect denial-of-service, upon entry to a container the hardware starts an instruction counter that is compared against a maximum value specified by the metadata. Finally, recovery is another component which is potentially faulty or maliciuos, thus the hardware also monitors recovery, allots fixed timeouts for each component's recovery process, and ensures each affected component is given a chance to recover.

Recovery. At this point, it is worth asking: what kinds of actions are typical in recovery? We envision three broad types of recovery actions: intra-component recovery, sub-component instantiation, and application restart. Transient faults within a component might be rare and managed by simply allowing a component to restart itself. Thus a component writer could set up intra-component recovery by re-initializing data structures, clearing internal buffers, or performing integrity checking of persistent data stores. If a component is consistently failing, its caller can re-instantiate it or replace it with an alternate solution, perhaps a library with a compatible interface or an older, more stable but slower version of the faulty component. This sub-component instantiation can stabilize a system from persistent faults with runtime loader support for dynamic loading and re-instantiation of components. Finally, if the chain of recovery reaches the main component, it may decide to inform the user and exit gracefully or attempt to reset the entire application. Writing a well-organized recovery procedure is not a simple task. However, once such recovery code is written, the hardware can help ensure that the recovery proceeds in an orderly and safe manner.

[1]As in software development, hardware design today partly consists of putting together blocks, so-called cores, created by third-parties. Our proposed module can be designed as one such separate core.

III. Architecture Overview

The effort for creating our dependable, secure execution environment lies on two fronts. On the software engineering front are extraction of program properties and writing of recovery code that satisfies the developer's security policies. On the computer architecture front, we design a trusted execution platform that validates fine grained memory access, checks control flow, and enforces an orderly recovery procedure.

In traditional software development, 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 application developer goes through a few additional steps. *Step 1: Create an application manifest*. The manifest consists of all the information needed by the hardware – a list of components, their identifiers, memory permissions, permitted call patterns, and approximate run times (in terms of the number of instructions executed) for selected methods of interest. Much of this information is compiler-generated. However, to estimate running times, the programmer would need to use a profiling tool or execute the entire application in a debugger. Memory regions dynamically allocated at run-time are identified to the hardware together with the access control metadata for sharing with other containers. To expose this metadata at the language level, we define a primitive termed ***ALLOW*** that the compiler and programmers can use for passing a memory range and associated permissions to other containers. This primitive can be implemented either with an addition to the instruction set or memory mapped operations. *Step 2: Write recovery code*. The developer writes small chunks of recovery code for each component for which recovery is desired. This code would be used for re-initializing the component's state or replacing a persistently faulty sub-component with an

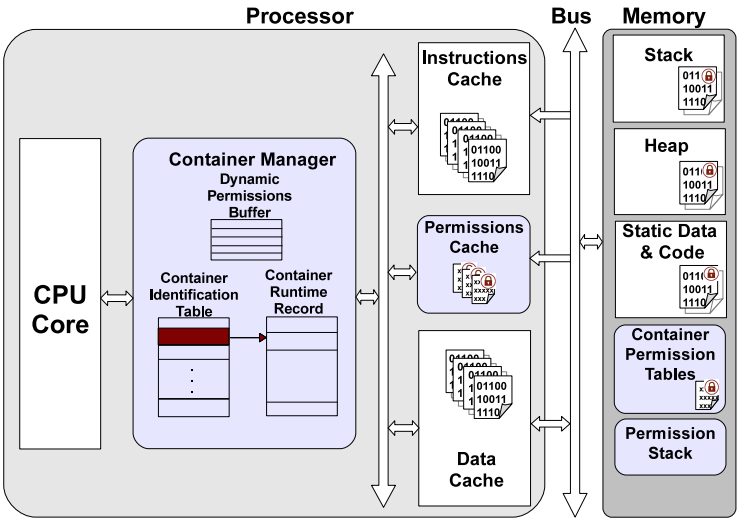

Fig. 2. System Overview. The Container Manager interposes on memory accesses so that access control can be enforced efficiently. Each component is uniquely identified by a Container Identifier and is associated with a Container Runtime Record (CRR) that stores static access permissions, which are loaded from the Container Permission Tables in main memory. For each executing container, the Dynamic Permissions Buffer stores permissions to dynamic memory, for example heap-allocated buffers. These dynamic permissions are evicted to the Permission Stack on a container switch. Static and dynamic permissions are stored in the cache hierarchy within a distinct Permissions Cache.

alternate solution, if available. The application developer can write such recovery code or may require component developers to provide it for their components. *Step 3: Build the application*. The entire application now consists of all the executable code and static data along with the manifest. The manifest is shipped with the executable. Further involvement of the developer occurs only when forensic data are retrieved after a violation occurs, to inform the next round of development.

The compiler tool-chain creates the manifest for each application. This manifest consists of one *Container Runtime Record (CRR)* for each container. It is the CRR that has the needed runtime information and is loaded into the container manager when a container executes. Metadata within a CRR depends on the type of data or code being accessed. For the *code* section, the compiler generates a list of static (global) data ranges that the container needs to access. These ranges, as well as the size of the code, are stored as the *static permission table* in the CRR. Read and writes to memory are validated against a list of permissions to memory ranges, expressed as a tuple *(start_address, end_addresss, access)*. In addition to memory ranges, the compiler generates all combinations of jumps and jump targets, so that only valid control flow is permitted by the hardware. For optional detection of denial-of-service, each container can be assigned a maximum execution time, counted in cycles obtained by profiling, that the hardware checks by setting a timer. Also, access permissions to *stack data* are extracted by the compiler, as offsets from the stack pointer, and are added to the CRR. For every *heap*-allocated block, the compiler instruments the code with the ALLOW primitive to enable functions to pass or return data safely. These ALLOW statements add dynamic permissions to the otherwise pre-computed CRR.

The primary task of our added hardware is to check that every memory access from a container is within the statically permitted ranges, within the stack bounds, or within permitted ranges of the heap. For this task, the hardware needs to look up permitted ranges quickly. This search is achieved through the use of content-addressable memory (CAM). CAM provides a fast parallel search for a finite amount of data, and is most well-known for its use in the translation lookaside buffer, or TLB, used in commodity systems for speeding up virtual to physical address mapping. The hardware also needs to check each jump and jump-target, and the number of cycles allocated to each function invocation. Figure 2 shows the placement of our added hardware, which we call the Container Manager. The Container Manager automatically identifies *container context switches* between the defined containers based on instruction fetching, loads the appropriate permission state from memory, and validates each memory access. Three CAM accelerated tables maintain container context. The *Container Identification Table* lists all code regions in a process and their mapping to individual containers. The *Container Run-Time Record Table* allows memory permission checking, timing and control flow for the currently executing container. Dynamically allocated memory ranges using *ALLOW* are transferred through the *Dynamic Permission Buffer*. A few architectural optimizations are necessary

TABLE I
DETECTION TIME IN CYCLES FOR VULNERABLE PROGRAMS

Application	Version	Buffer Size (bytes)	Cycles to Detect (SW stack guards)	Cycles to Detect (HW containers)
`villistextum`	2.6.6	32,768	86,108	[10-20]
`ringtonetools`	2.22	1,024	119,261	[10-20]
`ringtonetools`	2.22	1,024	119,261	[10-20]
`mplayer`	1.0pre5	102,400	99,465	[10-20]
`csv2xml`	0.5.1	1,000	94,828	[10-20]
`2fax`	3.0.4	256	6,129,857	[10-20]
`bsb2ppm`	0.0.6	1,024	92,206	[10-20]
`jpegtoavi`	1.5	4,096	3,016	[10-20]
`o3read`	0.0.3	1,024	109,332	[10-20]

for critical operation speedup. The loading of the container permission state is accelerated by a dedicated permission cache, which helps reduce the fetch delays and limits the memory bus load. Clearly, the performance overhead depends strongly on how range checking is performed for every memory access. For range checking, we leverage ternary CAM (TCAM) designs, which allow for matching "don't care" bits, thus enabling efficient range-checking [27]. Recent work on fast CAMs for range-checking has shown that the speed of the range-checking is on par with regular CAM lookups, which can be achieved in a single cycle [22]. Further details of our architecture design and optimization techniques are addressed in a separate paper [14].

IV. EXPERIMENTAL RESULTS

We show in this section two aspects of our experimentation. The first is a comparison of the detection time for real world vulnerabilities in a software only protection mechanism versus our hardware accelerated access monitor. The second is an example of the performance overhead for our reference monitor using a minimally invasive, unoptimized architecture.

A. Functional Evaluation

For our comparative analysis, Table I shows eight vulnerabilities found in real-world applications, as described in US-CERT Cyber Security Bulletin SB04-357 [9]. For each vulnerability, we compare the time between exploit and detection using our approach versus a software approach such as StackGuard [30], under the assumption that the software approach itself has no overhead. The last two columns illustrate the difference: Our hardware approach can detect an improper access within a few cycles whereas a software approach takes many orders of magnitude more, even when the best case is assumed for software and the worst-case assumed for hardware. The reason for such a gap is that software approaches need to wait at least until the moment the victim function is about to execute a return. Most software approaches wait even longer, until the runtime system makes a check after a function return. In contrast, our approach detects any unauthorized memory access as it is committed to memory, allowing for small delays in detection due to micro-architectural design choices. This detection can occur as soon as a few cycles, or occasionally take 10-20 cycles because the memory check information needs to be loaded into the CAM, as described earlier. In general, the additional speed possible in hardware is not surprising. What this speed implies, however, is significantly more time for recovery, which could be crucial in many embedded applications.

B. Performance Evaluation

In order to evaluate the performance overhead, we extended the SimpleScalar [3] simulation suite to accommodate measurements for our architectural modifications and memory access profiling. We used the characteristics of an ARM processor running at 400Mhz with an external bus and main memory at 100 MHz. The performance of our architecture was observed for a memory hierarchy that contains one level of separate instruction and data caches. For the baseline system the instruction cache has 32Kb of 32-way associative memory and the data cache is a 32Kb, 64-way associative cache. These architectural parameters are typical of an embedded system.

The benchmarks chosen for the simulations were computationally-intensive applications from MiBench [12]. In addition, we selected three data intensive benchmarks from the Data Intensive Systems (DIS) benchmark suite [18], and a heap intensive benchmark known as the Richards benchmark [24].

Figure 3 shows the overall performance penalty as the difference in execution time measured in processor cycles between our architectural modifications applied to the ARM processor and the unchanged ARM core. The executables for the augmented architecture are fully instrumented with instructions that handle dynamic mem-

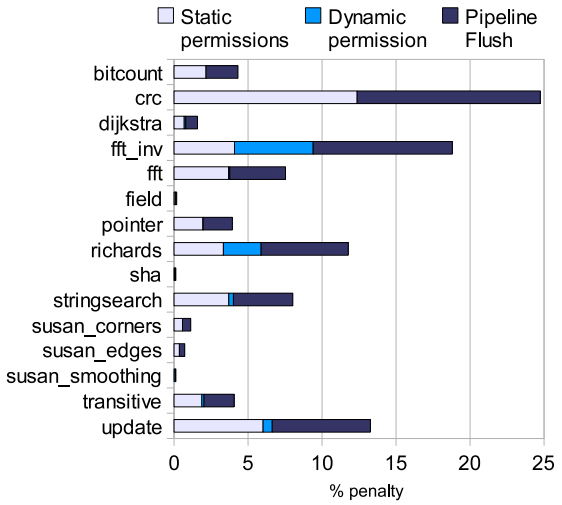

Fig. 3. Performance Overhead. Overhead added by code changes made to support containers and by architectural modifications.

ory permission assignment. This change adds to the length of the code, affects caching behavior, and uses additional processing cycles. As part of instrumenting the benchmarks, we also modified the C library (`glibc`), which is a major part of the functional code in the statically compiled benchmarks. Architecture details such as the internal organization of the pipeline, speculation, and out-of-order execution have a big impact on the performance of a memory reference monitor in general. In order to offer a fair and uniform evaluation on a wide range of processor architectures, we flushed the processing pipeline on every container context switch. While flushing every switch is a conservative approach, it can be easily applied to both simple in-order and more complex speculative out-of-order architectures.

The container granularity was made as fine as possible: *Every function and every heap object was assigned its own container*. This code organization allows for precise isolation and containment of faults, yet it represents the worst possible overhead, with frequent container context switches and many memory accesses for fetching the container permission records. The results show moderate additional overhead with a 5.97% average across all benchmarks and a maximum of 24.74% for the CRC program, which has a very high function call frequency (2.75 function calls per 100 processor cycles). More detailed performance results can be found in [14].

V. Related work

As mentioned earlier, our architecture and approach refines existing ideas (memory protection) in the architecture literature with new solutions (hardware-supported recovery, denial-of-service). The prior work on memory protection, mostly process-based, is modified to suit the needs of components and extended to handle dynamic memory efficiently.

We categorize the most-closely related work into projects that feature architecture-based protection or similar containment-related ideas in other areas such as operating systems. The idea of protecting memory goes back to the 1970's, with some protections available in older mainframes [17], [32]. Recently, several projects [2], [10], [25], [26], [33], [36] have taken up this line of work and proposed architectural mechanisms to check and protect access to memory, starting with the seminal Mondrian architecture by Witchel et al. [33]. The central difference between these and our project, is that all these efforts have focused on memory protection alone and do not consider either recovery or denial-of-service, or the wider issues surrounding component-based development. Even within the narrower realm of memory protection, our approach differs in several ways. First, the Mondrian approach targets processes whereas ours targets objects as small as individual functions. Second, we avoid passing permissions for dynamic ranges through the operating system by keeping the context state in the hardware guard. Perhaps more significantly, our approach to handling dynamic memory uses a stack-like approach so that permissions are not persistent but aligned with the natural activation and exit of running code. Another related project that features memory protection is the Infoshield project [25], [26]; however, their approach focuses on a limited set of dynamically assigned permissions aimed at protecting only certain crucial elements (keys, passwords) in the application.

Some operating systems projects such as the Nooks or Janus projects [28], [31] have explored execution-sandboxes with a narrower focus for operating systems drivers [28] or for preventing unauthorized system calls [31]. Similarly, Kiriansky et al. [13] describe how executables can be run in an interpreter, which can then check memory accesses. The Solaris-10 release (2006) [15] has address-space containment for large applications to facilitate server virtualization, intended for isolating large web service applications on a single machine. Yet other hardware-related projects, such as Smashguard [21], [34], [35] have studied hardware modifications for specific types of attacks.

Our focus on recovery is inspired by the Recovery-Oriented Computing (ROC) framework by Patterson et al. [5], [6], [23]. However, their work primarily concerns bugs in large server-based systems. Indeed, their work in the hardware support area [20] describes their ROC-1 hardware platform, a large scale cluster system designed to provide high availability for Internet service applications. Our recovery process goes further than that of [6],

[7], [8] in that we use hardware to additionally guarantee that recovery code executes in a fixed amount of time, enforces the recursion, and is itself protected.

VI. Conclusions and Future Work

In this paper, we have described useful hardware support for COTS-Based Systems (CBS), a major software engineering paradigm today. Application developers are currently forced to accept and trust components, but with the proposed hardware support they can integrate components without worrying about compromised privacy, undetected vulnerabilities, or significant disruption. The framework also facilitates recovery in the aftermath of a violation and, because a system snapshot is stored, also provides valuable debugging information for the next cycle of development. Future work will address hardware design issues and protection for I/O and networking.

Acknowledgments

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

References

[1] Aleph One, *Smashing the stack for fun and profit*, Phrack, vol.7, no. 49, Nov. 1996.

[2] D. Arora, S. Ravi, A. Raghunathan, N. Jha. Architectural Support for Run-Time Validation of Program Data Properties. *IEEE Trans. VLSI systems*, Vol. 15, No. 5, May, 2007.

[3] T. Austin, E. Larson, and D. Ernst. *Simplescalar: an infrastructure for computer system modeling*, Computer (Feb 2002).

[4] V. Basili and B.Boehm. COTS-Based systems top 10 list. *IEEE Computer*, Vol.34, No.5, pp.91-95, 2001.

[5] A. Brown and D. Patterson. Embracing Failure: A Case for Recovery Oriented Computing (ROC). *Proceedings of the High Performance Transactional Processing Symposium*, 2001.

[6] G. Candea and A. Fox. Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2001.

[7] G. Candea and A. Fox. Crash-Only Software. *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2003.

[8] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot: A Technique for Cheap Recovery. *Proceedings of the International Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[9] CERT. US-CERT cyber security bulletin sb04-357. Available at http://www.uscert. gov/cas/bulletins/SB04-357.html, Dec. 2004.

[10] M. Corliss, E. Lewis, and A. Roth. Using DISE to Protect Return Addresses from Attack. *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus*, October 2004.

[11] C. Ebert and C. Jones. Embedded Software: Facts, Figures, and Future, *Computer*, vol. 42, no. 4, pp. 42-52, April, 2009.

[12] 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).

[13] V. Kiriansky, D. Bruening and S. Amarasinghe. Secure Execution via Program Shepherding. *Proceedings of the USENIX Security Symposium*, 2002.

[14] E. Leontie, G. Bloom, B. Narahari, R. Simha, and J. Zambreno. Hardware-enforced fine-grained isolation of untrusted code. *in preparation*, 2009.

[15] C. Mackinnon. Solaris Containers: A Breakthrough Approach to Virtualization. *Processor Magazine*, Vol. 28, No. 8, p. 33, February 2006.

[16] M. Morisio and M. Torchiano. Definition and classification of COTS: a proposal, *Int. Conf. Composition-Based Software Sys. (ICCBSS)*, Feb, 2002.

[17] L.M. Molho. Hardware Aspects of Secure Computing, *Proc. Spring Joint Computer Conf.*, pp.135-141, 1970.

[18] J. Musmanno. Data Intensive Systems (DIS) Benchmark Performance Summary. AFRL Technical Report AFRL-IF-RS-TR-2003-198, 2003.

[19] Open-source Vulnerability Database (OSVDB), reference IDs 38527 (Apple iTouch / iPhone TIFF Image Handling Privilege Escalation) and 19665 (Sony PSP Photo Viewer TIFF File Overflow).

[20] D. Oppenheimer, A. Brown, J. Beck, D. Hettena, J. Kuroda, N. Treuhalt, and D. Patterson. ROC-1. Hardware Support for Recovery-Oriented Computing. *IEEE Transactions on Computers*, Vol. 51, No.2, pp. 100–107, February 2002.

[21] H. Ozdoganoglu, T. Vijaykumar, C. Brodley, B. Kuperman, and A. Jalote. Smashguard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. *IEEE Transactions on Computers*, Vol. 55, No. 10, pp. 1271–1281, October 2006.

[22] K. Pagiamtzis and A.Sheikholesami. Content-addressable memory (CAM) circuits and architectures: a tutorial and survey, *IEEE J. Solid-State Circuits*, pp. 712-727, 2006.

[23] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques and Case Studies. Technical Report CSD-02-1175, University of California, Berkeley, 2002.

[24] The Richards Benchmark.`http://research.sun.com/people/mario/java_benchmarking/`.

[25] W. Shi, J. Fryman, G. Gu, H-H. Lee, Y. Zhang, and J. Yang. InfoShield: A Security Architecture for Protecting Information Usage in memory. *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2006.

[26] W. Shi, C. Lu, and C. Lu. Memory-centric Security Architecture. *High Performance Embedded Architectures and Compilers*, Barcelona, Spain, November 17-18, 2005.

[27] E. Spitznagel, D. Taylor and J. Turner. Packet Classification Using Extended TCAMs, *Proc. IEEE Int. Conf. Network Protocols (ICNP)*, 2003.

[28] M. Swift, B. Bershad, and H. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Transactions on Computer Systems*, Vol. 22, No. 4, 2004.

[29] Trusted Computing Group. `http://www.trustedcomputing.org`.

[30] P. Wagle and C. Cowan. Stackguard: Simple Stack Smash Protection for GCC. Proceedings of the GCC Developers Summit, pp. 243256, 2003.

[31] D. Wagner. Janus: An Approach for Confinement of Untrusted Applications. Technical Report CSD-99-1056, UC Berkeley, 1999.

[32] W.H. Ware. Security and Privacy in Computer Systems, *Proc. Spring Joint Computer Conf.*, Vol. 30, pp.287-290, 1967.

[33] E. Witchel, J. Cates, and K. Asanovic. Mondrian Memory Protection. *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[34] X. Zhuang, T. Zhang, H-H. Lee, and S. Pande. Hardware Assisted Control Flow Obfuscation for Embedded Processors. *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, September 2004

[35] X. Zhuang, T. Zhang, and S. Pande. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2004.

[36] K. Zhang, T. Zhang, and S. Pande. Protection through Dynamic Access Control. *Proc. 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp.123-134, 2006.