Poster: PODARCH: Protecting Legacy Applications with a Purely Hardware TCB

Shweta Shinde[1], Shruti Tople[1], Deepak Kathayat[2], Prateek Saxena[1]
National University of Singapore[1], IIIT Hyderabad[2]
{shweta24, shruti90, prateeks} @comp.nus.edu.sg, deepak.kathayat@students.iiit.ac.in

***Abstract*—Secure execution of applications on untrusted operating systems is a fundamental security primitive that has been challenging to achieve. In this work, we propose a new architecture feature called PODARCH, which makes it easy to import executables on an OS without risking the target system's security or the execution of the imported application. PODARCH is implemented as a backwards-compatible extension to the Intel x86 ISA, and overall, offers strong compatibility with existing applications and OSes beyond those offered by several existing architectural primitives (e.g., Intel SGX). We present a complete system implementation of a PODARCH CPU, the associated toolchain and a modified Linux OS and find that the adaption effort requires 415 lines of code change to the Linux kernel. Thus, PODARCH offers a new design point in the space of architectural primitives that commodity CPU designers can consider in the emerging security extensions to their ISA.**

I. INTRODUCTION

Current CPUs separate software stacks in distinct privilege rings for security and isolation. Users often encounter a situation where they need to perform security-sensitive computation on an untrusted operating system or software stack — for example, a SSL or SSH server on vulnerable web servers, a face recognition computation running on an untrusted cloud server, or even an encrypted user-level file system client running on a borrowed untrusted laptop device. Executing such applications with strong isolation guarantees, even in a hostile or compromised legacy OS environment is an important security problem, which motivates rethinking existing architectural support for such a primitive. To address this concern, we seek a security primitive that allows an application to load, execute and terminate itself on a potentially compromised OS.

We propose PODARCH— a new architectural primitive that excludes all other software code, but for the sensitive application itself, from the trusted computing base (TCB). Our design takes into consideration several desirable primitives like scalability, portability, compatibility with legacy OS and carefully minimizes the assumptions, eliminating the hypervisor, to achieve a design that can be implemented completely in the CPU with zero software TCB. PODARCH introduces the concept of *pod*, a virtual execution environment for the user-level application. Pod applications are standard x86 user-level programs that are guaranteed to execute in an isolated virtual environment. Converting existing ELF x86 applications to PODARCH-compliant executables is straight-forward and requires no developer or user involvement. PODARCH architecture allows backward compatibility with legacy applications and commodity OS by supporting copy-on-write, demand-paging, invoking kernel level system calls, process memory management, context switches, scheduling, interrupt and exception handling. Our source code and extended report for PODARCH is available online [1].

Prev. Work	HW Attacks	OS Attacks	OS Compat	Hyp $\notin$TCB	Applicaton Portability
XOM	✓			✓	✓
AISE	✓				✓
OverShadow [2]		✓	✓		✓
SP3		✓	✓		✓
Bastion	✓	✓	✓		
HyperWall			✓	✓	✓
SecureME	✓	✓	✓		✓
Intel SGX [3]	✓	✓		✓	
PodArch	✓	✓	✓	✓	✓

TABLE I. RELATED WORK. COL. 2-6 DENOTE IF SOLUTION IS SECURE AGAINST HARDWARE ATTACKS, OS ATTACKS, MAINTAINS COMPATIBILITY WITH EXISTING OS, DOESN'T INCLUDE HYPERVISOR TCB AND THE EASE OF PORTING.

II. PROBLEM & CHALLENGES

Consider the scenario where Alice wants to read her CV from an encrypted file system and send it to a network printer. Alice borrows Bob's laptop to access and decrypt her CV before printing it. She does not trust the software running on Bob's machine which can be a compromised OS running a malware. To address Alice's problem, we can use a portable encrypted filesystem application — a user-level encrypted filesystem along with file utilities, which can be carried on a portable device such as a USB stick or imported over the network. We need a secure execution primitive to solve the problem of safeguarding Alice's files and application key from Bob's operating system. Following are the challenges and goals of such a solution:

Secure Execution vs. Detection. The secure execution primitive is meant to execute an application in ring 3 on a compromised OS, not merely serve to detect a hostile OS. The secure execution should not rely on the OS to boot up in a clean state.

Unprivileged Execution. We aim to run the application in ring 3, not below or within the OS (ring 0) as is common in VM introspection techniques. The secure execution primitive should not allow the application to have unfettered access to resources (e.g., physical memory, execution time slices).

Small TCB & No Software Trust. The additional logic to the application should be small, typically increasing the code size by a fixed small amount. Further, the primitive should require no software component outside the application binary to be trusted or to be verified.

Compatible with Legacy Applications and OS. Our goal is to keep the OS and process execution semantics as unchanged as possible, making it easy for legacy applications and OS to be deployed on the primitive. Further, we aim to maintain backwards compatibility to the ISA, i.e., other legacy applications (unmodified) can run in parallel with the secure execution enabled application. Our solution differs from SGX both from a conceptual perspective such as memory management and other features [3], offering better compatibility with legacy OSes and executables while achieving the same level of security. Refer to the full report for a detailed comparison [1].

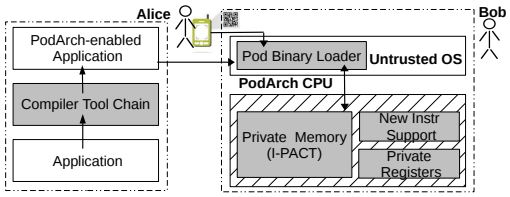

Fig. 1. PODARCH Design. The shaded regions indicate the modifications to the existing system. Only the hatched region is trusted.

III. SOLUTION OVERVIEW

PODARCH introduces the abstraction of a *pod* — a standard x86 process which acts as a unit of isolation at ring 3 (user-level) from all other software at all privilege levels. Each pod binary is encrypted with a distinct application secret key, which we refer as k_{app}. At runtime, this key acts as a separate principal isolated from other mutually untrusting principals — the OS, other pod and non-pod processes. PODARCH combines techniques of on-demand encryption (or memory cloaking [2]) with a set of new security invariants to implement checks completely in hardware. Rather than requiring any strict partitioning of physical memory (as in SGX [3]), PODARCH allows the OS to allocate arbitrary physical pages, possibly non-contiguous, to pod's virtual address space. PODARCH tags such physical pages as "owned" by the pod principal. If OS accesses these physical pages or swaps them out, they are encrypted on-demand using authenticated encryption and are decrypted lazily only when the legitimate pod accesses them in ring 3. This mechanism keeps compatibility with the OS's demand paging and memory allocation algorithms. Further, the PODARCH binary holds metadata which is used by the PODARCH CPU during the life cycle of the pod which includes key setup, loading, execution, entry, exist, termination. Specifically, PODARCH enforces a set of critical invariants for control flow transfers between the pod and kernel, page permission control, virtual address to page content bindings, page integrity protection, and distinctness of virtual-to-physical mappings to carefully protect against illegal access by the untrusted OS. The CPU reserves only a few MBs (for 1000 processes) of CPU-private physical memory to store its data structures called I-PACT (See Figure 1). The rest of metadata such AES-GCM integrity tags are stored in each pod within its virtual address (VA) space, thereby piggybacking on standard OS paging mechanisms for memory management. We guarantee secure execution with following security invariants:

SI-1: Secure Loading. CPU always executes an application loaded by the OS after verifying if the virtual address layout is consistent with that specified by the pod application.

SI-2: Secure Sharing between pod and OS. A pod can securely share data with OS or other pods only through virtual address space specified as public by that application.

SI-3: Sound Authority Tracking. PODARCH private register $\texttt{Reg}_{CEA}$ always tracks the current executing authority k_{app}.

SI-4: Single Owner. A physical page is mapped only to a single pod-owner identity.

SI-5: Safe Access of Resources. Any identity other than k_{app} always accesses physical resources of k_{app} in encrypted form.

SI-6: Distinctness. Every physical page is distinctly mapped to only a single virtual page.

SI-7: Secure Vectoring. The instruction executed after a context switch must be a valid VA defined by k_{app}.

IV. IMPLEMENTATION & EVALUATION

We implement a single core CPU prototype of PODARCH in QEMU,modify the Linux Kernel v3.2 by adding 415 LOC to support PODARCH. We modify the GNU GCC v4.6.3 to create PODARCH- enabled x86-64 applications.

Expressiveness & Porting Efforts. PODARCH executes all the applications that can be statically compiled and do not fork new processes. We port 12 SPEC CINT2006 and 50 CoreUtils applications. Our system transforms and executes them on PODARCH with no developer effort.

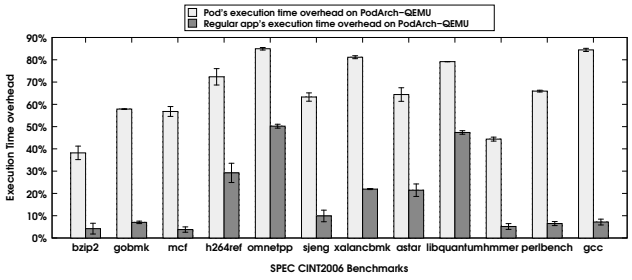

Fig. 2. Execution time overhead for pod-enabled & vanilla SPEC CINT2006.

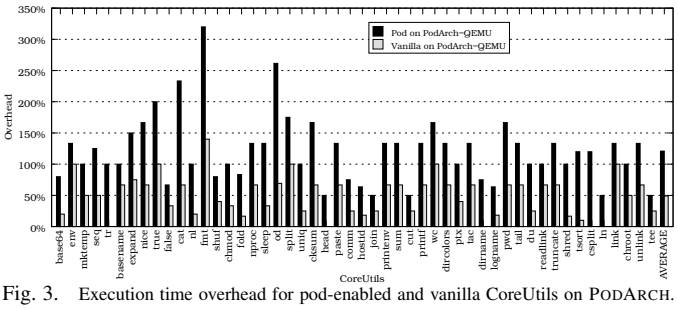

Fig. 3. Execution time overhead for pod-enabled and vanilla CoreUtils on PODARCH.

Robustness & Performance. The average increase in the execution time for the modified benchmarks as reported by QEMU is 66.07 % with negligible memory overhead (see Figure 2, 3). It is faster than the existing hypervisor-based solution OverShadow and comparable to SGX-based solutions [2]. The main overhead factors are metadata checks, cryptographic operations (encrypt and decrypt) for protection of code and data pages, and system call wrappers. The performance is only indicative since we have build a software prototype of CPU rather than a hardware implementation. To test the scalability of PODARCH, we launched multiple instances of SPEC CINT2006 benchmarks processes in 100 parallel processes. PODARCH implementation supports them without any hardware memory or register limitation.

In summary, PODARCH design is backward compatible to applications and OSes thus making it an appealing solution.

ACKNOWLEDGMENT

This research is supported in part by the Ministry of Education, Singapore under Grant No. R-252-000-495-133 and in part by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

REFERENCES

[1] "PodArch." http://www.comp.nus.edu.sg/~shweta24/podarch, 2015.

[2] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems," in *ASPLOS*, 2008.

[3] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *HASP*, 2013.