TECHNICAL REPORT

SECURITY LAB, SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

PODARCH: Protecting Legacy Applications with a Purely Hardware TCB

Shweta SHINDE, Shruti TOPLE, Deepak KATHAYAT and Prateek SAXENA

NUS-SL-TR-15-01
February 6, 2015

PodArch: Protecting Legacy Applications with a Purely Hardware TCB

Shweta Shinde, Shruti Tople, Deepak Kathayat, Prateek Saxena
School of Computing, National University of Singapore
{shweta24, shruti90, deepakka, prateeks}@comp.nus.edu.sg

***Abstract*—Secure execution of applications on untrusted operating systems is a fundamental security primitive that has been challenging to achieve. In this paper, 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 can be 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. For example, modern OSes use architectural features to separate themselves from malicious applications [5]. However, recently, this one-way isolation has come into question, making the OS the Achilles heel. Several traditional attacks such as buffer overflows and memory corruption in the OS can subvert it after which the malware gains unrestrained access to the sensitive application data in the memory. Users often encounter a situation where they need to perform security-sensitive computation on an untrusted operating system or software stack. There are several examples of such computation — 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. In these applications, the sensitive user-level application needs to be protected during execution from the co-resident malware application or the OS. 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 call such a primitive a *secure application execution* primitive. Several previous works have proposed hypervisor-based mechanisms to securely execute sensitive applications [20], [27], [47]. Instead, in this paper, 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. Though previous works offer various architectural solution in this space [17], [20], [21], in our work, we provide a complete end-to-end system with modifications to the kernel, compiler tool chain and ELF loader to securely execute an application in presence of an untrusted OS. Our solution supports transformation of any legacy applications to execute on PodArch using our modified tool chain with zero developer effort. PodArch retains compatibility with the commodity OS by supporting copy-on-write, demand-paging, invoking kernel level system calls, process memory management, context switches, interrupt and exception handling.

Our solution offers a new point in this design space, specifically: no trusted software TCB, good compatibility with commodity Linux system and process semantics, no assumptions about requiring a secure boot or remote attestation and zero developer effort in adapting several existing applications. PodArch retains the resource provisioning capacity with the OS while delegating the security of the applications to the trusted CPU. Thus, PodArch keeps the semantics of the virtual memory management, system call handling, exception handling, scheduling and process management largely unchanged, which makes it easy to adopt PodArch in commodity operating systems. Further, we provide a detailed security analysis of our design by identifying several avenues of subtle attacks which are protected by our design.

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. That is, a simple binary rewriting or compilation step (integrated in our PodArch compiler toolchain) performs this conversion. Designing PodArch as an extension to existing Intel x86 architecture allows backward compatibility with legacy applications. Our source code for PodArch is available online [7].

Purely architectural primitives for secure application execution are on the rise. Intel has recently proposed, independently of our work, a related primitive called SGX. We explain how our solution differs from SGX both from a conceptual perspective and in details, offering better compatibility with legacy OSes and executables while achieving the same level of security. We present a comprehensive security analysis of our solution. We implement PodArch in the MMU of QEMU x86-64 emulator [8] and benchmark PodArch implementation on Linux v3.2, running SPEC CINT2006 [26], HBench-OS [16] and CoreUtils [2] applications which demonstrates high compatibility. The average performance overhead is 66.07 % for SPEC CINT2006 on QEMU compared to 70-100% overhead reported by previous work [20]. PodArch's performance can be aggressively optimized in a full CPU-based deployment.

To summarize, we make the following contributions:

- We introduce PodArch, a new CPU architecture which fulfills the desirable secure primitives along with achieving compatibility and scalability with legacy applications and OS by trusting only the underlying hardware.
- We build a complete commodity Linux system on PodArch with changes to the CPU, OS and compiler tool chain including a binary rewriter and ELF loader. Our system is open source and available at `http://bitbucket.org/podarch`.
- We demonstrate the usage of PodArch system for 12 SPEC CINT2006, 18 HBench-OS stress test benchmarks and 50 case studies from Unix CoreUtils package. Porting these applications using PodArch toolchain requires zero developer effort.

II. Problem & Our Approach

A. Security Objectives

Secure application execution is useful for security-sensitive applications that need to protect themselves from a hostile OS, but which shouldn't be given access to OS state directly. Thus, providing a two-way isolation between applications and OS. Consider the scenario where Alice wants to read her CV from an encrypted file system and send it to a network printer. The encrypted file system can be downloaded in encrypted form from a cloud storage service or transferred via a USB device. 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. This malware can steal her secret key for the encrypted file system and decrypt all her files. On the other hand, Bob may not trust Alice's sensitive application to run in privileged ring (ring 0) because it can compromise his OS or other sensitive applications by running malware, or worse, consume all the system resources.

To address Alice's problem, we envision a portable encrypted filesystem application (PodFS) — 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 solution to solve this dual problem of safeguarding Alice's files, application key and Bob's operating system from one another. Our solution assumes a *trusted hardware, but an untrusted OS* [1] model, i.e., we assume that the adversary doesn't mount hardware attacks (e.g., cold-boot attacks [25], JTAG debuggers [10]) on the target device, but prevent any malicious software or OS from compromising the sensitive application. Our solution provides both integrity and confidentiality for the sensitive application's code and data memory throughout its execution, while maintaining standard process semantics available on commodity OSes today. The security goals and scope of our solution are described below.

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. For example, if the OS is not in a known white listed state at the time of application execution, the application should still run with isolation guarantees. 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). The OS should be able to pre-empt or deny execution of the application at any time, just like for regular applications.

Small TCB & No Software Trust. The additional logic to the application should be small, typically increasing the code size by a fixed small amount. As a side advantage, this leads to applications that can be imported easily and are much smaller than (say) bulky VMs making the solution easier to deploy. Further, the primitive should require no software component outside the application binary to be trusted or to be verified. The implementation of the TCB in the CPU is fixed, can be verified once during manufacturing and thus can be secured to software tampering attacks. Several previous solutions instead rely either on a trusted hypervisor software, a trusted VM or trusted LibraryOS [14], [17], [20], [21], [29], [36].

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.

Scope. Secure application execution is not intended to be a stand-alone primitive that is useful by itself and is fairly complementary to abstractions provided by existing hardware primitives (e.g., TPMs) and hypervisor-based mechanisms. PodArch doesn't rely on secure boot or remote attestation to import and execute a pod binary, unlike many existing solutions. PodArch's security semantics are defined at the level of subsystems implemented by the CPU, i.e., for MMU, DMA, interrupts and so on. To achieve higher-level semantic guarantees, such as ensuring that the keyboard inputs and screen outputs remain confidential, requires additional abstraction like trusted paths to I/O devices (e.g., keyboards) [48]. Similarly, secure application execution does not apply to subsystems outside the CPU's control like disk systems. For example, if the PodFS application requests access to file stored on the local disk, the filesystem namespace integrity (e.g., pathname resolution correctness) is not guaranteed [28], [37]. Techniques orthogonal to PodArch's primitives can be used to achieve these, and many of them may require additional assumptions

[1] We refer readers to work by Chhabra et al. [21] for detailed view of the landscape of threat models considered in previous works.

from the target OS and hypervisors.

Secure execution does not trust any software outside the sensitive application binary. However, this does not imply that application itself is free from vulnerabilities, such as low-level memory corruption bugs. Defenses against these must be implemented by the application itself. For example, the application can implement CFI enforcement using secret IDs in its code blocks to provide additional guarantees [12], [45]. But this is beyond our direct goals.

Finally, secure execution primitives do not protect applications outside of the CPU's environment or on CPUs which have been tampered with and reverse engineered. For example, the PODFS application is an encrypted binary. We assume that its encryption key is not accidentally delivered to a malicious user (say Mallory) who runs the binary on a compromised CPU. A secure key management mechanism is necessary to ensure that the key is delivered to the right CPU (Bob's) and not tunneled to a remote attacker's machine. However, key management is not the main focus of our work and we discuss one solution we have implemented (out of the many options) in Section III-A.

B. PODARCH *Approach*

To achieve all the properties outlined in Section II-A, PODARCH introduces the abstraction of a *pod*. A pod is 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.

Main Design. PODARCH combines techniques of on-demand encryption (or memory cloaking [20]) with a set of new security invariants (detailed in Section III-B) to implement checks completely in hardware. Rather than requiring any strict partitioning of physical memory, 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, 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. PODARCH stores all AES-GCM integrity metadata for each pod within the pod's virtual address (VA) space, thereby piggybacking on standard OS paging mechanisms for memory management. During the PODARCH design, we make no assumptions about the security of control state registers, OS-specific data structures, or access to sub-systems (such as disk or protected physical memory) that are under the control of a typical hypervisor-based solution.

PODARCH removes any assumptions on remote attestation or secure boot by enforcing that pods be encrypted until they begin execution. This means that pod binaries are encrypted (under k_{app}) with AES-GCM and can be imported over a USB device or from the cloud via an untrusted OS. We explain how an existing x86 application is setup and used in a PODARCH CPU below.

Setup & Usage. In PODARCH, the application binary to be securely executed is encrypted with authenticated encryption before it is imported onto the target device. We consider the example described in Section II-A to explain how PODARCH can be used for solving Alice's problem. The first step in using PODARCH for such secure execution is to prepare a *pod binary*. Alice first selects an AES application key (k_{app}) which is used to create her encrypted application binary. She uses the PODARCH toolchain which takes a set of ELF x86-64 object files and generates a statically linked pod binary encrypted under k_{app}. This step is called the *pod sealing*.

Every PODARCH CPU has its own secret unique private key (k_{cpu}). This key is embedded in the hardware during the manufacturing. The corresponding public key (k_{pub}) for each CPU is registered and made public by the manufacturing authority. This can be made available on the device (e.g., as a QR code) for easy access. When Alice wants to execute her sealed pod on a selected target CPU, she has to deliver the k_{app} securely. To do this, Alice encrypts her application key (k_{app}) with the CPU's public key (k_{pub}). The resulting encrypted key ($\hat{k}$) is such that only the target PODARCH CPU can recover the k_{app}. Alice imports the sealed pod binary and the target-specific $\hat{k}$, on to the target machine (say, via a USB or via network). The $\hat{k}$ can then be delivered to the genuine target CPU by the compromised OS; only the CPU of Alice's choice can recover k_{app} from $\hat{k}$ as its encrypted under k_{pub}.

To obtain the correct k_{pub} of the target CPU, we can use any secure key management mechanism. We discuss one concrete idea in Section III-A which is operationally user-friendly and needs no manual key provisioning in the BIOS/CPU. Our choice of key management infrastructure is based on simplicity; it can be achieved via different authenticated key exchange protocols piggybacking on TPM-based public keys in the future, without changing the core design of PODARCH. Once the $\hat{k}$ is delivered to the target CPU, only that CPU can execute the pod. This eliminates malware attacks which try to emulate a compromised (or revoked) CPU [1], [30] or tunnel Alice's application to a remote attacker's machine where the attacker can mount additional attacks like cold-boot attacks (See Section V-A) [25].

C. *Differences from Existing Solutions*

Various techniques piggybacking on hypervisor-based mechanisms [20], SRTM primitives in TPMs [6], DRTM mechanisms in TPMs [33], VM isolation [47] and library OSes [36] have been proposed. Some of these solutions like library OSes can be used in conjunction with PODARCH but they do not satisfy all the goals of PODARCH when used independently. All of the above solutions depend on software TCB unlike PODARCH which is purely an architecture based security

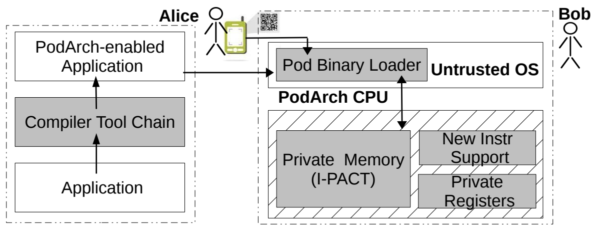

Figure 1: PodArch Design. The shaded regions indicate the modifications to the existing system. Only the hatched region is trusted.

primitive. In fact, the functions of their software layers are such that they cannot be moved to the CPU — e.g., reading disk or OS data structures, or spawning processes. We discuss the trade-off between the compatibility and security assumptions in each of these techniques in Section VII. Here, we limit the discussion to the only available architectural primitive for secure application execution proposed recently.

Intel SGX. PodArch is closely related to a recent proposal called Software Guard Extensions (SGX) from Intel (announced in September 2013) for its future CPUs [34]. These extensions allow an application to instantiate an *enclave* i.e., a protected area in the application's address space. It achieves secure execution by giving a separate region of *physical* memory to enclaves and prevents the OS from accessing the enclave's physical pages.

PodArch has been developed independently and concurrently with Intel SGX, and as of this writing, no public implementation (CPU, OS, applications, etc.) is available for Intel SGX. PodArch and Intel SGX seem to offer similar levels of security at a high-level: protecting against malicious compromised OSes. However, at a technical level, PodArch has at least 3 conceptual differences, and several other differences in design details, which we demonstrate with a complete system evaluation in this paper.

1. Physical memory partition. SGX requires the OS to partition out a contiguous region of physical RAM for enclaves, and both the OS and the applications need to be programmed to be aware of this separation. PodArch does not need the OS/applications to be aware of such partitioning, thus retaining transparency to the legacy OSes and applications. On the other hand, memory management semantics of a SGX enclave change the traditional OS design, since it has to manage both the system and the enclave memory, and hence losing on its transparency property.

2. Executable Binary Encryption prior to loading. SGX does not protect code/data in the executable binary before being imported in an SGX enclave; it relies on local and / or remote attestation to check code integrity after the EXE is loaded. This requires a quoting enclave to be running securely on the target machine [2]. The integrity of the quoting enclave is in turn checked by a remote verifier. In contrast, PodArch uses

[2]At least one remote attestation needs to run to start a root-of-trust enclave on the target machine.

Prev. Work	HW Attacks	OS Attacks	OS Compat.	Hyp $\notin$TCB	App. Port.
XOM [31]	✓			✓	✓
AISE [39]	✓				✓
OverShadow [20]		✓	✓		✓
SP3 [46]		✓	✓		✓
Bastion [17]	✓	✓	✓		
HyperWall [41]			✓	✓	✓
SecureME [21]	✓	✓	✓		✓
Intel SGX [34]	✓	✓		✓	
PodArch	✓	✓	✓	✓	✓

Table I: Insufficiency of related work. Column 2-6 denote if solution is secure against hardware attacks, OS attacks, maintains compatibility with existing OS, does not include hypervisor TCB and the ease of porting application respectively. A ✓ denotes that the technique demonstrates the property and a blank cell denotes that the property is not supported.

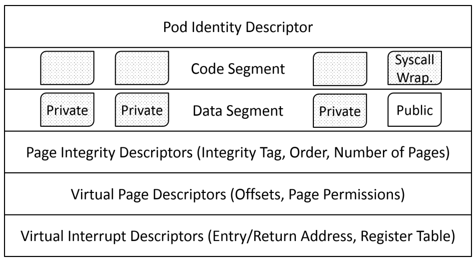
Pod Identity Descriptor
Code Segment
Syscall Wrap.
Private
Private
Data Segment
Private
Public
Page Integrity Descriptors (Integrity Tag, Order, Number of Pages)
Virtual Page Descriptors (Offsets, Page Permissions)
Virtual Interrupt Descriptors (Entry/Return Address, Register Table)

Figure 2: Structure of pod executable binary produced by the PodArch compiler toolchain after pod sealing.

encrypted executable binaries which are decrypted only after the OS loads it safely, without necessarily relying on remote attestation.

3. Restricted System Call support. Standard system calls are not available to an enclave since SYSCALL and SYSENTER instructions are illegal inside the enclave [4]. Alternatively, enclaves may invoke systems calls by explicitly passing the system call parameters out of the protection enclave. SGX does not specify the security semantics of such an interface, which may be susceptible to subtle attacks [18]. In contrast, PodArch allows pods to invoke system calls from protected code pages via a carefully designed interface (See Section IV).

In summary, Intel SGX can offer the same level of security as PodArch by relying on attestation abstractions. In contrast, PodArch aims to maintain compatibility (or transparency) with legacy OSes and binaries offering an end-to-end mechanism for protecting EXEs without necessarily relying on local / remote attestation. A complete comparison would only be possible when a SGX-based CPU prototype, OS and application infrastructure is available in the future.

III. PodArch Overview

PodArch applications are largely akin to regular x86 applications, except that they are designed to be run only on genuine PodArch CPUs. We discuss the steps in creating a pod binary, its loading, execution and termination below.

A. Pod Lifecycle

Key Management. An important challenge is that Alice does not know beforehand, the identity of the (borrowed) CPU she is going to use for execution. How does she seal the binary with k_{app}? To solve this, we envision a simple strategy, a mobile application which Alice can use before using the borrowed machine. Alice first scans the QR code which encodes the CPU's public key from the target machine and feeds it to her mobile application as shown in Figure 1. The mobile application then connects to the PodArch service and checks if the target CPU is not a revoked CPU. After the affirmation, Alice enters her application key in her mobile and asks it to generate *target specific* $\hat{k}$. Now Alice can submit this $\hat{k}$ along with the pod binary to the OS for setting up a secure pod. Note that other key delivery mechanisms such as authenticated key exchange can be used but that is not the focus of this paper.

Pod Loading. During sealing, the toolchain adds metadata and set of security descriptors to the ELF binary (See Section IV-D). Specifically, pod executables start with a pod identifier which allows the OS to recognize pod binaries. nbut it performs one additional step. Specifically, the pod loader reads encrypted metadata headers namely, the identity, page integrity, virtual page and virtual interrupt descriptors from the executable binary. It registers them with the PodArch CPU via a special instruction `pod_load`. The OS also loads the $\hat{k}$ into the CPU as a part of this instruction. PodArch CPU stores this information in its internal data structures. It uses this information later to check the post-load integrity and to further protect the pod.

Like in regular legacy applications, the OS also sets up the virtual address space for the pod, but doesn't allocate physical pages to it. The virtual to physical memory allocation is handled lazily during execution (demand paging) in commodity OSes. PodArch keeps compatibility with demand paging.

Pod Execution. The OS is not trusted to have loaded the binary correctly. When the pod begins execution in ring 3, the first instruction at the entry point is a special instruction called `pod_enter`. This instruction checks the pod setup by the OS in the previous phase. To this end, the PodArch CPU recovers k_{app} by decrypting $\hat{k}$ and uses it to decrypt the pod metadata in the CPU. The CPU further checks the number of pages, the contiguity of the pod's VA space using this metadata, details of which are discussed in Section IV-D. This is necessary to assert a faithful setup of pod before execution. If the OS has failed in this setup, `pod_enter` detects it and terminates the pod's execution securely.

PodArch CPU isolates pod's execution from the OS and non-pod applications. The OS can allocate any physical page to pod's VA (unlike SGX), and the allocated physical memory range need not be contiguous. The MMU checks each virtual-to-physical (V-P) address translations and thus tracks which physical pages are mapped to the pod (or are "owned" by the pod). Further, it enforces a key invariant: whenever the pod-owned pages are accessed by DMA-capable devices or mapped to the address space of other applications, they are encrypted with authenticated encryption transparently. Decryption of these pages is done lazily, that is, only when the encrypted page is accessed by the pod application in ring 3. This mechanism ensures that the OS can re-allocate physical pages to other applications or swap them out, if needed, and hence offers compatibility to the demand paging and memory management logic of the OS (See Section IV). Loading of the encrypted binary pages from disk also works in transparent way. This ensures that when the physical page is swapped out or reallocated out of the pod's VA space, it is always in encrypted state.

The execution of pods and the OS interleaves on the PodArch CPU. The transition from the pod (ring 3) to the kernel constitutes a security "context switch". These may occur due to hardware interrupts, software interrupts (including system call interrupts) and exceptions. PodArch incorporates additional mechanisms for preserving data integrity across context switches and allows control flow to return only to valid vectored entry points. We discuss the full details of these control flow restrictions in Section IV-D and Section IV-F.

Pod Termination. By design, PodArch allows the OS to kill the pod at any time including a normal process termination. The `pod_exit` instruction can be used to instruct the CPU to safely terminate the current pod. Also, if the OS violates any of the PodArch's security invariants during execution, PodArch securely terminates the pod application. During termination, the registers contents are zeroed and the unencrypted physical memory pages of the pod are encrypted. All entries of the pod pages are cleared in PodArch's internal structures.

B. Security Invariants

To guarantee secure execution of application in presence of a compromised operating system, our solution enforces the 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 at any instant in the TLB.

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

With the above outlined security invariants PodArch achieves the desirable primitives for secure execution and ensures confidentiality and integrity of an application's code and data. We discuss the detailed design of PodArch in the next sections and explain how each of these security invariants are enforced in our solution.

C. Security and Compatibility Challenges

Once the application's key is in the CPU, there are several challenges to load and execute the application in isolation from the OS. The main challenge in designing PODARCH stems from two sources: *compatibility* while ensuring *two-way isolation* between the OS and the application throughout its lifetime. We discuss several important ones below.

Secure loading. The normal binaries (say ELF executables) presently rely completely on the OS to load all its pages in the memory and start execution. A malicious OS can tamper with the section contents, make the physical page layout inconsistent with the executable's, overlap code and data pages maliciously, and so on. The pod delegate the checks against these attacks before starting execution to the CPU. This is addressed by enforcing security invariant **SI-1**.

Physical memory allocation. Once an encrypted executable is loaded, the pod's pages must be isolated from access to the OS and other applications. One way to do this is to allow the pod to control the region of physical memory allocated to it — indeed, Intel SGX and DRTM-based mechanisms delegate physical memory management to the application. However, this creates a large incompatibility to the semantics of legacy ring-3 applications (which solely deals with virtual memory). Further, the OS's physical memory management logic must be changed to accommodate the application's demands; for example, demand paging mechanisms in the OS allow processes to share physical pages as late as necessary. Growing and shrinking virtual pages on-demand requires the pod to implement page management logic. Ideally, we seek a mechanism which leaves the application's and the OS semantics of virtual memory largely unchanged — that is, the application should only be concerned with managing its virtual memory, while the OS can freely decide the physical memory management for all applications (including non-pod applications).

Physical memory protection. The PODARCH CPU must isolate the application's memory from the direct access by the OS, by default. However, when executing in ring 0, how does the CPU know which application's virtual page is a physical page allocated to? In many hypervisor-based solutions to introspect application processes, the hypervisor relies on the untampered state of process-identifying control registers (e.g., the `CR3` or OS specific process data structures in the memory) to identify processes and shadow page tables to determine page ownership. However, if the guest OS itself is malicious, identifying the process based on the assumption that process control register isn't spoofed is a challenge. Ideally, the CPU-based mechanism must make minimal assumptions, such as not depending on the state of CR3 register maintained by the OS.

Off-RAM protection. The application's data can be exfiltrated via DMA-capable devices off-RAM into swap space, where it is outside the CPU's restrictions and thus can be tampered with. In fact, page management functions of OS rely on these even for benign functionality. Ideally, the OS should be able to use these functions without much modifications to its existing logic for swapping and disk access. At the same time, the pod's application data must be protected on the swap / disk device.

Maintaining control flow & Data state across pre-emption. Application's execution can be pre-empted by interrupts and faults. During such pre-emption, the OS can change the control flow of the pod (by changing the EIP) or more subtly, reorder the physical pages allocated to a contiguous range of virtual addresses. These are ways to change the control flow without failing any virtual address based checks that the pod may implement. Similarly, the OS may tamper the register state. Designing a secure system against these attacks is important.

Interfacing with the OS. How does the pod exchange legitimate data with OS via system calls, if all its memory is accessible only to itself? The design requires safely sharing intended data between the OS and the application. That is, the mechanism must allow the application to specify which parts of its virtual address space should be accessible to the OS or to other applications/pods to enable legitimate data sharing. Hypervisor-based solutions do not permit such app-specified control over the application's address space which may force incompatibility with application's needs (e.g., for memory sharing) [20]. PODARCH enforces security invariant **SI-2** to allow secure interfacing with the OS.

IV. PODARCH DESIGN

So far, we have discussed the lifecycle of pods at a high level. We now discuss the design details of each phase along with the key factors that help us achieve our goals.

A. Security Invariants for Memory Management

PODARCH must prevent different principals (OS, non-pods and other pods) from accessing pod's memory pages. To this end, PODARCH tracks the ownership of each memory unit (physical page) allocated to the pod and prevents malicious access by non-owners. The semantics of the pod define a unique identity (k_{app}), that is the key associated with the pod application. Each pod has a specific owner identity principal i.e., its k_{app}. All other principals (OS, non-pod applications) on the system have their identity set to a `null` principal.

Tracking Current Executing Authority. At the time of any virtual to physical (V-P) address translation, the CPU ascertains the current executing authority and stores it in an internal register called $\texttt{Reg}_{CEA}$. The $\texttt{Reg}_{CEA}$ is initialized with the k_{app} of the pod when the OS first initializes the VA space of the pod. The OS calls a new special instruction `pod_load` asking the CPU to bind the k_{app} with a certain process identity — specifically the `CR3` value. Subsequently, whenever switching to ring 3, the CPU uses the `CR3` value to lookup the associated k_{app} and sets that as the current executing authority in $\texttt{Reg}_{CEA}$ (**SI-3**). In ring 0, the $\texttt{Reg}_{CEA}$ is set to `null`. Note that the CPU does not track the updates to `CR3` for the same application, since the OS can maliciously try to access the pages by manipulating the CR3 value. The first CR3 value that OS allocates to the pod process is used by the CPU throughout that pod's lifecycle. If the OS creates new page tables for the same pod process and uses a different CR3 value, PodArch will not use the updated CR3 value as the OS might have tampered with the page table to access pods pages

– which is a security threat (see Section V-D). PODARCH does not trust the CR3 value in enforcing any memory protection and security guarantees — it is merely used to lookup k_{app}.

Tracking page ownership. Once a physical page is allocated by the OS to a specific pod, the CPU marks the corresponding pod as its owner. Specifically, the CPU has an internal data structure that maps each physical page to its corresponding owner (identified by k_{app}) (**SI-4**) , i.e., the CPU maintains the physical page to owner mapping (P$\mapsto k_{app}$). This data structure is called an *Inverted page access control table* (I-PACT). The concept of I-PACT in PODARCH is similar to that of inverted page tables (IPT) which has been widely used for page management in traditional systems for better performance [3], [9] and for hardware-controlled protection of memory resources [41]. Figure 4 shows the structure of I-PACT.

The CPU's MMU observes V-P translations for all applications (and the OS) on the system. Thus, it is easy for the MMU to track a reverse mapping P-V. The CPU does not perform any translation in its internal table. Whenever the MMU encounters a new P-V mapping, it finds the current executing authority (Reg$_{CEA}$) and associates it as the owner of the physical page. This (P$\mapsto$<V, k_{app} >) mapping is stored in the I-PACT, which associates the physical page to its owner pod and the virtual address. With demand paging as in modern OSes, these bindings can change as the OS decides.

Address translation checks. The CPU performs certain security checks at each virtual to physical address translation at runtime, described below.

Ownership checks. One of the critical checks is to ensure that each physical page is only being accessed by its legitimate owner. That is, for each memory accessed in ring 3, the access is allowed if and only if the physical page is owned by the current executing authority (Reg$_{CEA}$). If the OS spoofs any values in determining the current executing authority (such as the CR3 register), the Reg$_{CEA}$ value will not match the owner in the I-PACT for the accessed page, and the ownership check will fail.

On-demand encryption and content integrity checks. During a V-P address translation, the CPU performs an on-demand encryption of the page if the page owner does not match the Reg$_{CEA}$ of the page, thus enforcing security invariant **SI-5**. The CPU encrypts the physical page on-demand using authenticated encryption with the key of the current executing authority (Reg$_{CEA}$) and updates the new owner in the I-PACT. Then, it marks this physical page as encrypted by setting a *swap bit* in the I-PACT; this bit is also used in the pages accessed by DMA-capable devices, as discussed below.

Encrypted pages are decrypted lazily, that is, only when they are later accessed by the legitimate owner in ring 3. Specifically, the CPU decrypts a page marked encrypted (swap bit = 1) if and only if the current executing authority's key in Reg$_{CEA}$ successfully decrypts the authenticated encryption. Note that authenticated encryption (e.g., AES-GCM) provides authenticity, i.e., the wrong key will result in a failed decryption. Any attempts by the OS to set the wrong Reg$_{CEA}$ will result in failed decryptions. The CPU performs these decryptions when it encounters a new P-V mapping in address translation. A successful decryption ensures that the page was encrypted under the owner's key, and that its confidentiality and integrity is preserved. Only if the decryption succeeds in the address translation, the CPU associates this physical page to k_{app} (stored in Reg$_{CEA}$) in its I-PACT. This way, the CPU consistently ensures that the content of the physical memory is associated to the right k_{app}, starting from the first decryption of the pod's pages after being loaded from the binary until the pod's termination. Note that this on-demand encryption and lazy decryption keeps compatibility with demand paging, just like in the semantics of regular x86 applications.

VA-to-content Binding. The PODARCH CPU allows the untrusted kernel to allocate arbitrary physical pages to the virtual address space of the pod. This can have subtle consequences. For example, an OS may not load the first page in the code section of the pod executable to its legitimate VA location; or, it may layout the memory out-of-order by (say) mapping the VA of the second page to the contents of the third page of the executable. Simply checking that a physical page is "owned" by a certain pod is therefore not enough.

One naive solution to this problem is to associate the virtual address to the actual content of the memory pages (and the static binary content). To do this, one could combine the start VA of each virtual page with the page's content when generating the authenticated encryption (AES-GCM), thereby binding them together. For example, if the virtual address 0x8048000 is the VA for the first code page in the executable, this VA could be appended to the page's content before computing the AES-GCM encryption of that page. This works but is incompatible with relocatable code, i.e., for code / data pages for which the VA addresses are determined at load-time dynamically. To keep compatibility with OS features like address space layout randomization (ASLR) and other position-independent code, in PODARCH we give each virtual page a relative virtual-page number (VPN), not its absolute VA. Therefore, all pages in the virtual address space are relatively positioned and receive VPNs in that order. This relative VPN is combined with the page contents whenever the AES-GCM encryption is computed, thereby eliminating attacks wherein the contiguous order of the virtual address space is subverted.

Metadata such as the AES-GCM integrity tag, for all the virtual pages are also stored in the reserved region of the process VA space and not in the CPU. Each pod maintains the metadata for all the virtual pages in a special data structure, namely Virtual Descriptor Table (VDT). If we treat the complete VA space of a process as contiguous, the VDT can be large —- proportional the 64-bit address space, if naively designed. However, only a small fraction of it may be in use. PODARCH only stores the information about this used fraction of VA space in the VDT. This results in having the VDT be segmented —- each virtual page is indexed by <VDTSeg, VPN>. We discuss further about this in Section VI.

Distinctness. Each page in the virtual address space of the pod process must be mapped to a distinct physical page (**SI-6**) . If not enforced by the CPU, it can be used by an OS to overlap the two different virtual pages to the same physical page.

CR3	Owner (k_{app})	C_VDTS	D_VDTS	S_VDTS	H_VDTS	...
$CR3_1$	k_1	c_1	d_1	s_1	h_1	.
$CR3_2$	k_2	c_2	d_2	s_2	h_2	.
$CR3_3$	k_3	c_3	d_3	s_3	h_3	.
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 3: Data structure in CPU private memory, contains `CR3` $\mapsto <k_{app}$, VDTBs $>$ mapping. PODARCH can support 8 VDTBs.

For example, the malicious kernel could overlap pod's stack and data pages, thereby causing unintended control flows and data corruption. As explained above, associating the relative VPN with the content of a page ensures that different virtual addresses map to different physical pages; in fact, more strictly the CPU enforces the linear ordering of the pod's VA space with each VDT segment. Such attacks are thus defeated.

Dynamic Changes to the VA. The pod process is free to grow or shrink its VA space at runtime; in fact, the stack segments on most Unix-like systems is dynamically allocated by the OS without explicit system calls from the pod. The pod need not do anything special for new pages added to its ownership. Whenever the CPU sees a new virtual to physical mapping for an owner, it implicitly adds it as the owner of that page. By default, newly added pages are treated as private to the pod. Integrity checks and encryption of dynamically allocated virtual pages is exactly the same as the statically loaded pages. If PODARCH wishes to override the default policy of treating all dynamically allocated pages as private, it can use two special instructions (`pod_addva` & `pod_delva`).

B. DMA

The OS needs the ability to swap out pages for its benign functioning. At the same time, DMA transfers can be used by a malicious OS to move a physical page content to the disk where it can be tampered with. PODARCH CPU, thus, intercepts on writes to the DMA ports (as specified by the chipset specifications) and detects if a page owned by a pod is scheduled for swap out. If so, the page is encrypted on-demand and a swap bit in the I-PACT is set to 1. As mentioned earlier, decryption is done lazily and only when the pod accesses the page later in ring 3, i.e., when a memory load instruction accesses a page marked with swap-bit $= 1$.

C. Context Switches

A switch from pod's user mode to kernel constitutes a "context switch". A typical context switch comprises of two tasks — register save-restore and control vectoring. This opens up attacks via the interface between the pod and the OS. The virtual interrupt descriptor holds the list of all such pod-specified registers.

Traps, Faults and Interrupts. The CPU handles these types of interrupts as follows. This is similar to the techniques used in hypervisor-based solutions and SGX [20], [34].

Register Save-Restore. When an application receives an interrupt from the CPU or a programmed software trap, it is delivered to the OS. Traditionally, the CPU stores the process's register states but does not clear it before passing the control to the OS. A malicious OS can glean sensitive information if the registers used by pod are not cleared/safeguarded before switching control from ring 3 to ring 0. Similarly, the OS can influence the execution of the pod by corrupting the register values used by the pod when transitioning from ring 0 to ring 3. Hence, PODARCH's default policy is to save all the registers (as in traditional CPUs) and additionally clear them when leaving/entering the pod. If the pod wants to over-ride this policy it can explicitly specify which registers are not to be cleared when entering and/or exiting the pod. System calls and signal handlers in the program can legitimately share the register values with the OS for passing arguments, program state, return values, etc. This way to support the necessary ABI, the pod can explicitly specify the set of registers as a part of the virtual interrupt descriptors in the pod binary (See Section IV-D).

Control Vectoring. After serving the interrupt / fault / trap / exception, the OS returns control to the pod in ring 3. The OS can exploit this interface, by tampering the return address into pod. Specifically, it can ask the pod to start execution at any arbitrary point in pod's code. Thus the OS should be limited to return to only fixed entry addresses in the pod. To ensure this, on switching from ring 0 to a pod in ring 3, the OS is allowed to enter to the pod only at valid entry points (**SI-7**). After handling faults and hardware interrupts, the execution must resume at the previously executing instruction or the next instruction in the pod respectively. The CPU checks this when the kernel returns to the pod to resume execution. The original semantics of the CPU for traps, faults, and interrupts are the same. For system calls and signal handlers, the pod can register a set of valid entry points to the pod. They should be specified in the virtual interrupt descriptors in the pod, details of which are discussed in Section IV-D. The CPU checks if the entry to the pod is one of these valid addresses and only then allows the pod to resume execution. If the check fails, the pod is securely aborted.

System Calls (User Defined Interrupts). All the pod's system calls pass through custom wrappers in the pod. These wrappers are added to maintain compatibility with the OSes semantics. Specifically, they marshal and unmarshal the call arguments and the return values respectively, by copying (deep-copy for pointers) them in pod's public pages. They also include sanitization code to check that all the data exchanged between the OS and pod lies within pod's public pages. The mechanism is carefully designed to be resistant to TOCTOU attacks i.e., to perform the copy to private pages before executing any sanitization checks or using the data [24]. We discuss the details of this mechanism in Section IV-F.

D. Changes to the Pod Executable

The VDT holds following three kinds of security metadata descriptors.

Page Integrity Descriptor. All the code and data sections in the pod executable are encrypted with authenticated-encryption using k_{app}. The page integrity descriptor contains the integrity tag of all these pages in the binary to preserve the page content integrity. The integrity tag is actually calculated by combining the page content and the virtual page number (VPN) in the binary. This follows from the VA-to-content binding as discussed in Section IV-A. It aids the CPU to check if the OS has not skipped loading some pages or swapped the order of two or more pages within the pod. Similar ordering checks need to be enforced across each preemption by the OS as we discussed in Section IV.

Virtual Page Descriptor. For fine-grained control of memory sharing, a pod can decide if a virtual page is public or private. Public pages are directly accessible to the OS (ring 0) and can be mapped into the address space of other applications by the OS (enabling shared memory), while private pages are not accessible directly outside the pod. A pod can also specify its own set of permissions for its resources (code and data pages) that are akin to rwx permissions enforced by the OS. The virtual page descriptor section contains a list of virtual page addresses, its corresponding page type (public or private) and 'rwx' bits if specified by the application. It is stored as a tuple *(V, page type, r, w, x)* in the application binary. PODARCH checks the page access type during address translation. In case of permission conflict between those set by pod and the OS, PODARCH performs a logical AND of the two permission sets and resorts to the most strict permissions.

Virtual Interrupt Descriptor. The descriptor contains a list of valid virtual entry points into the pod. It can be for the system call wrappers, signal handlers, or any other entry points permitted by the pod. It also contain the register save-restore metadata. All the registers that should not be cleared when leaving the pod and when entering the pod for a specific kind of abort/trap/faults/interrupts are explicitly stated by the pod as a bitmask in this descriptor. CPU will not clear these registers when entering and/or leaving the pod.

E. ISA Extension and Wrappers

Supporting PODARCH **instructions and data structures.** PODARCH introduces following four new instructions.

`pod_load`. The OS issues this instruction in ring 0 when it loads the pod binary for execution. On this instruction, PODARCH associates the k_{app} with the newly created pod for the first time and registers a new `CR3` - k_{app} mapping. The pod also registers its Virtual Descriptor Table Base (VDTB) with the CPU when the pod is first created. CPU uses this and VPN to index the pod's virtual descriptor table for accessing the descriptors later during execution.

`pod_enter`. This instruction is executed in ring 3 after loading the binary and before starting the execution. On encountering this instruction, PODARCH checks if the OS has faithfully loaded the binary and PODARCH metadata. It is a pre-execution check to ensure that the executable is securely loaded by the OS.

`pod_addva` & `pod_delva`. During the pod execution, a pod should be allowed to dynamically change (extend or shrink) their virtual address space. For example, an application can make a `mmap` call during program execution. By default all the dynamically allocated pages to the pod are private and the rwx bits are set as per OS semantics. If the pod wants to request pages which are public or with custom permissions, it is supported via `pod_addva`. Note that the extended VA space is also labeled with k_{app} (fetched from the $\texttt{Reg}_{CEA}$) and their allocated physical pages are similarly tied to the pod-owner identity k_{app}. Pod can delete the public pages or shrink its virtual address space explicitly using the `pod_delva` instruction. Semantically, the instruction clears selective entries for the pod from the I-PACT.

PPN	Owner (kapp)	VPN	Swap Bit	Page Type	Read Permn	Write Permn	Exec Permn	Integrity Tag
P_1	k_1	V_1	0	Public	1	1	0	h_1
P_2	k_2	V_1	1	Private	1	0	1	h_2
P_3	null	null	-	-	-	-	-	-

Figure 4: Inverted page access control table (I-PACT) structure in CPU private memory, contains P $\mapsto < k_{app}$, VPN $>$ mapping

`pod_exit`. This instruction can be issued only in ring 3. It reads the k_{app} from the current executing authority and deletes all the I-PACT entries corresponding to k_{app}. If the swap bit for the entry is D, PODARCH forcefully encrypts the physical page content before deleting it from the I-PACT. On `pod_exit`, CPU also zeroes the registers that hold pod specific values (as specified in the pod descriptors) to prevent the data leakage. If the OS does not use the `pod_exit` instruction, the I-PACT entries will lock the OSes access to the memory that was owned by pod. Hence, the OS must use the `pod_exit` to reclaim pod's memory.

System Call Wrappers. In general, the OS cannot access the pods pages. Specifically, for system call arguments, the OS cannot access the memory space of a process executing within a pod to access/dereference the arguments passed to it. This hinders the parameter passing in normal system call by reference and copy. In PODARCH, data passed into system calls is via public pages as they are accessible by more than one entities at a time and can be used as a controlled communication channel between the pod and the OS. Thus, all the arguments and return values for a system call are exchanged via public page between the pod and the OS.

To realize this solution, a pod binary has custom wrapper for each system call. When a system call is encountered, the CPU directs the execution control to a fixed address in the pod executable. The code at this address is essentially a table look up to invoke the appropriate system call wrapper. These wrappers copy the system call arguments (deep-copy) to a public page and then call the actual system call with the relocated arguments. When the system call returns, the wrappers copy the return values from public to private memory of the pod. Thus, system call wrappers bundled in pod application securely interface the call arguments both passed by value or reference (which are not accessible by default) to the kernel.

System calls are a special case of interrupts. Pod explicitly specifies in its descriptors not to clear the registers set by the pod, so as to send arguments to the system call, For ensuring the secure vectoring property during system calls, on `sysret` instruction the CPU makes sure that the system call returns to the previously stored address (See Section IV-C). This check is essential to prevent the OS from returning back to arbitrary points in the pod (a form of CFI [12]).

F. Changes to the Operating System

PodArch is designed with an aim to have modest changes to the current OSes. With zero programmer efforts, PodArch design keeps compatibility with copy-on-write, system call semantics, exceptions and interrupt handling, OS memory management and dynamic memory management requests by the pod.

Only two small changes are needed to support PodArch:
ELF Loader. During ELF loading, the binary (including code, data and pod descriptors) is in encrypted form. The loader issues a `pod_load` instruction and passes the $\hat{k}$ to indicates the PodArch CPU to register the newly created pod. Next, the pod loader also registers the address of virtual descriptors for the pod with the CPU. If the OS does not issue this instruction, the pod creation will be faulty. Once the pod is in execution the I-PACT check will report missing entries and signal the PodArch CPU about its malicious intent thus triggering the pod's kill during `pod_enter`.

Note that, it is possible to combine the functionalities of `pod_load` and `pod_enter`, but this might require the loader logic to be incorporated in every pod-binary. To avoid this, we choose to modify the ELF loader in the kernel and add only the `pod_enter` instruction as a part of the binary. This design decision helps to keep the pod light-weight and reduces the modifcation effort for every application, thus achieving our goal of easy portability.
VDT Fault Handler. The virtual descriptor table (VDT) is indexed by virtual addresses (virtual page numbers), and the Virtual Descriptor Table Base (VDTB) and marks the starts of the VDT. Each virtual page of the VDT stores its own AES-GCM integrity tags, so that no further page lookups are needed to check the integrity of the VDT. The CPU keeps this VDTB information along with the `CR3` - k_{app} mapping for the pod (See Figure 3). The VDT pages, thus, appear as normal private pod pages to the OS. They are lazily loaded with demand paging when the CPU needs to access them. Page faults may be generated when the CPU tries to access such VDT pages; we call these as *virtual descriptor faults* (VDF). In handling VDFs, the OS is not allowed to incur another page fault or a VDF. This is needed to ensure that VDFs don't trigger recursively. Therefore, the CPU enforces a strict no-second fault policy in the handling of VDFs — otherwise, it securely kills the pod for which the VDT is accessed. The OS needs to be aware of this restriction about double VDF faults, and handle them without raising further faults. This is the only change necessary to handle demand paging in our experiments. In PodArch, we implement VDF as a new error code in the existing Intel Interrupt 14 for Page-Fault Exception (#PF). By design, pod's pages are protected from being shared with other pod or non-pod processes. To support shared memory, two pods can share data via public pages; if they share a key, they can setup a secure channel between two processes. We envision that only the security sensitive part of the application logic will execute within the pod.

V. Security Analysis

For our threat model described in Section II-A, we revisit various threats to the confidentiality and integrity of a pod's execution with examples and explain how PodArch achieves its goals.

A. PodArch *CPU Bypass Attacks*

Local CPU emulation. A malicious OS can locally emulate a compromised CPU and bypass all the PodArch checks to compromise the pod. For this, the OS will give Alice the public key of the compromised CPU. When Alice submits $\hat{k}$, OS will use CPU's private to get hold of k_{app}. In our solution, Alice first checks if the CPU is compromised (using her mobile application) before submitting $\hat{k}$. Hence, this attack is not possible.

Remote CPU tunneling. Another scenario where the OS will try to bypass PodArch checks is by tunneling all the executions to a remote compromised CPU. The attack is not possible as Alice encrypts her k_{app} specifically for the CPU which is physically in front of her.

OS ignores `pod_load`. The OS can bypass the CPU checks by not loading the pod binary in a pod. If the OS does not load the pod executable's metadata during the pod load phase, `pod_enter` checks in ring 3 will fail. Thus, the pod's execution won't proceed past the first instruction.

B. Data Access Attacks

Direct access. The OS can try to maliciously access pod's pages from ring 0. However, PodArch's MMU checks make sure that whenever ring 0 tries to access pod's memory, it will always see an encrypted view.

Indirect access. The OS can manipulate page entries and corrupt the VA-PA mappings to claim that a pod's page belongs to the OS. PodArch's I-PACT keeps a track of the PA, <VA, owner> mapping and hence can detect any such attempts from the OS. Another subtle attack is wherein the OS maps two physical pages to the same virtual address in the pod. PodArch blocks this attack by enforcing the distinctness property, which makes sure that every physical address is uniquely mapped to a distinct <virtual address, owner> pair, via its VPN-based integrity check mechanism. Ordering attacks are also blocked as discussed in Section III.

Attacks using DMA. The OS can intercept the DMA transfers and steal or tamper the data when swapping it in or out of the disk. Every pod page is encrypted during this operation. Moreover, PodArch checks the integrity of the page before accessing it, thus preserving confidentiality and integrity of the pages respectively.

Attacks via I/O devices. PodArch doesn't provide higher-level semantic guarantees such as trusted paths to devices (e.g., keyboards, displays, network) [48]. Techniques orthogonal to PODARCH's primitives can be used to achieve these. For example, the pod application can use SSL protocol inside the pod to prevent the OS and the network from tampering the data, or use cryptographic keyboards for safe keyboard inputs.

C. Code Corruption Attacks

Application code modification. The OS or other applications can subvert the integrity of the execution by tampering the pages in the memory. It can maliciously ask the pod to execute arbitrary code by writing it to the pod's code pages. But PODARCH does not allow the OS to access the pod's code pages. Also, it checks the integrity of a newly loaded code page before starting execution in that page.

Loading malicious program/library. Instead of changing the application code, the OS can dynamically load malicious libraries in the pod and launch attacks through them. To this end, all the pod binaries are statically linked to trusted libraries by the user. A pod cannot dynamically link any code. Thus, the OS cannot use this channel to corrupt the pod.

D. Vectoring Attacks

Direct execution flow redirection. The OS can modify an interrupted state of a process and force the execution to an arbitrary location in the pod. To thwart this attack, PODARCH's secure vectoring mechanism checks if the return address during ring transition is the correct entry point in ring 3.

Indirect execution flow redirection. One indirect way to get the CPU to skip the vectoring checks is to spoof the $\texttt{Reg}_{CEA}$ and fool it into believing that it is not executing a pod. Consider the following hypothetical attack scenario. When handling an interrupt, the OS can replicate the pod's page table entries, and point the `CR3` to this copy. Since this `CR3` does not correspond to a registered pod, the CPU will skip the entry point check when the OS jumps to arbitrary location in the pod via the copied `CR3` and Page Table Entry (PTE). Thus the OS will succeed to fool the CPU in believing that it is entering a non-pod application in ring 3. So the CPU will not do the entry point check and the OS will succeed in running arbitrary code in the pod (assuming the code page is still decrypted). However, this attack will not work since PODARCH checks the I-PACT on every address translation. When the OS uses the new `CR3` and PTE, the physical address still belongs to the pod. Hence, the CPU will detect that the OS is maliciously trying to access pod's pages.

CFI guarantees. The above two defenses do not guarantee CFI or absence of ROP attacks from within the pod. We assume the pod binary adds defenses for such attacks [12].

E. System Service Attacks

Leaking interrupted program state. All applications on the system share the same set of general purpose registers. When the CPU changes its execution context, it can leak information via these shared registers.PODARCH preserves the confidentiality and integrity of the register values which are legitimately shared. For example, system call arguments passed via the registers, the interrupt number passed via the register, etc. Remaining register values are cleared by the CPU.

System call parameter tampering. Ports and Garfinkel [37] discuss the potential attacks performed by a malicious kernel through system call return values. Checkoway and Shacham introduce a real instance of such attacks - Iago attacks [18], wherein the OS can attack a pod by data returned from system calls, which is either a return value or a parameter by reference. Additional defenses against return values (including virtual address values) from system calls are the responsibility of the pod implementation; and can be provided as a part of the pod's system call wrappers. Currently, we provide simple system call wrappers which are a part of the pod that do basic parameter marshalling checks. They are carefully designed to prevent TOCTOU attacks and the checks are done only after copying them to the private memory of the pod. Section VI mentions the necessary design details.

DOS. The OS can always reject to serve the system service requests from the pod. It can also starve the pod from resources such as memory and execution time slice. These denial of service attacks are out-of-scope in this work.

F. Attacks on PODARCH

k_{app} **Leakage after import.** The OS, non-pod applications or co-existing pod applications can steal the pod's k_{app} and decrypt all the pod data. PODARCH design guarantees that the k_{app} is never accessible to either the OS or any other entity other than the owner pod. The k_{app} for each pod never leaves CPU's internal data structure. $\texttt{Reg}_{CEA}$ is flushed whenever the CPU transitions from ring 3 to ring 0. When CPU sees a change in executing authority (in ring 3), it loads the corresponding k_{app} in the $\texttt{Reg}_{CEA}$.

k_{app} **Leakage before import.** We assume that all the cryptography techniques are secure. PODARCH does not prevent attacks against cryptographic weaknesses or implementation side-channel [44].

VI. IMPLEMENTATION & EVALUATION

The current PODARCH prototype realizes the full system described in earlier sections. We demonstrate the practicality of the system by presenting quantitative results for experiments.

Implementation. We implement our prototype PODARCH for a single core CPU architecture in QEMU [8], a x86-64 system emulator. We approximately add 5164 lines of code to QEMU v0.14.1. We modify the Linux Kernel v3.2 by adding 415 lines of code to the elf loader `fs/binfmt_elf.c` and `arch/x86/mm/fault.c` to adapt it to support PODARCH. All the process specific metadata is stored as a part of the pod's address space in the VDT. The CPU uses fixed size data structures (proportional to the size of physical memory) and can be implemented as an on- chip cache.

VDT Management. Several data structures in our design require metadata per virtual page of the pod. For example, the AES-GCM integrity tags for each virtual page need to be stored somewhere as metadata. All the VA related mappings

Benchmark	**Description**	**LOC**	**Execution time (in sec)**			**Overhead%**	**Interrupts**			**Context switches**			**Enc**	**Dec**	**SC**
			t_{vv}	t_{vp}	t_{pp}	pp **VS** vv	I_{vv}	I_{vp}	I_{pp}	C_{vv}	C_{vp}	C_{pp}			
bzip2	Compression	5864	30.24	31.51	41.79	**38.21**	18001	18011	19091	15688	16474	19036	177275	168876	90
gobmk	Artificial Intelligence: go	158055	4.56	4.88	7.20	**57.86**	6891	7006	7448	6877	6587	7448	111498	105767	287
mcf	Combinatorial Optimization	1743	111.44	115.67	174.73	**56.79**	61003	59882	84215	58771	55662	83089	331	328	1409
h264ref	Video Compression	36280	204.44	264.192	352.35	**72.35**	93611	91881	98776	89876	92332	109889	287339	251891	453
omnetpp	Discrete Event Simulation	26979	18.01	27.05	33.31	**84.94**	8101	9150	10745	7888	9052	10482	8394	6513	188
sjeng	Artificial Intelligence: chess	10717	111.78	122.85	182.50	**63.27**	62899	69765	88769	65548	81773	100318	154737	110786	343
xalancbmk	XML Processing	268545	4.28	5.22	7.75	**81.14**	2875	3272	4282	3199	3044	4192	9003	7510	143
astar	Path-finding Algorithms	4453	997.96	1211.88	1640.59	**64.39**	278445	322145	587789	289611	388001	508471	42821773	42677618	1298
libquantum	Physics: Quantum Computing	2635	1.44	2.122	2.58	**79.17**	799	831	857	870	887	898	374	332	30
hmmer	Search Gene Sequence	20878	30.66	32.242	44.27	**44.38**	8863	8886	9630	7771	7886	9563	1134	1029	63
perlbench	Programming Language	128176	1.28	1.363	2.124	**65.94**	948	1078	1095	610	629	822	321	286	273
gcc	C Compiler	382852	32.66	34.998	60.2358	**84.43**	12776	14907	20516	15229	16224	20039	3680	720	477

Table II: SPEC CINT2006 performance summary for vanilla application on vanilla system (vv), vanilla application on PODARCH (vp), pod on PODARCH (pp). Column 4-6 gives the execution time in seconds and column 7 gives the % overhead of pod on PODARCH as compared to vanilla on PODARCH. Column 8-10 gives the no. of interrupts, column 11-13 gives the no. of context switches for all the three cases. Column 14 and 15 gives the encryption and decryption time required during swapping of pages during execution. The last column gives the no. of system calls made by the application.

and metadata are stored as virtual descriptors in virtual descriptor table (VDT), initialized in the static binary and loaded in memory at load time. This descriptor space need to be extended as the process grows its VA space. Since the VA space of a process is large, and we aim to support unlimited number of pod applications, we can't store them in a fixed region of physical memory (or else we would need to implement page management in the CPU). To avoid this, virtual descriptors are stored as private pages within the pod's VA space. Note that virtual descriptors need to only be modified by the CPU, so the CPU marks them inaccessible to the pod or OS in its address translations. Any accidental bugs in the pod, thus, cannot be exploited by kernel to change these descriptors.

To solve the large VDT size problem, PODARCH allows the application developer to specify different non-contiguous segments (upto eight in our design) in the address space thereby segmenting VDT storage in distinct regions. Each such segment is associated with a start address and the pages in the region are given VPNs relative to this address. Such segments are block of contiguous regions such as stack, heap, code and data, and are upto the binary to specify. The size for the metadata of these VDT regions are stored in PODARCH CPU's internal data structure (See Figure 3). The relative VPN (per segment) is used to index into the pod's VDT and fetch the page metadata.

Internal storage of CPU. There are data structures which only the CPU needs and are proportional to the physical RAM. For example, the I-PACT has a fixed number of bits per physical RAM page (See Figure 4); the CR3 - k_{app} mappings are proportional to the number of pods (See Figure 3). The CPU reserves a small amount of CPU-private physical memory to store these. This storage is small and thus can fit in a small amount of memory. In our solution, we use a fixed physical memory region to store these. This physical address range is made inaccessible to the OS and applications. This region of memory could be configured in the BIOS, for example. In a real CPU, there may be sufficient on-chip memory that could store this, but since this storage is proportional to the size of the physical pages, it's easier to keep it in private memory.

The size of Inverted page access control table (I-PACT) is constant for a given maximum RAM size supported by the CPU. One entry in the I-PACT represents one physical page, and takes 32 bytes. For a 4GB RAM size, the I-PACT requires 32MB private memory space. The remaining data structures (as discussed in Section VI) require a total of 125KB CPU private memory to support 1000 pods.

Modification to the tool chain. We modify the GNU GCC v4.6.3 tool chain to create PODARCH-enabled x86-64 applications in form of statically linked executables. We develop a binary re-writer that rewrites any executable to include pod specific metadata, thus generating a pod binary. We also modify the ELF loader to load this metadata and save it in the CPU data structures during the execution of a pod application. All the user has to do is use our toolchain to create user specific pod binary, with minimal changes to the `Makefile` of the application to add certain `.o` files for system call wrappers. The programmer has to include `pod_header.h` that contains all the system call wrappers. The source code is then converted to a pod executable using our modified compiler tool chain. Our PODARCH GNU compiler toolchain compiles the application and rewrites the binary to make it compatible with PODARCH design. Mainly, the modified toolchain performs the following steps.

(a) Statically compile the application and link all libraries (b) Link custom system call wrappers for marshalling (c) Align the sections to the page boundaries (d) Add bootstrap code and `pod_enter` instruction for the pod loading step (e) Support for public pages and custom virtual page permissions, if specified (d) Encrypt and authenticate the pages with AES-GCM while keeping the executable's semantics intact (f) Add descriptors for the integrity tags for all the encrypted page

A. Evaluation Goals & Benchmarks

We experimentally evaluate the following main goals:

- Efforts to support existing OS and legacy x86 applications to operate on PODARCH.
- Robustness and scalability of PODARCH.

PodArch Component	LOC/size (% change)
QEMU	5164 (**0.98%**)
Pod Application Wrappers	148
Linux Kernel	415 (**0.0051%**)
GNU Compiler Toolchain	1749
Increase in Pod executable binary size	168.03 KB (1.2 **%**)

Table III: PodArch: Number of lines of code added and increase in the size of executables for PodArch system.

- Performance of applications on PodArch CPU.

We port 12 SPEC CINT2006 and 18 [3] HBench-OS benchmarks and 50 case studies from CoreUtils package on PodArch architecture [2], [16], [26]. We compare the following configurations:

- Non-pod applications on Vanilla CPU (Baseline)
- Non-pod applications on PodArch CPU
- Pod applications on PodArch CPU

All the experiments are conducted on a Dell Latitude 6430u host, configured with Intel(R) Core(TM) i7-3687U 2.10GHz CPU, 8GB RAM. The QEMU VM is configured with one CPU, 4 GB RAM running 64-bit Linux 3.2.53 Kernel on Debian Jessie for all the experiments. All the data results are averaged over five runs and are reported with a 95% confidence interval.

B. Results

We find that PodArch offers easy portability, strong robustness, along with performance comparable to previous solutions.

Porting Effort. For adapting the Linux Kernel v3.2, we applied a patch of 415 lines of code. We statically compile and convert 12 vanilla SPEC CINT2006, 18 HBench-OS benchmarks and 50 coreutils applications to pod binaries via PodArch toolchain and the developer effort was negligible using our tool. Our system effortlessly transforms these applications to pod-binaries and executes them on PodArch. We exclude those applications that either require dynamic loading of libraries or accessing shared libraries at run time. We report that PodArch does not result in intrusive change to the OSes and does not require any developer effort to port existing legacy x86 applications to PodArch design. The specific LOC changes are in Table III.

Robustness. We test PodArch's scalability and robustness by stress testing our prototype implementation, in following two ways.

Scaling with Number of Pods: PodArch is carefully designed to support many multiple pods simultaneously. To test the scalability of PodArch, we launched multiple instances of SPEC CINT2006 benchmarks processes in 100 parallel pods. We report that PodArch implementation supported these pods without any hardware memory or register limitation.

System Stress Test: We chose HBench-OS, a system stress test benchmark which overcomes the shortcomings of LM-Bench [15]. We study the interactions between the operating system and the hardware architecture with 18 OS and memory intensive HBench-OS [16] benchmarks and determine the precise sources of overhead. These results do not necessarily represent the overheads for real-world applications, but aggressively test the OS and CPU robustness under stress. Table IV summarizes the results of microbenchmarks. For memory intensive operations, the tables shows a decrease in bandwidth value. Future work can aggressively optimizing these overheads when implemented in real hardware.

Property	Sub-property	% O_{vp}	% O_{pp}
Cache Latencies	L1 & L2	299.94	1499.03
	L3	699.88	699.93
Memory Intensive Operations	Raw Memory Read Bandwidth	74.56	94.98
	Raw Memory Write Bandwidth	78.84	96.05
	libc Bzero Bandwidth	74.72	95.36
	Unrolled Write Bandwidth	78.84	96.06
	bcopy Bandwidth	75.79	95.94
	Mmap'd Read	17.03	53.24
	Raw Hardware Read	74.56	94.98
File System	Filesystem Latency	47.81	470.12
Context Switch	Ctx (excluding)	67.38	118.19
	Ctx2 (including)	71.36	182.59
System Call	getpid	46.94	268.52
	getrusage	60.70	427.86
	gettimeofday	110.44	608.95
	sbrk	94.99	545.64
	sigaction	32.69	384.3
	write	49.69	477.98
Signal Handler	Installing Signal	35.42	387.02
	Handling Signal	72.80	314.03

Table IV: HBench-OS Summary: Overhead for vanilla apps on PodArch (O_{vp}) & pods on PodArch (O_{pp}) compared to vanilla apps on vanilla system(vv). lat_ctx, lat_ctx2: context switch latency by excluding & including cache conflict resolution time respectively. The numbers indicate a decrease in bandwidth for memory intensive operations.

Performance. Figure 5 shows the comparison of vanilla benchmarks (both on vanilla CPU and PodArch CPU) to those of pod-enabled benchmarks on PodArch CPU and gives the percentage increase in the execution time for each benchmark. We also report the number of interrupts, context switches, system calls, and encrypt-decrypt operations for each benchmark on PodArch in Table II. In our present evaluation, all the computations are performed on a single core. The average increase in the execution time for the pod-enabled benchmarks as reported by QEMU is 66.07 % and negligible memory overhead. Even when measured on QEMU hypervisor, the overhead is consistent and better than other solutions. Specifically, it is faster than the existing hypervisor based solution OverShadow [20] which report 70-100 % execution time and 100% memory overhead for SPEC CINT2006 benchmarks [20]. The main factors in the overhead in PodArch are I-PACT operations, cryptographic operations 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 like previous other work [20] rather than a hardware implementation. If done in a real CPU, the performance will be better. For example, offloading / parallelizing I-PACT and cryptographic operations with the instruction execution, or speculative execution of I-

[3]We do not include 8 benchmarks from HBench-OS which use `fork`.

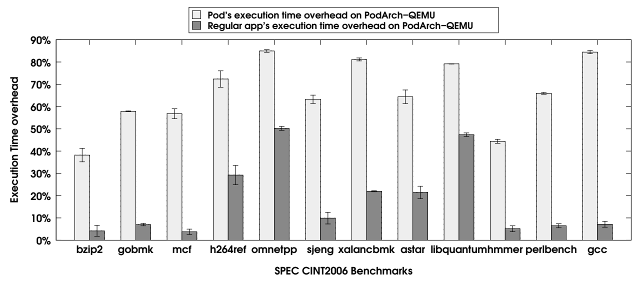

Figure 5: Execution time overhead for pod-enabled & vanilla SPEC CINT2006 on PodArch-QEMU vs. vanilla SPEC CINT2006 on vanilla-QEMU.

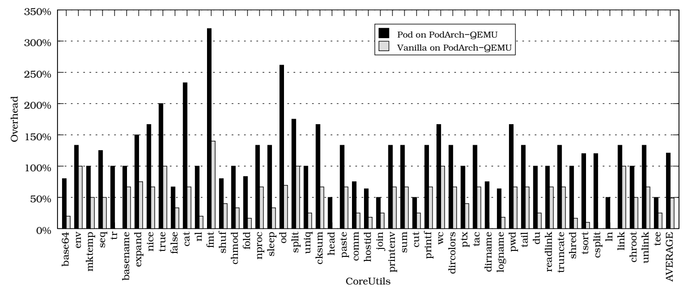

Figure 6: Execution time overhead for running pod-enabled and vanilla coreutils on PodArch and as compared to vanilla coreutils on vanilla-QEMU.

PACT operations, can enhance the efficiency of PodArch CPU.

In summary, PodArch shows high robustness with comparable performance for intensive real world benchmarks. We also demonstrate that the design is backward compatible to applications and OSes thus making PodArch an appealing solution.

CoreUtils as Case Studies. We demonstrate the expressiveness of PodArch by evaluating on 50 case studies that could be statically compiled from the coreutils package [2]. We select CoreUtils as our case studies since they commonly exists on every Unix OS and include programs which mainly perform file, text and shell manipulation operations. Our experiments show that PodArch is expressive enough to execute any application that can be successfully generated using static compilation. Our performance overhead is 120% for executing pod-enabled binaries on PodArch and 50% for vanilla binaries on PodArch as compared to running vanilla binaries on a vanilla-QEMU (see Figure 6). In our experiments, the input to all the file-based utilities is a text file of size 10 KB. The overhead is high for utilities that perform I/O intensive operations like fmt, cat, od, etc. which read every character from the input file and write to an output file.

VII. Related Work

We have already discussed the closest related work in Section II-C (See Table I and V for summary). We compare other past works to PodArch in terms of unit of protection, architectural support for secure application execution, piggybacking on protection via TPM attestation and purely hardware solutions.

VM as a Protection Unit. One way of securing the execution of mutually untrusted applications is to run each of them in separate VMs. Previous solutions such as Cloudvisor [47], Terra [23], NoHype [29] isolate the VMs either via the hardware or software protection. These solutions include the VM's OS into the TCB over and above the application. Proxos [42] and Drawbridge [36] reduce the OS TCB by moving much of OS's functionality to the ring 3 process [36]; however, this bloats the original application significantly and the mechanism requires thousands of lines of change to the commodity kernel [36]. A more recent applications of Library OSes — Graphene and Haven allow execution of multiple process but still relies majorly on including portions to the TCB. [14], [43].

Architectural Support. Various solutions such as Bastion [17], SecureME [21], Overshadow [20], SP3 [46], InkTag [27], TrustVisor [32], Chaos [19] directly support secure execution primitive via virtualization or special architectural support. However, they include hypervisor in their TCB, whereas PodArch eliminates it. For example, a closely related work Bastion [17] is designed to launch the trusted hypervisor (included in the TCB of the solution) securely. Hence, majority of the security enforcement tasks are delegated to the hypervisor without any changes to the OS or the application. The `secure_launch` hypercall involves actions such as disk access which cannot be implemented in the CPU. As opposed to PodArch design which stores all the metadata either in the CPU (I-PACT) or the pod (VDT), Bastion leverages scalable secure storage introduced in the hypervisor functionality. Unlike pods, the key management assumptions of Bastion's TSMs is not clearly stated. Our descriptors enforce VA-PA distinctness property and also bind the VA to the content to defeat attacks discussed in Section V, that Bastion may be susceptible to (as discussed in [21]).

Piggybacking on SRTM & DRTM Mechanisms. Static root-of-trust management (SRTM) primitives allow determining if the OS is compromised, or is in a pre-specified safe state before the sensitive operation commences via remote attestation [6]. White listing safe software state is impractical as the number of possible software stack configurations (e.g., OS versions, drivers) becomes large with time.

Flicker and TrustVisor, are DRTM based systems which rely on remote attestation to run pieces of application logic (PALs) in ring 0 [32], [33]. Piggybacking on these "privileged execution" mechanisms to achieve execution on unprivileged ring 3 code is thus feasible, however, it has many limitations. First, the application needs to be broken up into PALs which get sufficient time quantas to execute meaningful tasks without pre-emption. Transformation of legacy x86 applications, though could be automated to a fair extent, is not straightforward and requires significant developer effort. Second, the mechanism requires an intrusive change to the OS — that is, the OS scheduler needs to be aware of additional kernel

Techniques	Pvs.S	UExec	No STCB	Comp.
Software-based Sol.				
OverShadow [20]	✓	✓		✓
CloudVisor [47]	✓	✓		✓
NoHype [29]	✓	✓		✓
Drawbridge [36]	✓	✓	✓	
Hardware-based Sol.				
TPM SRTM Mechanism [6]	✓		✓	✓
Flicker [33]			✓	
`SMEP, SMAP` [5]	✓	✓		
ARM TrustZones [13]			✓	
XOM, XOMOS [22], [31]	✓	✓		
IBM SecureBlue++ [38]	✓	✓	✓	
Intel SGX [34]	✓	✓	✓	
PodArch	✓	✓	✓	✓

Pvs.S	Separation of provisioning and security
UExec	Unprivileged execution of secure applications
No STCB	Security does not rely on software TCB
Comp.	Compatibility with legacy OSes & apps

Table V: Existing techniques for secure application execution on untrusted operating systems. Columns 2 - 5 represent desirable security properties. A ✓ denotes that the technique demonstrates the property and a blank cell denotes that the property is not supported.

components, needs to allocate physical memory to be used by the application, needs various drivers to be aware of PAL delays, and so on. Third, if the application uses a lot of sensitive memory pages and is pre-empted, Flicker and TPM-based sealing mechanisms don't offer mechanisms to protect the application's large memory space across a preemption [11], [33].

Purely Hardware Solutions. IBM SecureBlue++ [38], Intel SGX [34], XOM [31], AISE [39], AEGIS [40], TrustZone [13] propose a trusted physical hardware design to protect against untrusted OSes. Unlike these, PodArch is designed to maintain high compatibility with legacy OS and applications. HyperWall [41] does not include hypervisor in the TCB and protects the guest VMs from malicious hypervisor in the cloud. However, it assumes that the guest VM's OS is uncompromised. OASIS [35] adds new instruction to the CPU to support secure execution of applications, but it relies on remote verification of the attested code that requires the client to be online.

Discretionary User-from-Kernel Isolation. New processor features, such as the `SMAP` and `SMEP` instructions on Intel, have been added to disable access to user-level pages from the kernel [5]. These mechanisms can help prevent user processes from exploiting kernel bugs to access payloads stored in user memory with the OS's supervisor privilege. However, these controls are discretionarily set by OSes and can be bypassed by persistent malware running with the kernel's authority.

VIII. Scope for Future Work

Our current implementation does not support pods to `fork` or `clone` another pod process. Nonetheless, these can be supported in future with our toolchain. For now, pods can `fork` to non-pod processes, which are treated so by default. The present PodArch design considers only a single CPU architecture. Extending PodArch architecture to a multicore environment is a promising next step. Also performing a detailed hardware implementation of PodArch along with fabrication on chip can be done in future. Further performance improvement can be gained by better data structure management or parallelising the implementation in hardware.

IX. Conclusion

We present a new architecture — PodArch, for secure execution of legacy x86 applications on untrusted operating systems. PodArch is a purely hardware-based mechanism achieving transparency with existing systems without intrusive changes to the applications or the OS and requires zero developer effort to port legacy applications on commodity OS. Our tools and patches for PodArch are open source and available for exploration.

X. Acknowledgements

We thank Zhenkai Liang, Hu Hong and Hoon Wei Lim for their insightful comments on an early presentation of this work. We also thank Anh Dinh, Xinshu Dong, Zheng Leong Chua and Yaoqi Jia for their suggestions in preparing the final version of the report.

References

[1] "Bluepill Project," theinvisiblethings.blogspot.sg/2006/06/introducing-blue-pill.html.

[2] "GNU CoreUtils," http://www.gnu.org/software/coreutils/.

[3] "IBM PowerPC," www-01.ibm.com/chips/techlib/techlib.nsf-/literature/PowerPC.

[4] "INTEL SGX Programming Reference." software.intel.com/sites/default/files/329298-001.pdf.

[5] "Intel Software Developer Manuals," www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[6] "Intel Trusted Execution Technology: Software Development Guide," www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf.

[7] "PodArch," https://github.com/shwetasshinde24/PodArch.

[8] "QEMU, Open Source Processor Emulator," www.qemu.org.

[9] "UltraSPARC," www.oracle.com/technetwork/systems/coolthreads/ultrasparc/index.html.

[10] "Universal JTAG," urjtag.org.

[11] "Trusted Computing Group. Trusted platform module." July 2007.

[12] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," in *Proceedings of ACM conference on Computer and Communications Security*, ser. CCS, 2005.

[13] ARM, "ARM Security Technology Building a Secure System using TrustZone Technology. ARM Technical White Paper," 2013.

[14] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. USENIX Advanced Computing Systems Association, October 2014.

[15] A. B. Brown and M. I. Seltzer, "Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture," in *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1997.

[16] A. B. Brown, “A Decompositional Approach to Computer System Performance Evaluation,” 1997.

[17] D. Champagne and R. Lee, “Scalable architectural support for trusted software,” in *High Performance Computer Architecture (HPCA)*, 2010.

[18] S. Checkoway and H. Shacham, “Iago attacks: Why the System Call API is a Bad Untrusted RPC Interface,” in *ASPLOS*, 2013.

[19] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, and W. Mao, “Tamper-Resistant Execution in an Untrusted Operating System Using A Virtual Machine Monitor,” 2007.

[20] 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,” 2008.

[21] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, “SecureME: A Hardware-software Approach to Full System Security,” in *ICS*, 2011.

[22] M. H. David Lie, Chandramohan A. Thekkath, “Implementing an Untrusted Operating System on Trusted Hardware,” in *SOSP*, 2003.

[23] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, “Terra: A Virtual Machine-based Platform for Trusted Computing,” 2003.

[24] T. Garfinkel, B. Pfaff, and M. Rosenblum, “Ostia: A Delegating Architecture for Secure System Call Interposition,” in *NDSS*, 2003.

[25] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Cal, A. J. Feldman, and E. W. Felten, “Lest We Remember: Cold Boot Attacks on Encryption Keys,” in *USENIX Security Symposium*, 2008.

[26] J. L. Henning, “SPEC CPU2006 Benchmark Descriptions,” *SIGARCH Comput. Archit. News*.

[27] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, “Inktag: Secure applications on an untrusted operating system,” ser. ASPLOS, 2013.

[28] E. jin Goh, H. Shacham, N. Modadugu, and D. Boneh, “Sirius: Securing Remote Untrusted Storage,” ser. NDSS, 2003.

[29] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, “NoHype: Virtualized Cloud Infrastructure without the Virtualization,” in *Proceedings of International Symposium on Computer Architecture*, ser. ISCA, 2010.

[30] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch, “SubVirt: Implementing Malware with Virtual Machines,” in *Proceedings of IEEE Symposium on Security and Privacy*, 2006.

[31] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, “Architectural Support for Copy and Tamper Resistant Software,” in *ASPLOS*, 2000.

[32] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, “TrustVisor: Efficient TCB Reduction and Attestation,” in *Proceedings of IEEE Symposium on Security and Privacy*, ser. SP, 2010.

[33] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, “Flicker: An Execution Infrastructure for TCB Minimization,” *SIGOPS Oper. Syst. Rev.*, 2008.

[34] 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 *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP, 2013.

[35] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan, “Oasis: On achieving a sanctuary for integrity and secrecy on untrusted platforms,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’13, 2013.

[36] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, “Rethinking the Library OS from the Top Down,” ser. ASPLOS, 2011.

[37] D. R. K. Ports and T. Garfinkel, “Towards Application Security on Untrusted Operating Systems,” ser. HOTSEC, 2008.

[38] Rick Boivie, “SecureBlue++: CPU Support for Secure Execution,” 2012.

[39] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, “Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly,” in *MICRO*, 2007.

[40] G. E. Suh, C. W. O’Donnell, I. Sachdev, and S. Devadas, “Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions,” *SIGARCH Comput. Archit. News*, 2005.

[41] J. Szefer and R. B. Lee, “Architectural Support for Hypervisor-secure Virtualization,” in *ASPLOS*, 2012.

[42] R. Ta-Min, L. Litty, and D. Lie, “Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable,” ser. OSDI, 2006.

[43] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porte, “Cooperation and security isolation of library oses for multi-process applications,” in *EuroSys ’14: Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[44] Z. Wang and R. B. Lee, “New Cache Designs for Thwarting Software Cache-based Side Channel Attacks,” in *ISCA*, 2007.

[45] Z. Wang and X. Jiang, “HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity,” in *Proceedings of IEEE Symposium on Security and Privacy*, 2010.

[46] J. Yang and K. G. Shin, “Using Hypervisor to Provide Data Secrecy for User Applications on a Per-page Basis,” in *VEE*, 2008.

[47] F. Zhang, J. Chen, H. Chen, and B. Zang, “CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization,” in *SOSP*, 2011.

[48] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, “Building Verifiable Trusted Path on Commodity x86 Computers,” in *Proceedings of IEEE Symposium on Security and Privacy*, 2012.