Reducing World Switches in Virtualized Environment with Flexible Cross-world Calls

Wenhao Li, Yubin Xia, Haibo Chen, Binyu Zang, Haibing Guan

Shanghai Key Laboratory of Scalable Computing and Systems
Shanghai Jiao Tong University
{liwenhaosuper, xiayubin, haibochen, byzang, hbguan}@sjtu.edu.cn

Abstract

Modern computers are built with increasingly complex software stack crossing multiple layers (i.e., worlds), where cross-world call has been a necessity for various important purposes like security, reliability, and reduced complexity. Unfortunately, there is currently limited cross-world call support (e.g., syscall, vmcall), and thus other calls need to be emulated by detouring multiple times to the privileged software layer (i.e., OS kernel and hypervisor). This causes not only significant performance degradation, but also unnecessary implementation complexity.

This paper argues that it is time to rethink the design of traditional cross-world call mechanisms by reviewing existing systems built upon hypervisors. Following the design philosophy of separating authentication from authorization, this paper advocates decoupling of the authorization on whether a world call is permitted (by software) from unforgeable identification of calling peers (by hardware). This results in a flexible cross-world call scheme (namely CrossOver) that allows secure, efficient and flexible cross-world calls across multiple layers not only within the same address space, but also across multiple address spaces. We demonstrate that CrossOver can be approximated by using existing hardware mechanism (namely VMFUNC) and a trivial modification of the VMFUNC mechanism can provide a full support of CrossOver. To show its usefulness, we have conducted case studies by using several recent systems such as Proxos, HyperShell, Tahoma and ShadowContext. Performance measurements using full-system emulation and a real processor with VMFUNC shows that CrossOver significantly boosts the performance of the mentioned systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ISCA '15, June 13 - 17, 2015, Portland, OR, USA
© 2015 ACM. ISBN 978-1-4503-3402-0/15/06$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750406

1. Introduction

The virtualization evolution adds more hierarchies vertically to the commodity software stack, which now has at least four protection rings accommodating hypervisors, hypervisor utilities, operating systems and user-level programs. The multi-tenancy nature of virtualization further horizontally introduces multiple protection domains (i.e., virtual machines) in a single computer. Further, the increasing popularity of nested virtualization [20, 4, 46] for usage scenarios like "cloud on cloud", cloud interoperability [41] and new security foundation [46] further complicate the software stack both vertically and horizontally.

The complicated software stack brings both opportunities and challenges. On one hand, programmers have more freedom to leverage the software layers to implement various features, including security [9, 10, 45, 11, 46, 28], decoupling functionalities [17, 41, 18, 1, 28] and improved management [43]. On the other hand, the proliferation of software layers and protection domains (uniformly called *worlds* in this paper) also leads to more complex control transitions across worlds. For example, a cross-VM call usually requires frequent transitions between VMs and the hypervisor (see section 2 for more examples and details). Further, it usually requires non-trivial coding effort, which is complex and hard to reason about.

We attribute such unnecessary overhead and complexity to the lack of flexible cross-world control transitions. Typically, current hardware mechanisms mostly only support calls from user-level programs to operating systems (*syscall*) and those from a VM to the hypervisor (*vmcall*, also called hypercall), as shown in Figure 1. Hence, a ring crossing call from one ring to another one needs to bounce multiple times from/to the hypervisor, which does the authentication by checking the identities of calling peers and the authorization by checking whether such a call is allowed or not. Further, calls that not only cross rings but also protection domains (e.g., address space) make such invocation even more complex.

This paper argues that it is time to rethink the cross-world call mechanisms in contemporary processors with multiple rings and running multiple software layers. Following the design philosophy of separating authentication from authorization, this paper advocates offloading authentication of com-

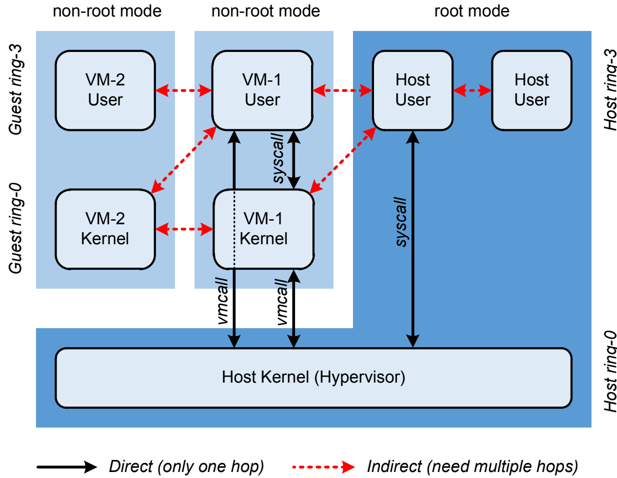


Figure 1: Ring crossing in virtualized machines. Solid lines indicate direct ones supported by current hardware, dashed lines are those indirect ones.

municating worlds into hardware and distributing the authorization into the callee. This results in a flexible cross-world call scheme (called CrossOver) that allows secure, efficient and flexible calls directly across multiple layers not only within the same protection domain, but also across multiple domains.

Specifically, before the first invocation of a cross-world call (world-call for brevity), both the caller and callee ask the hypervisor to register themselves in hardware and each gets an unforgeable world ID. Then they can directly switch from one to another through a new instruction named *world_call* without the involvement of other privileged software components (e.g., the OS kernel for a syscall or an inter-process call, and the hypervisor for a vmcall or a cross-VM call). CrossOver achieves mutual distrust between the caller and its callee, which are isolated in different memory spaces. For each world-call, the hardware will provide a world ID to the callee, by which the callee can further implement authorization policy in software, and the caller can ensure the control flow integrity of call/return by maintaining calling states in its own memory space.

Though there is currently no commodity processor supporting CrossOver, we approximate CrossOver by reusing a recent hardware extension for virtualization. Specifically, CrossOver leverages the *VMFUNC* mechanism [22] to provide intervention-free cross-VM calls. Further, we show that trivially extending the *VMFUNC* mechanism can provide the full support of CrossOver.

We have implemented a prototype of CrossOver based on KVM to support cross-VM calls by leveraging *VMFUNC*. To demonstrate its effectiveness and efficiency, we mimic stripped-down implementation of four recent (close-sourced) systems and use CrossOver to accelerate the cross-VM interactions.

In summary, this paper makes the following contributions:

- A comprehensive study that motivates the necessity of flexible cross-world calls to reduce cross-world switches.
- The design of CrossOver that separates authorization and authentication for secure and flexible cross-world calls.
- A real-world approximated implementation and evaluation of CrossOver using *VMFUNC*.

The rest of this paper is organized as follows. The next section presents a detailed study on recent systems, which motivates the necessity of secure and flexible cross-world calls. Section 3 presents the general design of CrossOver, followed by how to approximate CrossOver by leveraging VMFUNC to provide cross-VM calls. Section 6 describes how to apply CrossOver to several recent systems. Next, section 7 presents the evaluation results of CrossOver and section 8 discusses the related work close to CrossOver. Finally, we discuss the limitations and future work of CrossOver in section 9 and conclude this paper.

2. The Case for Flexible Cross-World Calls

The demand of cross-world calls: With virtual machine monitors (also called hypervisors) being the new system software foundation for commodity software stack, there are more software layers and modes with different privileges. Processor vendors like Intel and AMD have introduced corresponding hardware extensions (VT-x or SVM), by introducing a new set of privilege modes (e.g., VMX root operation in Intel's term). This results in at least four protection rings and multiple address spaces. Similarly, the support of TrustZone and virtualization on ARM platforms also introduce four privilege rings (User, Kernel, Hyp and Monitor Mode).

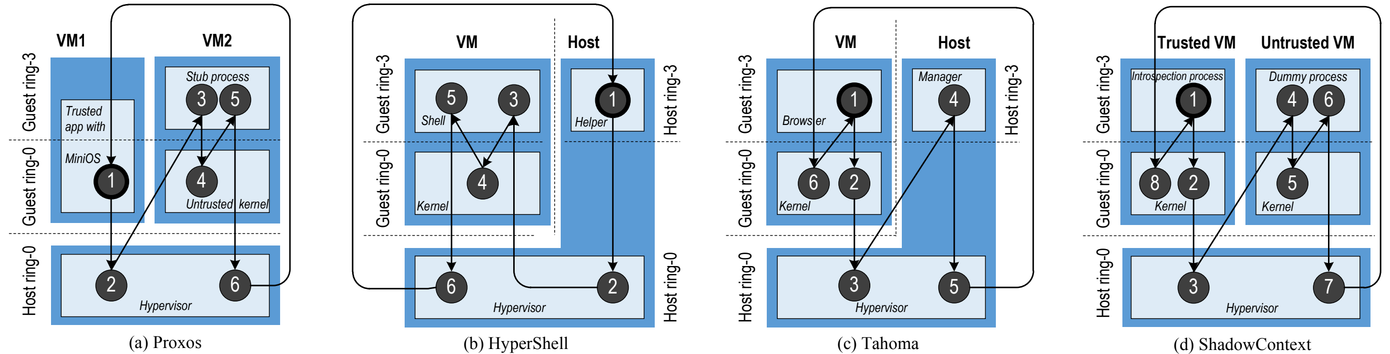
The pervasive deployment of virtualization also enables a number of innovative usages for various purposes, such as security, decoupling and management. Table 1 lists a set of example systems whose core techniques rely on calls that cross multiple protection domains and rings (uniformly called *world* here). With existing mechanisms such as syscall and vmcall, a simple cross-world call usually needs to detour multiple times to/from the most privileged software (e.g., hypervisor). The following uses some example systems to illustrate this issue (also shown in figure 2[1]):

- *Proxos:* Proxos [38] is a system that enables an application to selectively route its system calls between a trusted private OS and an untrusted commodity OS. This system is very useful for isolating some security-sensitive parts (like SSL certificate service) from other non-sensitive parts handling application logic. Hence, Proxos strikes a good balance between security and functionality. The core of Proxos is selectively redirecting system calls between the two OSes (i.e., VMs). However, as shown in Figure 2 (a), redirecting a syscall requires at least 6 ring crossings and context switches. This incurs non-trivial performance

[1] Note that some systems were designed when there is no hardware virtualization extensions like VT-x and SVM. We slightly adjust the designs to fit contemporary processors with VT-x or SVM support.

Table 1: A list of example systems that rely on cross-world call to implement critical functionality.

	System	Description	Semantic	Theoretically Minimal	Actual Cross-ring Calls	Times
Security	Proxos [38]	A system that splits system calls from application to redirect those critical ones to trusted OS. All the system calls are intercepted and redirected by a trusted hypervisor.	syscall	$K_{VM1} \rightarrow K_{VM2} \rightarrow K_{VM1}$	$K_{VM1} \rightarrow K_{host}^{hypervisor} \rightarrow U_{VM2} \rightarrow K_{VM2} \rightarrow U_{VM2} \rightarrow K_{host}^{hypervisor} \rightarrow U_{VM1}$	3X
	Tahoma [13]	A system that use virtual machine to isolate browser. Each web instance runs in a VM, and a manager running in domain-0 controls all instances by cross-VM IPC. In this paper, we use VT-x (which is not available that time) to re-implement Tahoma.	IPC call	$U_{VM} \rightarrow U_{host} \rightarrow U_{VM}$	$U_{VM} \rightarrow K_{VM} \rightarrow K_{host} \rightarrow U_{host} \rightarrow K_{host} \rightarrow K_{VM} \rightarrow U_{VM}$	3X
	Overshadow [11]	A system that protects user applications from untrusted OS. All the system calls are intercepted by a trusted hypervisor for security checking. Two user-level shims are used for interaction between OS and application.	syscall	$U_{VM} \rightarrow K_{VM} \rightarrow U_{VM}$	$U_{VM} \rightarrow hypervisor \rightarrow U_{VM}^{shim-cloaked} \rightarrow hypervisor \rightarrow K_{VM} \rightarrow U_{VM}^{shim-uncloaked} \rightarrow hypervisor \rightarrow U_{VM}^{shim-cloaked} \rightarrow hypervisor \rightarrow U_{VM}$	4.5X
	MiniBox [29]	A two-way sandbox. It uses hypervisor to intercept and selectively redirect system calls from protected applications to a trusted kernel.	syscall	$U_{VM1} \rightarrow K_{VM2} \rightarrow U_{VM1}$	$U_{VM1} \rightarrow hypervisor \rightarrow U_{VM2} \rightarrow K_{VM2} \rightarrow U_{VM2} \rightarrow hypervisor \rightarrow U_{VM1}$	3X
	CloudVisor [46]	A system that uses nested virtualization to protect guest VM from untrusted hypervisor. The original hypervisor runs as a VM, while CloudVisor runs at the highest priority. All the VMExit are intercepted by CloudVisor for security check.	I/O op	$K_{VM} \rightarrow U_{dom0}^{qemu} \rightarrow K_{VM}$	$K_{VM} \rightarrow CloudVisor \rightarrow K_{VM}^{hypervisor} \rightarrow CloudVisor \rightarrow K_{dom0} \rightarrow U_{dom0}^{qemu} \rightarrow K_{dom0} \rightarrow CloudVisor \rightarrow K_{VM}^{hypervisor} \rightarrow CloudVisor \rightarrow K_{VM}$	5X
Decoupling	FUSE [1]	A user space file system. OS intercepts and redirects FS-related system calls to a user space daemon.	syscall	$U_{app} \rightarrow U_{fuse} \rightarrow U_{app}$	$U_{app} \rightarrow K \rightarrow U_{fuse} \rightarrow K \rightarrow U_{app}$	2X
	Emulated devices in Xen [3]	Xen runs a specific VM (dom-0) to manage devices, including real ones and emulated ones (e.g., by Qemu). A guest VM communicates with dom-0 for I/O, which is intermediated by the hypervisor.	I/O op	$K_{VM} \rightarrow U_{dom0}^{qemu} \rightarrow K_{VM}$	$K_{VM} \rightarrow hypervisor \rightarrow K_{dom0} \rightarrow U_{dom0}^{qemu} \rightarrow K_{dom0} \rightarrow hypervisor \rightarrow K_{VM}$	3X
	ClickOS [30]	ClickOS is a Xen-based software platform optimized for middlebox processing. It leverages Xen's backend/frontend driver model and uses miniOS to minimize the performance overhead.	I/O op	$K_{VM} \rightarrow U_{dom0}^{qemu} \rightarrow K_{VM}$	$K_{VM}^{netfront} \rightarrow hypervisor \rightarrow K_{dom0}^{netback} \rightarrow hypervisor \rightarrow K_{VM}^{netfront}$	2X
	Xen-Blanket [41]	A system that use nested virtualization to implement "virtualize once and run everywhere". A nested layer named Xen-Blanket that can homogenize today's diverse cloud infrastructures, e.g., run a KVM VM on Xen platform and vise versa.	I/O op	$K_{VM} \rightarrow U_{dom0}^{qemu} \rightarrow K_{VM}$	$K_{VM}^{ring-1} \rightarrow K_{VM}^{ring-0} \rightarrow K_{guest-dom0}^{ring-1} \rightarrow K_{VM}^{ring-0} \rightarrow hypervisor \rightarrow K_{host-dom0}^{ring-1} \rightarrow U_{host-dom0}^{qemu} \rightarrow K_{host-dom0}^{ring-1} \rightarrow hypervisor \rightarrow K_{VM}^{ring-0} \rightarrow K_{guest-dom0}^{ring-1} \rightarrow K_{VM}^{ring-0} \rightarrow K_{VM}^{ring-1}$	6X
	HyperShell [18]	A VM management tools which uses system call execution redirection to run a host user-level shell on top of a guest kernel.	syscall	$U_{host} \rightarrow K_{VM} \rightarrow U_{host}$	$U_{host} \rightarrow K_{host} \rightarrow K_{VM} \rightarrow U_{VM} \rightarrow K_{VM} \rightarrow K_{host} \rightarrow U_{host}$	3X
VMI	ShadowContext [43]	A VMI tools that uses system call redirection for introspection. The system calls from a trusted VM are selectively redirected to an untrusted VM to execute.	syscall	$U_{VM1} \rightarrow K_{VM2} \rightarrow U_{VM1}$	$U_{VM1} \rightarrow K_{VM1} \rightarrow K_{host} \rightarrow U_{VM2} \rightarrow K_{VM2} \rightarrow U_{VM2} \rightarrow K_{host} \rightarrow K_{VM1} \rightarrow U_{VM1}$	4X



(a) Proxos (b) HyperShell (c) Tahoma (d) ShadowContext

Figure 2: Cross-world calls in existing systems. The theoretically minimal cross-world calls are two, for each case.

overhead due to the additional world switches and locality loss. In fact, a read or write syscall incurs more than 30X overhead (0.45 and 0.42 *u*s vs. 13.51 and 13.24 *u*s). Given that a number of syscalls to the untrusted OS need to be redirected, this may turn into notable performance overhead for system-intensive workload. For example, the average number of instructions executed between two syscalls for BIND, Apache and MySQL are 2,445, 3,368 and 12,435 [36]. For an Apache web server, where the overhead for an empty write for Proxos is 31.5X (39,720 vs. 1,350 cycles), Proxos would incur up to 8X overhead, depending on the percentage of redirected syscalls.

- *HyperShell:* HyperShell [18] is a recent virtual machine management tool that allows a shell (e.g., bash) inside a host machine to execute management utilities (e.g., "ps aux") on behalf of the guest VMs. It significantly eases the

process of managing VMs in a private cloud with a large number of VMs. A key challenge is the semantic gap, by which the management VM cannot understand the semantics inside a guest VM. HyperShell addresses this by introducing a technique called *reverse syscall execution*, by which a syscall issued by a utility from the host is redirected from the host to the guest VM for execution. This, similarly, requires frequent bouncing from the hypervisor, which incurs at least 6 ring crossings and context switches for a single system call. As a result, the reported average slowdown for a set of common utilities is 2.73X.

- *Tahoma:* Tahoma [13] is a browser operating system that provides strong isolation among browser instances by running the browser kernel in the host and running each browser instance inside a guest VM. Hence, Tahoma can contain many attacks to a browser instance inside the VM, without affecting other instances and the browser kernel. One key enabling technique of Tahoma is the RPC between the browser kernel and a browser instance. However, it also requires multiple ring crossings and context switches due to frequent bouncing to the hypervisor, as shown in Figure 2 (C). This makes the cost of interactions between a browser kernel and its instances much more expensive (i.e., compared to a single function call).
- *ShadowContext:* ShadowContext [43] is a virtual machine introspection tool that allows the host to transparently check security variants of guest VMs. The key idea is stealthily creating a dummy process inside a guest VM and redirecting some introspection-related syscalls to the guest VM, which is executed by the dummy process. However, under existing syscall mechanism, this requires multiple bouncing from both the kernel and the hypervisor, causing at least 8 ring crossings and context switches. This turns into non-trivial overhead for each redirected syscalls.

Existing mechanism is NOT flexible: Table 1 lists more systems that require cross-world call mechanisms. The semantics of many calls are simple and clear, which involve transitions from the kernel or user mode in one VM to that in another VM. However, all of them require a non-trivial number of additional bouncing to/from the hypervisor or the OS kernel. Hence, a flexible and efficient cross-world call mechanism can significantly boost the performance and reduce the implementation complexity of such systems.

One may question that bouncing from hypervisor or kernel is necessary for the sake of security and isolation. This is true in general. However, for many cases in Table 1, the hypervisor or OS kernel only simply does the bouncing work (like saving and restoring context), without doing much complex security checks. This is because, many of these systems only require one-way isolation (isolating an untrusted part from a trusted part, not vice versa). For examples, Proxos requires isolating untrusted OS from a trusted OS; HyperShell, ShadowContext, ShadowContext, Xen-Blanket and front/back end drivers in Xen mostly require only isolating guest VMs from the host/management VM. Hence, in such cases, there is little need to check security policy in the hypervisor.

Even if the two systems require two-way isolation, like Minibox [29] and FUSE [1], they can still be done by using separating authentication from authorization. That is, before making a call, a caller and a callee can authenticate themselves by the hardware through the hypervisor or OS kernel. Afterwards, the call can be done by direct authorization between the peers without intervention from the hypervisor before the authentication was revoked.

Hence, to efficiently support many systems on current virtualized stack, providing a flexible cross-world call mechanism is a key to high performance and low implementation complexity. This can be done by separating the authorization on whether a cross-world call is allowed from the authentication of calling peers during the world transition from one peer to another.

3. Cross-world Call

This section first describes the abstraction of world and its hardware embodiment (called world table), and then presents the interfaces and hardware extensions to support cross-world calls (*world-call* for short in the following text).

3.1. Design Principle

Deciding whether a cross-world call is allowed or not resembles deciding a capability in capability-based systems [40, 42]. Traditional capability-based systems usually have to make a tradeoff between flexibility and performance when deciding the types of capabilities and the form of unforgeable token.

Commodity OS and hypervisor use a coarse-grained hardware privilege separation mechanism (e.g., kernel level and user level or root mode and non-root mode) and let the privileged software define capabilities as well as tokens. It is flexible to define various capabilities, such as the permissions to a file. The unforgeability of tokens, like FD or VM ID, are also checked by the privileged software, while the hardware only has to protect privileged software from less privileged ones. However, since each access has to be checked by the privileged software to enforce capabilities, the performance overhead caused by world crossing is inevitably high.

Recent hardware proposals extend hardware to define and enforce capabilities, as well as ensure the unforgeability of token, in the form of fat-pointer and tag table in CHERI [42], HardBound [15], Intel's MPX [2]. While this solution can significantly improve the performance, the types of capabilities are limited by hardware.

CrossOver uses a hybrid approach of the above to support cross-world calls by separating authentication from authorization: the authentication of worlds between callees and callers is done in hardware through unforgeable tokens and the authorization whether a caller is permitted to call a specific callee is done in software (e.g., a callee may refuse a call from a caller

Table 2: Comparision with traditional capability systems

Systems	Capabilities	Token	Enforce capability
CHERI	HW-defined, not flexible	HW-maintained, efficient	HW: Hardware maintains tokens and capabilities, and checks each fat-pointer dereference
File System	SW-defined, flexible	SW-maintained, not efficient	HW+SW: Hardware protects OS and isolates processes, OS checks each file operation by file descriptor
CrossOver	SW-defined, flexible	HW-maintained, efficient	HW+SW: Hardware isolates worlds and provides unforgeable WIDs for each call, callee authorizes

based on tokens). This achieves both flexibility and performance for cross-world calls. On one hand, CrossOver still keeps the coarse-grained hardware privilege separation mechanism and lets the privilege software (e.g., OS or hypervisor) to define different capabilities in a flexible way. On the other hand, CrossOver extends hardware to provide unforgeable tokens so that for each cross-world call, no privileged software is involved, which reduces the time of world crossing and thus improves performance.

3.2. World & World Table

We use a world table to implement the unforgeable token to uniquely identify a world. We define a *world* as an address space in a specific mode. The address space is determined by page tables (including nested page table) and the mode is determined by all privilege modes (e.g., rings, root/non-root modes). Each world also has one entry point address. A name space needs to register itself to the hardware to be a world, which adds an entry in a table named *world table*. Each entry in the table represents the information of a world, and is indexed by a unique WID (World ID). The WID will be used as an unforgeable token for authentication during a cross-world call.

Inspired by the design of software-managed TLB, we place the world table in a region of memory that can be accessed only by the highest privileged software (e.g., the hypervisor). There is a world table cache (like a TLB) accessed by hardware during a cross-world call. The privileged software provides a software interface to create and delete world entries to upper layers (e.g., guest VM). When creating a new world, the hypervisor will create a unique WID for world identification. WID is unforgeable as the hardware will check if WID is valid and accurately represent the caller and callee by checking the world table cache. A hypervisor can limit the number of worlds a VM can create to avoid DoS (deny of service) attacks from a malicious VM.

3.3. World Call & Return

There are multiple choices to design the world-call mechanism. One is using asynchronous call through message passing, i.e., the caller sends a message to the callee and wait for the reply. However, such asynchronous calls have several drawbacks. First, the design may only be suitable for batch-style workload that have a massive amount of threads and syscalls to batch [36], but not for latency-sensitive workloads and workloads with a small number of threads (e.g., equals to #cores). This is because the callee must wait until it is scheduled to run, which could lead to long and unpredictable latency. Second, for data-intensive workloads, switching to another CPU core is not cache-friendly since the caller and callee usually run on different processor cores. Another design choice is using synchronous calls through IPI (inter-processor interrupt). However, such a design will be tightly bound with scheduling mechanisms (in both guest OS and hypervisor) in order to ensure that a specific processor is running the callee just before calling. It means that for each world-call, the caller needs to invoke a privileged operation to the schedulers for such binding, which requires ring crossing itself and is not suitable for our requirements.

CrossOver chooses a non-disruptive synchronous call scheme, just like syscalls or vmcalls. The processor switches states from a caller to its callee and executes the callee code's immediately. This preserves cache locality of the calling process, has very low latency and is non-disruptive to other VMs/applications. Specifically, once an address space has created its world, it can use a new instruction, *world_call*, to switch to a callee's world to execute. When return, the processor still uses *world_call* to change back to the caller's world.

The mechanism of CrossOver enables the mutual distrust between a caller and a callee. First, the two memory spaces are isolated by conventional hardware like MMU and privilege modes. Further, the callee needs to authenticate the caller to enforce access policies, while the caller needs to keep the integrity of control flow to prevent control flow hijacking attacks such as return-oriented attack [33, 5].

World-call setup: A world-call requires some initialization. First, the caller and callee must both have created their own worlds. Second, the caller needs to create a shared memory mapping with the callee to store calling parameters and return data. Such mapping may require vmcalls or syscalls, but it is a one-time effort that only has to be done for the first time. The calling convention can be negotiated by the two communicating worlds during setup and simple parameters can be passed directly through registers for performance. Third, right before issuing *world_call*, the caller needs to save its running states for resuming execution after return, as well as the WID of callee for checking.

World-call: Once a *world_call* is executed, the processor first identifies the caller by using its current WID to search the corresponding entry in the world table with current page table pointers and modes. If no entry is found in the world table, it means that a namespace issues a world call without creating a world first, which will raise an exception to the hypervisor. Then, the processor will look up the callee entry by its WID which is passed as the parameter of *world_call*. Similarly, an exception will be raised if no entry is found in the world table. After that, the processor switches to the callee's environment by changing the page tables and modes, and jumps to the en-

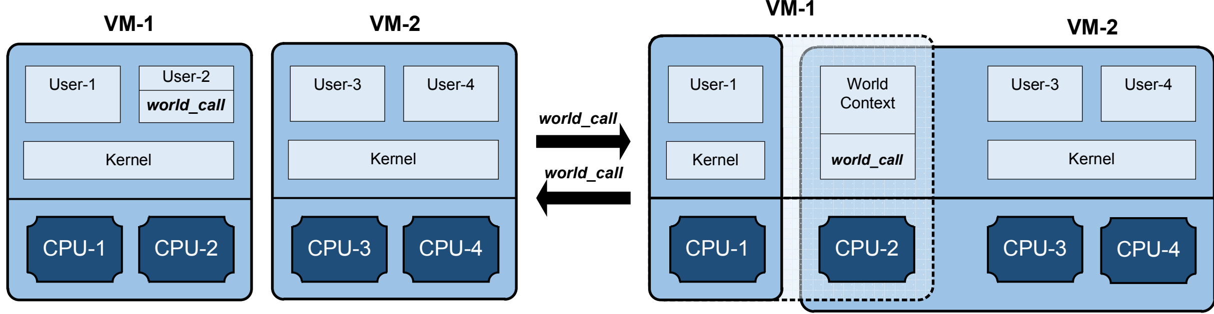


Figure 3: World-call process. The user-2 process in VM-1 calls a world in VM-2. After that, the status of CPU-2 switches from user-2 to the callee and executes the world-call. When the world returns, the status will be back to the left part.

try point address to run. It also passes the caller's WID to the callee (e.g., through a register).

Now the processor is in callee's context and executing the callee's code. Typically, the callee first checks whether the caller is authorized with the caller's WID. Once the authorization is OK, the callee fetches parameters from shared memory, executes service, and puts the results on the shared memory as return value. Finally, it issues *world_call* again with caller's WID as parameter.

World-call return: Again, the processor searches the WID of both the caller and callee, switches to caller's context, jumps to caller's entry point address, and passes callee's WID to it. The caller first checks whether this is a new world-call or a return. If it is a return, the caller just restores all the states and resumes execution. Since the running states are maintained by the caller in its memory space, they are isolated from the callee.

3.4. Discussion

Mutual distrust between caller and callee: The mechanism of CrossOver enables the mutual distrust between a caller and a callee by both software and hardware. The hardware provides two functionalities: isolation between different worlds, and authenticating WIDs during a world switch. The rest is done by software, including authorization and calling flow control, as well as preventing DoS attack.

For callee DoS attack, e.g., a malicious callee that never return, a caller can use timeout mechanism to detect such situation and cancel the world-call by force. Once timeout, the hypervisor will switch the context to the caller and invoke the handler. Thus, the caller can have a chance to cancel the world-call. Since setting up a timeout requires a vmcall to hypervisor, the caller can set a relatively long timer for multiple world-calls to amortize the overhead.

Put calling authorization to hardware: An alternative design is to check the calling authorization in hardware by adding another table, named *binding table*, which records the bindings between callers and callees. This table is also managed by privileged software and checked by the processor. Before calling, a callee first authorizes a legal caller the *calling capability*, and then requires the hypervisor to create an entry in the *binding table* with both WIDs. This binding is needed only once between two worlds. When a caller issues *world_call*, the hardware first checks whether the binding has been established and refuses to continue if not. Thus when the callee is running, it can ensure that the calling has been authorized.

Such a design may further improve the performance of authorization in the callee but may be less flexible. In CrossOver, we minimize the hardware modification and provide only WID for each call. Thus the callee can implement more flexible policies such as offering different services for different worlds by creating only one world in the hardware.

4. Approximating CrossOver with VMFUNC

There are ten types of cross-world calls in modern virtualized architecture, among which only four can be done in one hop, as shown in Table 3. Fortunately, we observe that a new feature introduced recently in Intel's processor, namely *VMFUNC*, can be reused to approximate the functionality of CrossOver. More specifically, by using VMFUNC, we can implement three types of world-calls: $U_{VM1} \rightarrow U_{VM2}$ (user to user on different VMs) in one hop, $K_{VM1} \rightarrow K_{VM2}$ (kernel to kernel on different VMs) in one hop, and $U_{VM1} \longleftrightarrow K_{VM2}$ (between user and kernel on different VMs) in two hops.

Table 3: World-calls Classification

Types	H/G Swtch	Ring Swtch	Space Swtch	Hop (HW.)	Hop (SW.)	Hop (VM-FUNC)	Hop (Cross-Over)
$U_{VM1} \longleftrightarrow K_{host}$	√	√	√	1			1
$K_{VM1} \longleftrightarrow K_{host}$	√	√	√	1			1
$U_{VM1} \longleftrightarrow K_{VM1}$		√		1			1
$U_{host} \longleftrightarrow K_{host}$		√		1			1
$U_{VM1} \longleftrightarrow U_{host}$	√	√	√		3		1
$K_{VM1} \longleftrightarrow U_{host}$	√	√	√		2		1
$U_{host} \longleftrightarrow U_{host}$			√		2		1
$K_{VM1} \longleftrightarrow K_{VM2}$			√		2	1	1
$U_{VM1} \longleftrightarrow U_{VM2}$			√		4	1	1
$U_{VM1} \longleftrightarrow K_{VM2}$		√	√		4	2	1

4.1. VMFUNC in Intel's VT-x

VMFUNC is designed as a general interface to support multiple functionalities specified by different indexes. Cur-

rent processors have only implemented one function (with index 0x0), which enables a guest VM to switch its extended page table (EPT) without triggering any VMExit. With this function, one intended use case is to isolate the running environments of guest kernels from user applications to prevent "return-to-user" attack [24], by switching to a new EPT with the same mapping but different privileges when the kernel starts to run. VMFUNC could be invoked in either kernel or user mode in a guest VM.

The VMFUNC with index 0x0 is implemented as follows: First of all, the hypervisor needs to set up an EPTP list for the guest VM, which is pointed by a new MSR register. Each item in the list points to a valid EPT that the guest VM can use. A guest VM then invokes VMFUNC with function ID (0x0) and an EPT index as parameters. The hardware then switches to the new EPT by searching the EPTP list, and then resumes execution. The hypervisor will not get involved during the entire process after setup. VMFUNC instruction can be used in either user mode or kernel mode.

4.2. Approximating CrossOver using VMFUNC

Other than its conventional use, we found that VMFUNC could be crafted to implement parts of functionalities required by CrossOver. Specifically, for $K_{VM1} \rightarrow K_{VM2}$ and $U_{VM1} \rightarrow U_{VM2}$, both involve neither H/G mode switching nor ring switching, but only address space switching. Hence, it is possible to implement such switches using VMFUNC in one hop. However, since $U_{VM1} \longleftrightarrow K_{VM2}$ needs to switch ring levels, we need at least two hops: one for ring switching and one for EPT switching. With the support of switching VMs without hypervisor intervention, we can implement cross-VM calls by VMFUNC to evaluate its effectiveness.

To approximate CrossOver as much as possible, we use VMFUNC to simulate the following components. First, we use the EPT list in VMFUNC to simulate the world-table in CrossOver. We use one EPT for each simulated world. It is required that the caller and callee must have the same value in CR3 register, since switching EPT will not change CR3. Using VMFUNC cannot switch the H/G mode, ring level or page table root, so it can only implement $U_{VM1} \rightarrow U_{VM2}$ and $K_{VM1} \rightarrow K_{VM2}$. $U_{VM1} \rightarrow K_{VM2}$ is based on $K_{VM1} \rightarrow K_{VM2}$, with an additional ring switch from U_{VM1} to K_{VM1}.

Second, we construct a helper code snippet to implement the process of world call and return. The code of helper is mapped to the same virtual address region of the caller and callee, so that after EPT switching, the PC will continue to execute the next instruction smoothly. The function call and return are both implemented by executing VMFUNC, with different directions. Third, we create a shared memory region in a commonly mapped virtual address range for parameter passing. The caller will set up the region with the callee in advance.

4.3. Case: Supporting Cross-VM System Call

Cross-VM system call mechanism provides an application the ability to issue a system call service that is implemented in another VM [38, 29, 18, 43]. Here we show how CrossOver could improve performance by reducing world-calls.

The minimal cross-world path for cross-VM system call is: $U_{VM1} \rightarrow K_{VM2} \rightarrow U_{VM1}$. A direct world-call of $U_{VM1} \rightarrow K_{VM2}$ requires switch ring levels, VM contexts and address spaces. To this end, we need to modify the ring level, the EPT pointer and the CR3 register correctly to those in another VM. A valid CR3 value means there is a corresponding page table pointed by the CR3 value in another VM, while a valid EPT value means the corresponding entry indicated by an offset of the VMFUNC instruction in EPTP-list address is an EPT pointer of another VM, and changing the CR3 register and the EPTP pointer must be done in order. However, since changing the CR3 register could be done only in privileged level 0, we have to do a ring-crossing to K_{VM1} first. The actual execution context path is: $U_{VM1} \rightarrow K_{VM1} \rightarrow K_{VM2}$. Since cross-VM system call requires seamless code execution across different address spaces, we map a non-writable code page to the same guest physical address in kernel space of each process in a VM during process creation time so that changing address space does not require loading and storing all context information.

Figure 4 illustrates how the cross-VM system call executes. To invoke a cross-VM system call, an application needs to specify which VM it wants to call. To this end, after a VM boots up, the hypervisor will assign a unique VM ID to each VM and keep track of each VM's EPT pointer by storing it in the EPT-list address with an offset, which is the same as the VM ID. We provide a hypercall for applications to query all the existing VMs and their own VM ID. A cross-vm system call will firstly be intercepted by a syscall dispatcher. The syscall dispatcher will issue a special system call and jump to the cross-world code page mapped earlier. The cross-world code will first change the page table to that of a helper context, which is created during VM boot and its page table entry has the same guest physical address in all VMs so that after changing the EPT pointer to another VM, this helper context could still continue execution seamlessly.

This helper context is designed to do cross-world context maintenance and execute the designated system call. When its page table is loaded, the helper context saves current execution context and prepares the necessary calling information in another inter-VM shared memory page located in user space. After that, the helper context will invoke the VMFUNC instruction to switch to the paging-structure of the designated VM. A syscall dispatcher in the designated VM will execute the system call and put the necessary returned buffer to the user space inter-VM shared memory page. The return path of cross-VM system call is similar: restoring the current process context, switching back to the original VM helper context by invoking VMFUNC.

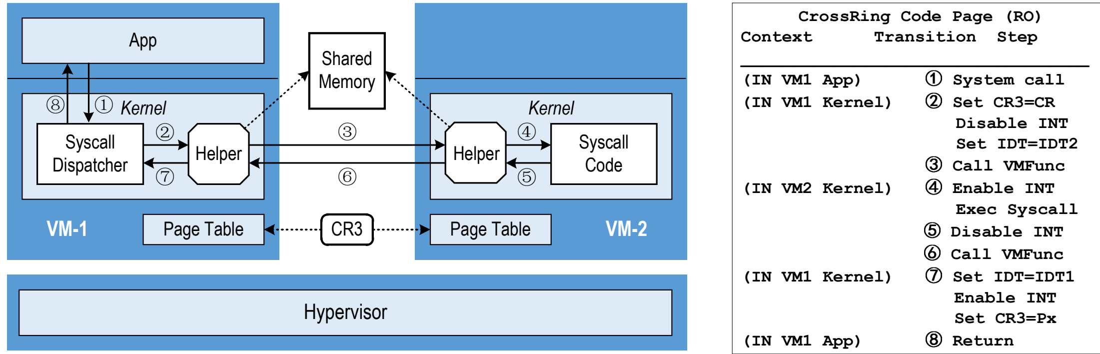


Figure 4: Cross-VM system call process.

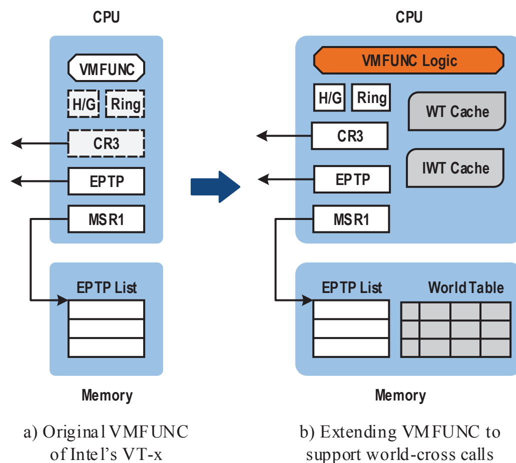


Figure 5: Extending VMFUNC to implement CrossOver. Added elements are grey and extended elements are orange

5. Supporting CrossOver

Though a clever use of VMFUNC may support a part of cross-world calls, it also has several limitations. First, as mentioned before, VMFUNC does not support changing of ring level, H/G mode and page table root. Second, it does not has the semantics of world call and return, thus requires both the caller and the callee to be aware of the call address and return address. One possible problem is that a malicious callee may return to arbitrary address within the caller's memory space, thus we have to assume that both the caller and callee cooperate with each other. Third, it is not aware of *world context*, thus the software needs to maintain such information.

This section first describes how to extend the processor support of VMFUNC to fully support cross-world call for x86, and then describes a full-system implementation of the mentioned mechanisms in QEMU.

5.1. Extending VMFUNC to Support CrossOver

We propose a design by extending the current VMFUNC mechanism, as shown in figure 5. There are two major components: the data path for the world table, and the processing logic for world call/return.

World table: The format of the world table is shown in figure 6, the semantics of each field is listed as following:

- **P** is the present bit, which means whether the current entry is active or not.
- **WID** is a world ID that is used to identify a specific world.
- **H/G** is a bit to show whether the world is in host mode or guest mode.
- **Ring** is the ring level of the world.
- **EPTP** is a pointer to the world's extended page table, which is a host physical address.
- **PTP** is a pointer to the world's page table, which is a guest physical address.
- **PC** is the entry address of the world, which is a guest virtual address. Each world has only one entry point.

During a *world_call*, the processor needs to look up thw world table twice, one is to look up the world ID of the caller based its context (H/G, Ring, EPTP and PTP), the other is to look up the callee's context using a world ID. To accelerate such lookup operations, two caches for the world table, *WT Cache* and *IWT Cache* (Inverted World Table Cache), are added. WT Cache is keyed by WID and is mainly used for finding callee context during a *world_call*, and IWT Cache is used for finding the world ID of the caller, which is keyed based on H/G, Ring, EPTP and PTP.

An alternative design that may further improve performance is to add a hardware controlled register called *Current World ID* that store the world ID of the current context that is reloaded by the CPU automatically after context switches, which is similar to hardware cache prefetch mechanism. As this reloading process is not in the critical path during a world call, the latency of a world call could be reduced. This design, however, may be not feasible when only a few worlds create their world entries. In that case, prefetching a non-existed world at every context switch will cause cache miss and useless world table walk.

For caches and world table management, one possible design choice is to let the hardware manage cache consistency and world table walk after a cache miss. However, this design requires non-trivial hardware changes. In CrossOver, both the WT-Cache and IWT-Cache are managed by software, similar as software-managed TLB. More specifically, a privileged software (e.g., the hypervisor) is responsible for maintaining

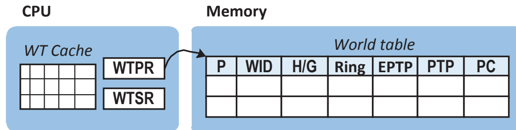


Figure 6: World table structure

the consistency of the two caches, including filling, eviction and invalidation. When a cache miss occurs during a world call, an exception will be raised and caught by the privileged software, which will fill the cache entry by walking the world table. Since most of world calls will not be intervened by the privileged software and thus the design has little performance impact. Further, the policies of cache filling and eviction are more flexible, since they can be adjusted by the privileged software according to the interaction patterns to achieve high hit rate.

Support world_call: We implement *world_call* and *manage_wtc* as new functions to VMFUNC, with indexes as 0x1 and 0x2, respectively. For *world_call*, instead of only changing the EPT pointer, the processor will also change the address space and processor mode to the target world, and jump to the entry point indicated by the world table. Before that, software needs to save the caller's states to its world stack maintained by itself. For *manage_wtc*, it contains two world table cache operations, namely world table cache entry filling and world table cache invalidation, according to the parameter provided.

As shown in figure 5 (b), the per-core overhead of extending VMFUNC to support CrossOver is trivial: two small world table caches (WT Cache and IWT Cache) and the instruction logics for two new instructions. Thus, this may be added to existing processor with small changes and without interacting with other core CPU logic like cache hierarchy and coherency.

5.2. Full-system Emulation

To evaluate the functionality of CrossOver, we have implemented the functionality of CrossOver based on QEMU (version 2.2). Currently, QEMU simulates CPU virtualization by modeling AMD's SVM architecture. Since QEMU does full-system simulation by translating the target platform opcodes to host's ones, instructions introduced by CrossOver are implemented by extending new operands in INT instruction. QEMU will recognize such special instructions in the translation phase and translate them into several target opcodes, where we check and updat the world table cache as well do world switches. Further, we also implemented a few handlers to catch and handle world table related exceptions, like looking up world table in memory and filling the world table cache during a world table cache miss. In total, we added around 500 LOCs in QEMU to support CrossOver.

5.3. Software Support for CrossOver

The design of CrossOver extends currently vertical calling abstraction (i.e., syscall and vmcall) to both horizontal and vertical directions. To support CrossOver, the system software needs to have several minor changes. Most of the changes are about OS awareness. A *world_call* will change the current process without OS awareness. Consider following scenario: process-*a* *world_call*s process-*b* on the same VM. The CPU will change its PC and page table register to process-*b*. However, after the call, the OS still thinks that the current running process is process-*a*. Thus, if there comes a timer interrupt that further triggers a context switch, the OS will save process-*b*'s context to the data structure of process-*a*. This will lead to an unrecoverable state and may cause an unexpected exception. Moreover, Linux Kernel has some optimizations in single core mode as a way of preventing unnecessary lock holdings. However, after a *world_call* there are more than one CPU executing the same piece of code, which may cause deadlock. Meanwhile, the software is responsible for maintaining states of itself as a caller or states of all the callers coming from other worlds (as a callee).

Specifically, in our implementation on xv6 [14], we make the OS scheduler aware of *world_call* by reloading the process state before a context switch and avoid rescheduling a process which is being executed by a "new" CPU from a world call. We use a list of caller and callee stacks to maintain the states of cross world calls. Currently, our software implementation does not support concurrent cross world call from one world. For Linux kernel, the deadlock problem is bypassed by preventing more than one vcpu with the same ID from executing the same piece of code.

6. Usage Scenarios

In this section, we describe four systems relying on world-calls and how they are implemented with CrossOver using VMFUNC. As we do not have the source code for these four systems, we reimplemented the core functionalities of the original systems both with and without the optimization enabled by CrossOver[2]. We also discuss the world switch reduction of these systems in our implementation.

Case Study 1: Proxos Proxos is implemented in Xen as a libOS linked with each private application and there is a host process in the untrusted OS. Each redirected system call will first trap to the VMM using a hypercall and the hypervisor injects the system call to guest VM host process with a virtual interrupt. The redirected system call will be enqueued to the host process descriptor and executed when the host process is scheduled. After the system call execution, the VMM will be notified only when the guest OS is running out of runnable processes or preempted by VMM. Hence, the num-

[2]The re-implemented version could be accessed through http://ipads.se.sjtu.edu.cn/projects/crossover.html

Table 4: Microbenchmark Results of Cross-World Systems with/without Optimization

Benchmark	Guest Native Linux (μs)	Proxos			HyperShell			Tahoma			ShadowContext		
		Original (μs)	Optimized (μs)	Latency Reduced	Original (μs)	Optimized (μs)	Latency Reduced	Original (μs)	Optimized (μs)	Latency Reduced	Original (μs)	Optimized (μs)	Latency Reduced
NULL system call	0.29	3.35	0.42	87.5%	2.60	0.72	72.3%	42.0	0.68	98.4%	3.40	0.71	79.1%
NULL I/0	0.34	2.44	0.50	85.5%	2.57	0.80	68.9%	42.6	0.72	98.3%	3.67	0.79	78.5%
open & close	1.38	8.18	1.91	76.7%	6.03	2.29	62.0%	89.1	2.21	97.5%	7.52	2.26	70.0%
stat	0.55	4.31	0.69	84.0%	2.87	0.98	65.9%	43.5	0.94	97.7%	3.69	0.99	73.2%
pipe	3.34	15.79	4.73	70.0%	13.1	4.99	61.9%	172.6	4.95	97.1%	17.10	5.02	70.6%

ber of world-calls in Proxos during a system call redirection is 6 (Figure 2(a)).

We implement the core functionalities of Proxos in KVM, which follows the design of Proxos' paper and requires six world-calls. To redirect the system call, we write a kernel module, which acts as a system call dispatcher, that intercepts the necessary system call and redirects them to another VM. Another optimized version using VMFUNC is implemented without any ring-crossing. The internal implementation is similar to cross-vm system call in section 4.3.

Case Study 2: HyperShell The redirected system call in HyperShell will be handled by the host OS (or KVM) and KVM will inject the system call to a helper process in the designated guest VM when the helper process traps to KVM. On finishing the system call execution, the helper process will trap to KVM again and let KVM resume execution of the host user-level shell. The helper process keeps executing INT3 instruction trapping to KVM so that the redirected system call could be handled timely. We implemented the core functionalities according to HyperShell's paper.

We find that strictly following the original design using VMFUNC could lead to a security hole: after switching a host to a guest, CPU executes a guest VM with host privilege. We remedy this problem by implementing HyperShell in a guest VM instead of a host OS. This provides better security isolation for the HyperShell utilities. Note that, with the full design of CrossOver, we can completely avoid this problem. Now to execute a redirected system call, the original HyperShell requires 8 world-calls and the optimized one using VMFUNC only requires 4 world-calls.

Case Study 3: Tahoma Each web instance in Tahoma runs in a VM, and a manager running in domain-0 controls all instances by cross-VM RPC. The cross-VM RPC is implemented as XML-formatted and carried over a TCP connection using point-to-point virtual network link. Instead of implementing a full prototype of Tahoma, we only implement the communication between manager VM and browser instances (called browser-calls) of Tahoma in KVM.

Case Study 4: ShadowContext The system calls from a trusted VM are selectively redirected to an untrusted VM to execute in ShadowContext. We implement a prototype of ShadowContext according to the paper and another world-call optimized version using VMFUNC. The original cross-world path of ShadowContext is like this: a user space application in a trusted VM issues a cross-vm system call, which will be introspected by an introspection interface in kernel and this introspection interface will raise a VMExit to trap to KVM. KVM will create a dummy process and inject the redirected system call to the dummy process with a software interrupt. Another VMExit will be raised when the redirected system call is done and the original guest introspection process will be awakened and resumed. In the above steps, all the necessary parameters and buffers are copied in and out across VMs. In our implementation, we further apply an optimization through avoiding copying all parameters and buffers by using inter-VM shared memory.

An optimized version of ShadowContext is implemented using VMFUNC, which directly reuses the design and implementation of the cross-VM system call: a redirected system call issued by a user space application, will be intercepted by the syscall dispatcher in kernel space. The syscall dispatcher will switch to the untrusted VM directly using VMFUNC and as a result, the execution context becomes the untrusted VM kernel space. The redirected system is processed in place, then another world switch will switch execution context back to the trusted VM kernel space and return back to user space.

7. Evaluation

This section conducts performance evaluation of CrossOver on a real hardware and QEMU. The platform is an Intel Core i7-4770 Haswell machine with four cores. The machine runs at 3.40 GHz with 32 GB memory. The host OS is 64-bit Debian 7.0 with Linux kernel 3.16.0-rc4+ and the guests run 64-bit Ubuntu 14.04 with Linux kernel 3.16.1. All guest VMs are configured with one virtual CPU and 2GB memory. The QEMU version is 2.2; the host and guest Linux is 32-bit, the same version as the native evaluation.

7.1. Evaluation on Real Hardware

We evaluate the efficiency of the VMFUNC mechanism in overhead reduction for cross-world calls using the four systems we implemented in section 6.

7.1.1. Microbenchmarks Table 4 shows the latency of the tested benchmark. The system call forwarding overhead of Proxos ranges from 3.4X to 8.4X while the optimized version using VMFUNC is 1.4X to 1.5X, which, as a result, brings about 80% latency reduction. Note that the overhead of Proxos in its paper evaluation could be up to 35X due to the delay required to schedule the VM and the app to run. The performance improvement in our prototype mainly comes

from hardware feature improvement and software optimization enabled by CrossOver.

The overhead for Tahoma is much higher than the optimized one using CrossOver, since Tahoma uses point-to-point virtual network link TCP connection based RPC as communication channel. With the optimization, the overhead for inter-VM communication is reduced by over 97%.

The original system call redirection overhead without cross-world call optimization in ShadowContext ranges from 4X to 11X compared with the native Linux. The overhead of system call redirection using VMFUNC, not surprisingly, does not exceed 2X and the average system call redirection latency reduction is about 73%. Similarly, the overhead of HyperShell is reduced by around 75%.

Table 5: Evaluation Result of 6 Utility Tools

Utility	Guest Native Linux (ms)	Cross-World w/o CrossOver (ms)	Cross-World w/ CrossOver (ms)	Overhead Reduction
pstree	6.00	26.32	8.40	68.1%
w	3.78	20.00	5.58	72.1%
grep	0.93	3.50	1.57	55.1%
users	1.00	3.67	1.63	55.6%
uptime	1.09	6.97	1.85	73.5%
ls	1.14	6.55	1.72	73.7%

Table 6: Evaluation Result of OpenSSH

File Size (MB)	Guest Native Linux (MB/s)	Cross-World w/ CrossOver (MB/s)	Cross-World w/o CrossOver (MB/s)	Throughput Improvement
128	64	42.7	25.6	67%
256	64	42.7	23.3	83%
512	56.9	42.7	23.3	83%
1024	53.9	44.5	23.3	91%

7.1.2. Application Benchmarks Utility Software: One scenario with cross-world call is to do VM introspection (e.g., ShadowContext) or VM management (e.g., HyperShell) using a shell. We show how CrossOver improves the overall performance of such systems. Table 5 shows the results using 6 common utility tools that inspect states of another VM, using hypervisor intervenes the system call redirection and with CrossOver respectively. Specifically, we redirected all the system calls of these utilities to another VM and executed the utilities each with 10 times and computed the average time of the execution.

As shown in the right column of Table 5, CrossOver leads to an overhead reduction ranges from 55% to 73%. The reduction varies due to the proportion of time system calls spend in each utility. Note that the synchronous nature of CrossOver could lead to even larger overhead reduction when the target VM has a higher load.

OpenSSH Server: Cross-world call is useful to achieve security isolation while keeping a good performance. For example, one can place security-critical operations in a security VM, while leaving the common functionality in a private VM. This has been studied extensively in the virtualization and trusted computing community [19, 38].

To demonstrate the effectiveness of CrossOver in protecting secrecy of sensitive data with low overhead, we partitioned the OpenSSH server system calls into two parts according to whether it will access the private key and those system calls related to the private key and the user land code are executed in a private VM, while all other system calls such as network operations are still in a public VM. This is done in a manner similar to how Proxos separates the OpenSSH server.

We used CIL [32], a static analysis tool to find out functions that will access the private key in the *crypto* library in OpenSSL and manually instrumented all the system calls that needed to be executed in the private VM in those functions. To evaluate the average overhead, we ran an SSH client tool called *scp* in the host to connect the the OpenSSH server and copy security-sensive files ranging from 128MB to 1GB in size from the server. The files locate in the private VM (already cached) and will be encrypted before sending to the client through the public VM.

Table 6 shows the average througput of 10 runs. Since *scp* is not a syscall-intensive application but involves a number of memory and network operations, the performance improvement is smaller than micro benchmark results. Still, CrossOver enjoys more than 67% performance speedup compared to the original ones. Note that, again, with the load of the private VM increases, the performance of the hypervisor-based cross-world call drops rapidly because of frequent context switches, while the performance of CrossOver would be largely not affected.

Table 7: Instruction Count in QEMU using LMbench3

Benchmark	Native Linux	Cross-World w/ CrossOver	Cross-World w/o CrossOver
getppid	1847	1880	2575
stat	1224	1257	3308
read	482	515	2337
write	439	472	2200
fstat	494	527	2321
open/close	1924	1957	3931

7.2. Evaluation on QEMU

As currently there is no cycle-accurate full-system simulator with virtualization extension support, we use QEMU as a full-system simulator to collect the instruction counts for the LMbench3 benchmark suite [31] with and without CrossOver. By default, we forward the tested system calls involved in each application. As shown in Table 7, CrossOver only incurs 33 additional instructions for each application (which only has one system call redirected), including the *world_call* instruction, several instructions to save and restore stack and parameter passing. Note that, as the corresponding world has already been loaded into the world table cache, there is no world table cache miss during the process. Besides, stacks are all pre-allocated in initialization phase and thus are not counted here. Moreover, software didn't authenticate the caller during this evaluation.

8. Related Work

Cross-layer design: CrossOver continues the line of cross-layer designs to provide performance and security [23, 8, 46, 37, 44]. Specifically, CrossOver advocates the need of efficient cross-world calls in the form of crosscutting interfaces [12], with the proliferation of protection rings and domains. We believe the design philosophy of CrossOver aligns with the support for virtualization from hardware vendors, which reduces the intervention from hypervisors. For example, the evolution of Intel's VT-x progressively relaxes more conditions to be configurable to cause VMExit [22]. Further, the design of single-root I/O virtualization (SR-IOV) also reduces hypervisor's intervention by separating control from execution, which notably reduces the number of VMExits [16]. One of the goals of the VMFUNC mechanism is providing more flexible mechanism for a VM to control its execution. CrossOver can be considered as a case of virtualizing inter-VM communication, which can be implemented efficiently by extending the existing VMFUNC mechanism.

Optimizing cross-layer calls: There have been several efforts in optimizing communication in virtualized environments [25, 6, 34, 21, 30]. For example, XWAY optimizes inter-VM socket performance through bypassing TCP/IP stacks, avoiding page flipping, and providing a direct communication path between VMs in the same machine [25]. Fido [6] further avoids the cost of parameter marshaling by unifying the address space layout between two communicating VMs. Shuttle [34] instead weakens the isolation property among VMs to optimize intra-VM communication for OS-level virtualization systems. However, all these approaches still require bouncing from the hypervisor, and thus incur additional world switches. In contrast, CrossOver aims at providing direct switches between two communicating peers, yet still preserves isolation by separating authentication from invocation.

Capability-based systems: There have been various researches on capability-based systems, either implemented in either hardware [42, 2, 27, 7] or software [26, 35]. As mentioned before, these systems define and enforce various kinds of capabilities in either software or hardware, which has to make a tradeoff between flexibility and efficiency. CrossOver tries to provide a hardware-software co-design solution. In some sense, CrossOver can be viewed as a hardware extension providing a new capability of cross-world call which supports up-level software to implement authorization and provide various services across worlds without the intervention of other privileged software to improve performance. CODOMs [39] provides efficient protection among multiple software components that share the same address space in a capability manner, while CrossOver allows secure, efficient and flexible cross-world calls across multiple layers not only within the same address space, but also across multiple address spaces.

9. Conclusion

This paper described CrossOver, a cross-world call mechanism that provides intervention-free cross-world calls. We leverage a most-recent hardware feature, namely VMFUNC, to implement parts of functionality needed for CrossOver. Finally, we also show how existing VMFUNC mechanism could be trivially enhanced to fully enable CrossOver. Case studies using four recent systems show that CrossOver can significantly improve their performance due to notably reduced world switches.

In our future work, we plan to extend a full-system cycle-accurate simulator with the support of VT-x and CrossOver's design. By this mean, we plan to conduct a cycle-level evaluation of the performance of the full design of CrossOver. Further, we plan to explore more usages scenarios where CrossOver could be helpful and identify possible implication of CrossOver on system design.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work is supported in part by the Program for New Century Excellent Talents in University, Ministry of Education of China (No. ZXZY037003), the National Natural Science Foundation of China (No. 61303011), a foundation for the Author of National Excellent Doctoral Dissertation of PR China (No. TS0220103006), the Shanghai Science and Technology Development Fund for high-tech achievement translation (No. 14511100902), a research grant from Intel and the Singapore NRF (CREATE E2S2). Prof. Haibing Guan is the corresponding author.

References

[1] "Filesystem in userspace," http://fuse.sourceforge.net/.

[2] "Introduction to inte memory protection extensions," http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, 2013.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[4] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization," in *Proc. OSDI*, 2010.

[5] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: generalizing return-oriented programming to risc," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 27–38.

[6] A. Burtsev, K. Srinivasan, P. Radhakrishnan, K. Voruganti, and G. R. Goodson, "Fido: Fast inter-virtual-machine communication for enterprise appliances." in *USENIX Annual technical conference*. San Diego, CA, 2009.

[7] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," in *ACM SIGPLAN Notices*, vol. 29, no. 11. ACM, 1994, pp. 319–327.

[8] D. Champagne and R. B. Lee, "Scalable architectural support for trusted software," in *Proc. HPCA*, 2010, pp. 1–12.

[9] H. Chen, J. Chen, W. Mao, and F. Yan, "Daonity-grid security from two levels of virtualization," *Information Security Technical Report*, vol. 12, no. 3, pp. 123–138, 2007.

[10] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, P. Yew, and W. Mao, "Tamper-resistant execution in an untrusted operating system using a virtual machine monitor," *Parallel Processing Institute Technical Report, Number: FDUPPITR-2007-0801, Fudan University*, 2007.

[11] X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, C. Waldspurger, D. Boneh, J. Dwoskin, and D. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *Proc. ASPLOS*. ACM, 2008, pp. 2–13.

[12] T. C. C. Consortium, "21st century computer architecture: A community white paper," http://www.cra.org/ccc/files/docs/init/21stcenturyarchitecturewhitepaper.pdf, 2012.

[13] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy, "A safety-oriented platform for Web applications," *Security and Privacy, 2006 IEEE Symposium on*, pp. 15–364, 2006.

[14] R. Cox, M. F. Kaashoek, and R. Morris, "Xv6, a simple unix-like teaching operating system," 2011.

[15] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the c programming language," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 103–114, 2008.

[16] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010, pp. 1–10.

[17] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe hardware access with the xen virtual machine monitor," in *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[18] Y. Fu, J. Zeng, and Z. Lin, "Hypershell: a practical hypervisor layer guest os shell for automated in-vm management," in *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*. USENIX Association, 2014, pp. 85–96.

[19] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *Proc. SOSP*. ACM, 2003, pp. 193–206.

[20] R. Goldberg, "Architecture of virtual machines," in *Proceedings of the workshop on virtual computer systems*, 1973, pp. 74–112.

[21] J. Hwang, K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). Seattle, WA: USENIX Association*, 2014, pp. 445–458.

[22] I. Intel, "Intel-64 and ia-32 architectures software developer's manual," *Volume 3A: System Programming Guide, Part*, vol. 1, 64.

[23] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, "Nohype: virtualized cloud infrastructure without the virtualization," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2010, pp. 350–361.

[24] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kguard: Lightweight kernel protection against return-to-user attacks." in *USENIX Security Symposium*, 2012, pp. 459–474.

[25] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim, "Inter-domain socket communications supporting high performance and full binary compatibility on xen," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2008, pp. 11–20.

[26] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 207–220.

[27] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight Jr, and A. DeHon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 721–732.

[28] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, "Building trusted path on untrusted device drivers for mobile devices," in *Proc. APSys*, 2014.

[29] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "Minibox: A two-way sandbox for x86 native code," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, 2014.

[30] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). Seattle, WA: USENIX Association*, 2014, pp. 459–473.

[31] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.

[32] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *Compiler Construction*. Springer, 2002, pp. 213–228.

[33] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.

[34] Z. Shan, X. Wang, T.-c. Chiueh, and X. Meng, "Facilitating inter-application interactions for os-level virtualization," in *ACM SIGPLAN Notices*, vol. 47, no. 7. ACM, 2012, pp. 75–86.

[35] J. S. Shapiro, "Eros: A capability system," Ph.D. dissertation, University of Pennsylvania, 1999.

[36] L. Soares and M. Stumm, "Flexsc: Flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–8.

[37] J. Szefer and R. B. Lee, "Architectural support for hypervisor-secure virtualization," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 437–450, 2012.

[38] R. Ta-Min, L. Litty, and D. Lie, "Splitting interfaces: Making trust between applications and operating systems configurable," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 279–292.

[39] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero, "Codoms: protecting software with code-centric memory domains," in *Proceeding of the 41st annual international symposium on Computer architecture*. IEEE Press, 2014, pp. 469–480.

[40] D. Wentzlaff, C. J. Jackson, P. Griffin, and A. Agarwal, "Configurable fine-grain protection for multicore processor virtualization," in *Proc. ISCA*, 2012.

[41] D. Williams, H. Jamjoom, and H. Weatherspoon, "The xen-blanket: virtualize once, run everywhere," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 113–126.

[42] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," 2014.

[43] R. Wu, P. Chen, P. Liu, and B. Mao, "System Call Redirection: A Practical Approach to Meeting Real-world Virtual Machine Introspection Needs," in *DSN*, Jun. 2014, pp. 1–12.

[44] Y. Xia, Y. Liu, and H. Chen, "Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks." in *Proc. HPCA*, 2013, pp. 246–257.

[45] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," in *Proc. VEE*, 2008, pp. 71–80.

[46] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proc. SOSP*, 2011, pp. 203–216.

All in-text references underlined in blue are linked to publications on ResearchGate, letting you access and read them immediately.