Safe Execution of Untrusted x86 Machine Code Using Hardware-Assisted Virtualization

Yi-Fan Zhang

January 2017

Abstract

The ability to confine user-space software is critical for protecting users from malicious or dubious application behavior. With that in mind, we designed and implemented *Hvexec*, a virtualization layer that is interposed between applications and the kernel. Clients of *Hvexec* are given an API which they can use to monitor, filter or transform system calls from any application. To achieve this without requiring re-compiling or editing the application binary, we implemented *Hvexec* using an atypical application of hardware-assisted virtualization where the restricted software executes in the context of a hardware virtual machine. In this paper, we describe the benefits of using virtualization, the design and lessons learnt from our implementation and benchmarks to weight the benefits against the overhead.

1 Introduction

Local operating system security has traditionally focused on policing user interactions on multi-user systems [1]. However, in recent years, the pervasive use of personal computing along with the convenience of downloading applications from the Internet has shifted the focus from restricting user privileges to limiting capabilites per application process — sometimes referred to as sandboxing, application compartmentalization or process mitigation [1]. While conventional OS access controls based on notions such as users and groups have proven to be valuable, they become awkward tools for applying security policy definitions on a per application basis [1]. Although numerous approaches to solve this particular problem have been proposed (see related works), research into the best designs for applying sandboxing to software along with methods for decreasing overhead, remains active [4, 3, 1].

In this paper, we propose *Hvexec*, an application sandbox designed to protect users from malicious or dubious application behavior. To his end, we designed *Hvexec* against these principles:

- **Resource Control:** Any access to system resources must be delegated through the sandbox.

- **Fidelity:** Applications must be loaded as native executables with no modifications to the binaries (no recompilation or binary translation) and behave equivalently as running outside the sandbox.
- **Flexibility:** Users should be given an API to program custom policies while the burden of enforcement falls upon the sandbox.
- **Performance:** Sandboxed applications should run as close as possible to unsandboxed speed and performance must not degrade significantly as more concurrent sandboxed processes are started.
- **Safety:** The sandbox must not weaken, circumvent or introduce new vulnerabilites into conventional OS access controls.

To help meet those requirements, our sandbox incorporates strategies from various related works. At the foundation, we based *Hvexec* on an atypical application of hardware virtualization extensions (Intel VT-x) where the sandboxed software (guest) executes in the context of a hardware virtual machine (HVM). The hardware enforced nature of the virtual machine (VM) implicitly provides the advantage that the guest runs completely out-of-band with the host OS software. By simulating a complete hardware environment within the VM, the guest has its own set of processor registers, physical memory and privilege modes. More importantly, this requires the guest to cross a VM to host boundary before it can access any host resources — similar to the hardware protection facilities provided for switching between user and kernel mode which is fundamental to implementing a reliable and secure operating system. We exploit this property by dividing *Hvexec* into two parts: one runs as a user process on the host and another in kernel mode in the VM. This assures that all guest system calls and accesses to privileged memory ranges can be trapped, and by running *Hvexec* as an unprivileged user process on the host, we implicitly subject the entire sandbox system to existing OS access control restrictions. As part of the fedelity requirement, sandboxed applications must also not be able to detect the presence of the sandbox. Otherwise, applications may choose to deny the user services in order to coerce permissions to be granted.

Aside from providing resiliance and isolation, *Hvexec* improves on prior work in terms of flexibility and performance. In terms of flexibility, *Hvexec* exposes a user programmable API for defining security policies rather than a prescribed system wide security model commonly seen in previous approaches. This returns control to the users while future proofing the system against the rapid changes in application development. Furthermore, we mitigate the risk of introducing new system vulnerabilities by leveraging existing kernel support for virtualization hardware such that no additional kernel modules need to be installed and no administrator privileges are needed to run *Hvexec*. Although programs can execute natively within a VM, frequent context switching between VM and host can degrade performance. We address this problem with a novel application of exitless notifications [5] for VM and host communication that avoids context switching entirely.

To validate our design, we provide an implementation of *Hvexec* built on OS X 10.11.4 — an operating system with a rich set of user applications and kernel support for HVM. For our evaluation, we weighed the benefits provided by *Hvexec* against the overhead using a suite of benchmarks on graphical and commandline applications as well as microbenchmarks simulating worst case performance scenarios. While the total virtualization overhead can be amortized over the total execution of the program [4, 3], the immediate affect on latency sensitive applications has not been measured. Similarily, the feasibility of scaling to hundreds of HVM sandboxed proccesses in is not known. Thus, we also focused our investigation on aspects of scalability and context switching latency that were not previously examined.

In summary, we have proposed an application of hardware-assisted virtualization aimed at solving the problem of software confinement. Our contributions include providing a user-programmable security system and a method of mitigating VM context switching latency. Lastly, we believe the virtualization cost is well justified by the benefits attained, and we confirm our findings by benchmarks on *Hvexec* which we have also made open source.

The rest of the paper is organized as follows: Section 2 compares *Hvexec* with related work. Section 3 describes the design issues we faced and our solutions. Section 4 presents benchmarks to evaluate *Hvexec*. Section 5 suggests future direction and concludes the paper.

2 Related Work

In this section, we compare *Hvexec* with several representative works. Our comparison does not imply that other approaches are not suitable for their original usage scenarios. Instead, we intend to show that *Hvexec* improves on prior designs for addressing the requirements in the introduction section.

Directly modifying the kernel is the most direct approach and can achieve the best performance as well as offering excellent assurances. Within the Linux community, SELinux (Security Enhanced Linux) has been in use since the late 2000s [7]. SELinux adds finer granularity to existing access controls and allows the system's administrator be able to specify which users an read, write, append, unlink, move or execute a file and so on [7]. Unlike existing access controls, SELinux can also enforce restrictions on network access and inter-process communication channels [7]. However, kernel extensions requires nontrivial code changes to kernel which has risk of introducing new vulnerabilities [8, 9] and requires recompiling the kernel and elevated permissions for installation which inconveniences users and administrators. Solutions like SELinux typically prescribe a fixed (non-programmable) security model whereas we aim to provide general purpose programmability. Other existing kernel features like ptrace are not suited for sandboxing due to argument race conditions [9]. Shill Shell and Capsicum on FreeBSD provide programmable capability-based interface [11, 10]. However, the Shill Shell relies on FreeBSD's MAC framework which limits the granularity at which it can protect resources. Both do not support transformations on arguments and return values from system

calls.

Alternatively, implementing a sandbox purely in userspace has been explored in prior works, most notably, Google's Native Client (NaCl) [12, 13]. NaCl avoids the performance overhead of dynamic binary translation by statically verifying machine code at compile time [12]. To address the difficulty on architectures with variable length instructions and jump targets not aligned to instruction boundaries binaries, NaCl requires that no instructions or pseudo-instructions overlap a 32-byte boundary. In short, in order to make this type of analysis feasible, NaCl imposes specific constraints on the pattern of machine instructions used [12]. Our approach allows arbitrary sequences of machine code to be executed without any static analysis binary translation.

The last category leverages the widespread deployment of virtualization capable hardware (HVM) to do sandboxing. While full OS Virtualization provides the necessary isolation is not scalable because of the large overhead of running an entire guest operating system. This motivates the need for implementing a more specific light-weight solution. The most similar existing works that we were able to find are Ether, VCall, KVM sandbox and Microsoft Application Guard [6, 4, 3, 14]. Each is either implemented using KVM on Linux or Hyper-v on Windows. To the best of our knowledge, there is currently no implementation for macOS.

3 Design and Implementation

The primary responsibility of *Hvexec* is to isolate any side effects that an arbitrary untrusted sequence of x86 machine code may have on the underlying system. One can simply think of *Hvexec* as two components: the untrusted code which executes inside a VM and the management program running on the host as a normal user process. As described in the introduction, the VM serves as a native execution environment that is isolated from the host at the hardware level. To access any host side resource, the untrusted code in the VM must trigger a VMEXIT condition which turns the execution over to the management program on the host. By interposing on these VMEXIT events, we are able to control what resources the untrusted code can see and use as shown in Figure 1.

We chose to build the prototype on macOS, a popular operating system among consumers and application writers in this era. MacOS has kernel support exposing the underlying x86 VT-x extensions through a system called `Hypervisor.framework` (analogous to KVM on Linux). The rest of this section will describe how *Hvexec* manages memory, interposes on system resource access and exposes a programmable API. It is organized as follows: Section 3.1 initialization Section 3.2 address translation. Section 3.3 syscall handling. Section 3.4 explains the plugin system. Section 3.5 addresses current limitations.

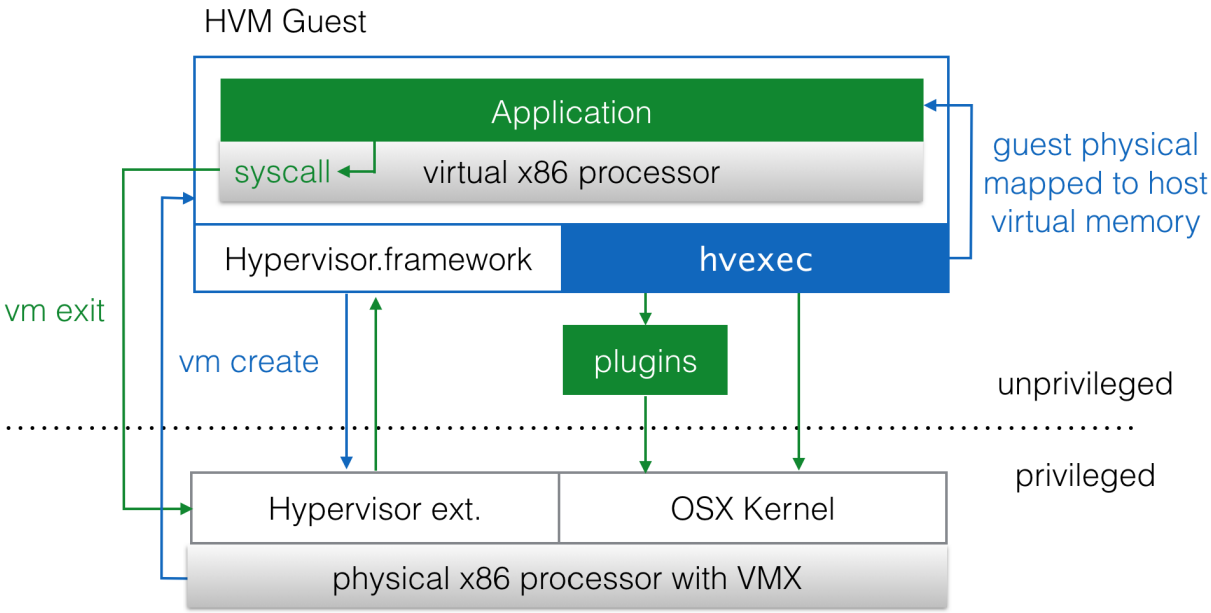

Figure 1: Diagram showing sandbox creation and system call interposition

3.1 Initialization

We designed *Hvexec* to be used as a command-line application. Upon start-up, *Hvexec* needs to create a new VM using the API exposed by `Hypervisor.framework`. `Hypervisor.framework` enforces a one-to-one mapping between VM and host process; however, each VM may be configured with one or more virtual CPUs (VCPU). Each VCPU may be mapped to a host thread. While `Hypervisor.framework` handles some of the setup, the client is expected to do most of the configuration. VM creation involves configuring the register state of VCPUs, the conditions for VMEXITs and the physical memory allocation for the VM. Most of this configuration is done by setting specific fields in an in memory structure called the Virtual Machine Control Structure (VMCS) [15]. The exact definition of each field can be referenced in Intel's official systems programming manual [15].

The VM is capable of supporting 16, 32 and 64 bit execution modes. The 16 bit mode (real mode) is mostly used for legacy applications and firmware and most applications on macOS are compiled for 64 bit execution (IA-32e, also known as long mode). Thus, we only aim to support x86_64 machine code; however, we made *Hvexec* boot the VM in 32 bit mode, in case we want to add 32 bit support in the future, then transition to 64 bit. For x86, 64-bit execution requires virtual memory to be enabled and it is *Hvexec*'s responsibility to setup the initial global descriptor tables and segment selector registers. Lines 33 to 41 of `boot.asm` describe the initial memory layout. To help determine the correct configuration values, we

referenced an open source hypervisor called Xhyve and based our configuration on Xhyve's during the boot process.

3.2 Address Translation

Code in the VM (guest) executes in a separate address space than the host process. The address spaces from guest to host are as follows: guest virtual address (GVA), guest physical address (GPA), host virtual address (HVA) and host physical address (HPA). The guest is allowed to manage its own page tables, but *Hvexec* initializes the guest to use 4 Kbyte size pages. Translation from GVA to GPA is performed by the MMU in the VM. The mapping between GPA and HVA is tracked by means of a list of `struct mem_region` nodes defined in `vm_mem.c`. The function `gpa_to_hva()` in the same file implements the logic to traverse the list and look-up the address mappings. HVA to HPA translation is handled entirely by macOS.

It's important to point out that the address translation functions in `vm_mem.c` are only called during VM initialization or after a VMEXIT has been triggered. GPA to HPA translation is normally handled by the hardware MMU through x86 extended page tables (EPT) managed by `Hypervisor.framework`. With EPT, two page tables are traversed on a guest TLB miss. The first is the guest's page table and the second is the EPT configured on the host by `Hypervisor.framework`. This doubles the cost of a guest TLB miss. To mitigate this cost, Intel added an Address Space Identifier (ASID) to track which entry in the TLB belongs to which VM. This way, a context switch between VMs won't necessarily trigger a TLB flush.

3.3 VMEXIT Handling

Once initialization is complete, the `vcpu_run()` function in `vcpu.c` is called. Within this function, the program enters a run loop which performs the following logic on each iteration. First, `hv_vcpu_run()` is called and this transfers execution to a VCPU. This is a blocking call and will only return when a VMEXIT condition is triggered. This list describes the VMEXIT conditions handled by *Hvexec*.

- `VMX_REASON_IRQ:` The time slice or quantum for the VM is up.
- `VMX_REASON_EXC_NMI:` A non-maskable interrupt (NMI) was triggered. This most likely indicates an error condition since *Hvexec* does not explicitly use NMIs.
- `VMX_REASON_VMCALL:` The code in the VM executed the `vmcall` instruction. This instruction is used to explicitly trigger a VMEXIT. Currently it is used for debugging.
- `VMX_REASON_HLT:` The `hlt` instruction was executed and now the VM should terminate.
- `VMX_REASON_EPT_VIOLATION:` Triggered when the VM references a memory location that has not been paged in. The page fault will be handled by macOS and `Hypervisor.framework`; however, if we

wanted to implement MMIO, this is where it should be done.

- `VMX_REASON_RDMSR,VMX_REASON_WRMSR:` This is for handling reads and writes to model specific registers (MSR). Since we boot the VM in 32 bit mode, transitioning to 64 bit requires setting MSR values.

- `VMX_REASON_MOV_CR:` Paging must be enabled in order to use 64 bit mode. To turn on paging the VM has to assert bit 31 in `cr0` (control register 0).

- `Default:` The `syscall` instruction normally does not trigger a VMEXIT. To force a VMEXIT we register an invalid syscall handler and trigger a triple fault. Since the `syscall` instruction stores the next instruction address in the `rcx` register before jumping to the syscall handler, we can read the instruction at the address of `rcx` minus 2 bytes (size of the `syscall` insturction) to determine if the triple fault was caused by a `syscall`. If not, then this is an error condition. Otherwise, we handle the `syscall`.

In any non-error case, we also increment the value of `rip` (instruction pointer) to the next instruction. The run loop terminates when an unhandled error occurs or when the VM executes a `hlt`.

3.4 Plugin System

When *Hvexec* is invoked on the command-line, the path to a plugin can be passed in as an optional argument. The purpose of plugins is to allow users to implement their own programmable security policies. If this argument is not supplied, *Hvexec* simply functions in pass-through mode and all system calls are forwarded to the host unmodified.

Plugins are macOS dynamic libraries (`*.dylib`) that conform the the interface defined in `hvexec_plugin.h`. The plugin writer is only required to implement the `plugin_init()` function. A `struct plugin_ops` object is initalized by *Hvexec* and passed as an argument to `plugin_init()` as a way for the plugin to hook into *Hvexec* events. Function pointers can be optionally assigned to `plugin_ops` to override default system call behavior.

3.5 Limitations

While some applications ship as single statically linked executable, most applications link against dynamic libraries. It is the responsibility of the loader to map the dynamic libraries into the process address space and fill out the symbol relocation tables and properly setup position independent code. However, the dynamic loader that ships with macOS (`dyld`) is not compatible with *Hvexec* because it depends on concurrency support (`pthread` and `OSSpinLock`). Currently, our implementation is limited to running single threaded applications. We do not support system calls to OS level threading or fork() style multiprocessing and *Hvexec*

only uses a single VCPU. Although *Hvexec* only supports statically linked binaries, this does not preclude users from implementing their own dynamic loader and running that in the VM along with their application.

4 Evaluation

The only question remaining is the performance overhead of our approach. In this section, we compare the performance in terms of execution time between programs running on the host versus the same programs running in a VM under *Hvexec*. Naturally, any code executing in the VM will suffer some performance cost; however, we hope to show that by using HVM, most instructions execute unhindered and the bottleneck is limited to the transition between VM and host. Figure 2 shows a breakdown of execution time between native and *Hvexec*.

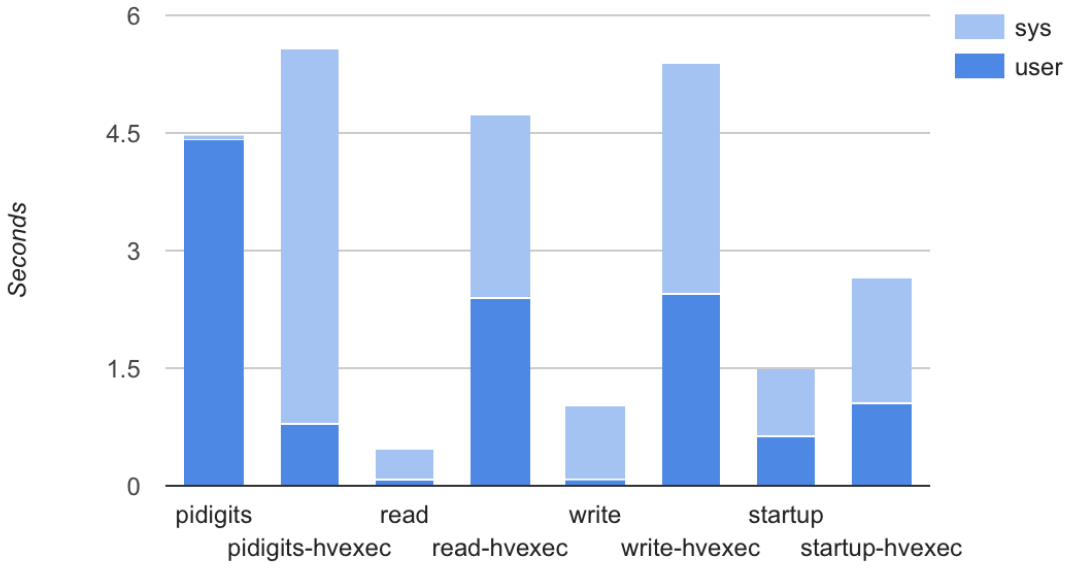

Figure 2: Chart comparing virtualization overhead.

Sys refers to time spent in the kernel and user refers to time running in user-mode. The pidigits benchmark is a program that repeatedly calculates the first eight hundred digits of pi. It contains almost exclusively integer arithmetic and system call transitions are kept to a minimum. The write benchmark opens a temporary file and writes a million bytes of data. Then the read benchmark reads back the contents of that temporary file. Both the read and write benchmarks performs the IO a single byte a time. So unlike the pidigits benchmark, these programs heavily use system calls in a tight loop and spends negligible time doing

arithmetic computations. Lastly, the start-up benchmark compares the cost of program loading and termination on the host versus the VM. The test program used in this benchmark is an empty main function that returns immediately once it is called. In the case of running under *Hvexec*, most of the time will be used for copying the test program and *Hvexec* binaries into memory, booting and tearing down the VM. Running natively on the host avoids the cost of VM start-up as well as the additional overhead of loading the *Hvexec* binary.

Hvexec is overall 1.24 times slower than native execution in the pidigits benchmark. As expected, the fraction of sys time spent is reversed because `Hypervisor.framework` turns execution over to the kernel for VM execution. If we compare the sys execution time of *Hvexec* against native, we see that it is only 1.07 times slower which is aligned with our expectation of HVM performance. The user execution time is also slightly higher because *Hvexec* incurs additional user time for handling the VMEXIT events and initial program load.

Our system performs the worst in the read and write benchmarks because of the high number of VMEXITs triggered by repeatedly issuing system calls. The cost of making a system call under *Hvexec* is the time it takes for the kernel to service the system call plus the time it takes for `Hypervisor.framework` to perform a context switch plus the logic inside of VMEXIT handlers in *Hvexec*. Thus, the proportion of time spent between user and sys is much higher for *Hvexec*.

Start-up time is 1.77 slower for *Hvexec*. This is expected because we are essentially loading two programs (application and *Hvexec*) whereas the native host execution only loads one. A future optimization may be to keep *Hvexec* process alive so we can hot-load application binaries and amortize the cost of start-up.

5 Conclusions

Here we have documented *Hvexec*, a sandbox for safely executing untrusted x86 machine code. We have shown how *Hvexec* can be used to guard against undesirable side effects on underlying system resources and evaluated the performance cost. We hope to have convinced the reader that hardware-assisted virtualization has the potential for achieving a higher level of protection and control for users compared to previous technologies. However, *Hvexec* is simply a starting point. It is easy to imagine many other applications using this type of technology:

- ABI compatibility layer reducing burden on the OS for maintaining binary compatible interfaces.
- Portable binaries across different operating systems and foreign system call emulation.
- Live snapshotting and migration for applications.
- Tools for debugging and tracing.

References

[1] M. K. McKusick, G. V. Neville-Neil and R. N. M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2nd edition, 2014. ISBN-13: 978-0321968975.

[2] T. Garfinkel, B. Pfaff and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. *NDSS, The Internet Society*, 2004.

[3] B. Li, J. Li, T. Wo, C. Hu and L. Zhong. A VMM-based system call interposition framework for program monitoring. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, 2010.

[4] A. Ayer. KVMSandbox: Application-Level Sandboxing with x86 Hardware Virtualization and KVM. Master's Thesis, Brown University, 2012.

[5] N. Amit, A. Gordon, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster and D. Tsafrir. Bare-metal performance for virtual machines with exitless interrupts. *Communications of the ACM, 59*(1). 2015.

[6] A. Dinaburg, P. Royal, M. Sharif and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008.

[7] F. Mayer, D. Caplan, and K. MacMillan. SELinux by example: using security enhanced Linux. *Pearson Education*, 2006.

[8] W. Purczynski and A. P. Moore. Vulnerability note VU#176888, Linux kernel contains race condition via ptrace/procfs/execve. CERT.org, 2002.

[9] A. Cox. CAN-2003-0127, Linux kernel ptrace() flaw lets local users gain root privileges. cve.mitre.org, 2003.

[10] S. Moore, C. Dimoulas, D. King and S. Chong. SHILL: A secure shell scripting language. In *11th USENIX Symposium on Operating Systems Design and Implementation*. 2014.

[11] R. N. Watson, J. Anderson, B. Laurie, K. Kennaway. Capsicum: Practical Capabilities for UNIX. In *USENIX Security Symposium (Vol. 46)*, 2010.

[12] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*. 2009.

[13] J. Ansel, P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff and B. Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *ACM SIGPLAN Notices (Vol. 46)*. 2011.

[14] Introducing Windows Defender Application Guard for Microsoft Edge. https://blogs.windows.com/msedgedev/2016/09/27/application-guard-microsoft-edge/#t7eWrQizF5YCMBiU.97

[15] Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 3 (3A, 3B, 3C & 3D): System Programming Guide.