How soon can we start a hypervisor? Consideration about enabling hypervisor in open-source firmware

Krystian Hebel[1], Piotr Król[2], Michał Żygowski[3], and Łukasz Wcisło[4]

[1] 3mdeb Embedded Systems Consulting, Gdansk, Poland
krystian.hebel@3mdeb.com
[2] 3mdeb Embedded Systems Consulting, Gdansk, Poland
piotr.krol@3mdeb.com
[3] 3mdeb Embedded Systems Consulting, Gdansk, Poland
michal.zygowski@3mdeb.com
[4] 3mdeb Embedded Systems Consulting, Gdansk, Poland
lukasz.wcislo@3mdeb.com

Abstract

Until now SPI flash memories were not considered to be a storage for a hypervisor, because they were relatively too small. We've embedded Bareflank-based hypervisor into SPI flash to be launched directly from coreboot and load SeaBIOS, also embedded inside SPI flash. For this purpose, we had to change architecture from 32-bit used by coreboot to 64-bit used by a hypervisor, and then get back to 32-bit to load SeaBIOS as a payload. This is a compact solution for multiple purposes using Virtual Machines that provides separation, stability, and security. Fact, that the hypervisor is embedded in the SPI means, that simple disk removal doesn't affect it. In this paper, we will show how we've done it and what are the possible extensions and usages of our concept.

1 Introduction

Our ultimate goal was to create firmware that can start multiple applications in isolated virtual environments directly from SPI flash. To achieve that we used two open and flexible frameworks Bareflank and coreboot combining it into solution.

The Bareflank Hypervisor [1] is an open-source hypervisor Software Development Kit (SDK), led by Assured Information Security, Inc. (AIS), that provides a set of APIs needed to rapidly prototype and create new hypervisors. To ease development, Bareflank is written in C/C++, and includes support for C++ exceptions, JSON, the GSL and the C++ Standard Template Library (STL).

coreboot [2] is an extended firmware developing framework that delivers a lightning fast boot experience on modern computers and embedded systems. As an open-source project it provides auditability and maximum control over the technology. In our solution coreboot brings up the platform, and handles hypervisor as a payload.

2 Hypervisors

Hypervisors are divided into two types:

- type 1 - native or bare metal hypervisors
- type 2 - hosted hypervisors

The division has been presented on Figure 1.

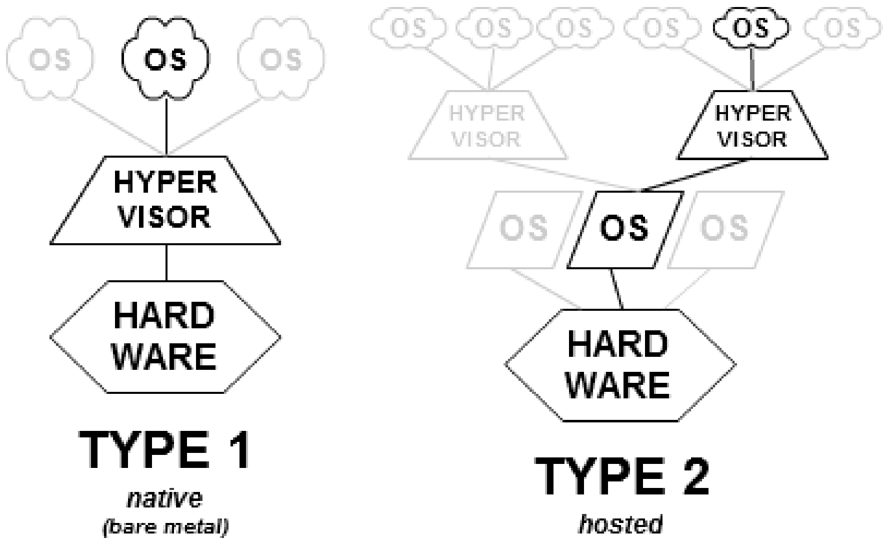

Figure 1: Hypervisor types [3].

Type 1 hypervisors run directly on hardware and are responsible for controlling the hardware and managing the guest operating systems. Examples of modern type 1 equivalent hypervisors are: Xen [4], Microsoft Hyper-V [5], VMware ESX/ESXi [6]. Bareflank-based hypervisors also belong to this group.

Type 2 hypervisors run from a conventional operating system, thus they are called hosted. The guest operating systems are just the processes of the host operating system. Example implementations of type 2 hypervisors are: QEMU [7], Oracle VirtualBox [8], VMware Player [9].

There are also other examples of virtualization software that can be categorized as both types of hypervisors. They include Linux's KVM (Kernel-based Virtual Machine) [10] and FreeBSD's bhyve [11], which, as kernel modules, can convert the OS they are running on into a type 1 hypervisor. On the other hand Linux and FreeBSD are still operating systems which host hypervisor software and have their own applications which compete for resources. In such case they can also be called type 2 hypervisors.

There is also an informal division into type 0 hypervisors. The type 0 is an hardware hypervisor implemented by firmware [12]. Typically those hypervisors have limited feature set and each guest has a logical dedicated hardware.

3 Terminology

Modern processors have evolved to an extent that they can accelerate the operation of hypervisors with the help of virtualization extensions embedded in the silicon. Virtualization is the application of the layering principle through enforced modularity, whereby the exposed virtual resource is identical to the underlying physical resource being virtualized [13].

A virtual machine (VM) is an abstraction of a complete compute environment through the combined virtualization of the processor, memory, and I/O components of a computer. In this document VM may sometimes be used to describe CPU mode in which virtual machine operates, in contrast to privileged mode reserved for hypervisor.

The hypervisor is a specialized piece of system software that manages and runs virtual machines.

The virtual machine monitor (VMM) refers to the portion of the hypervisor that focuses on the CPU and memory virtualization. Similarly to VM, we may use VMM to describe privileged mode of operation.

The terminology presented above may not always be accurate or may have different meanings in terms of other hypervisor implementations. It is quite common to see different definitions across projects and specifications.

Virtual-machine extensions (VMX) is a hardware implementation of virtualization in newer Intel processors. It is marketed under the name VT-x. Virtual machine control structure (VMCS) is a VMX structure describing transitions between VM and VMM, it also holds virtual CPU state. VMX and VMCS are described in detail in Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 [16].

4 Embedded Hypervisors

An embedded hypervisor is primarily designed for embedded systems and generally provides the same functionality as a type 2 hypervisor but puts emphasis mostly on security and performance. They are also less customizable at runtime and have their set of isolation and memory rigidly hardcoded.

An embedded hypervisor is most often a type 1 hypervisor, which means it runs directly on the host's hardware to control the hardware and to manage guest operating systems. For this reason, they are sometimes called bare metal. They are able to provide secure encapsulation for any subsystem defined by the developer so that a compromised subsystem cannot interfere with other subsystems. For example, an encryption subsystem needs to be strongly shielded from attack to prevent leaking the information the encryption is supposed to protect. As the embedded hypervisor can encapsulate a subsystem in a VM, it can then enforce required security policies for communication to and from that subsystem.

Another reason why to use an embedded hypervisor is that a VM may be restarted without platform initialization (unlike watchdog-generated reset), which can reduce time required for recovering the system back to useful state (because no memory training, PCI bus scan etc. is required). It is possible to restart only one VM without restarting all of them. This helps with running broad spectrum of software with different stability requirements on one platform. For example, media player can be stopped with almost no impact while even a short downtime in safety-critical systems like smoke detection, collision avoidance system in automotive or terrain awareness and warning system (TAWS) in aviation can result in lost of life. Thanks to secure encapsulation they can run on the same hardware, without impacting each other.

The main purpose of developing the hypervisor in the embedded system firmware is to increase security and reliability with restriction to load code entirely from SPI flash only. Such systems may be used in safety and life-critical applications. Any kernel panic or sustain could be lethal, so hypervisor is necessary to guard the operating system on a very low level.

5 Bareflank initialization flow

The Bareflank payload can be divided into two parts:

- The VMM code delivered as a C header file with bytecode as a result of Bareflank SDK build.

- bfdriver (Bareflank driver) - minimal C code providing necessary hypervisor hooks and code for VMM launching.

VMM part is compiled separately, from Bareflank source code[1]. We prepared Bareflank build environment in form of Docker container [18]. Instructions for building are available there as well. Although instructions mention building for UEFI, the VMM part of hypervisor can be used with our implementation for coreboot. File required for the next step is still available in the same directory as UEFI application, but its name is *vmm.h*. It contains hypervisor in ELF format, saved as table of bytes ready to be used by C programs. Original ELF file has size of about 1.2 MB[2], header file is over 6 times bigger due to its text format. This size gets back to its previous value after compiling. A table of bytes is used instead of loading binary file in order to be independent from system or firmware-specific APIs for loading files and accessing media devices.

The second part, bfdriver, is compiled as a part of coreboot build process. It assumes that *vmm.h* is located in any of the included paths. We also needed to include some header files from Bareflank, for now they are copied into the coreboot tree directly. This makes Bareflank version used to build VMM dependent on copied headers , but we plan to address this problem.

A simplified flow of Bareflank initialization is shown on Figure 2. It does not go into details of specific platforms. Second row of VMM block contains actual uses of VMX instructions. While that code is compiled as a VMM module, not all of it actually runs at VMM level. Before *vmxon* CPU runs in non-VMX mode, code between *vmxon* and *vmlaunch* runs with VMM privileges, and everything after that is a VM. Since *vmlaunch* VM exits may occur; this fact is used to implement required post-launch initialization[3]. Note that a large part of the code must be run on each core.

6 Solution

We've embedded Bareflank-based hypervisor into SPI flash to be launched by coreboot as a payload. With the help of SeaBIOS [17] also embedded inside SPI flash we load the guest operating systems. This is a compact payload for multiple purposes and in our solution it is using virtual machines deployed with Bareflank-based hypervisor. We outline challenges related to embedded hypervisor integration in open-source firmware and possible ways to overcome those.

At some point Bareflank executable file must be started. For other platforms it uses either standard C libraries with OS-specific API for type 2 or UEFI Boot Services for type 1 hypervisors. None of these are available in coreboot, so we had to devise another option.

We have integrated the Bareflank hypervisor with coreboot thanks to the libpayload, which is a static library containing common and useful functions for coreboot payloads. Our hypervisor builds result in a C header file with an array of hypervisor bytecode in ELF format. We take the header file and load it as a normal program via bfdriver linked with libpayload. Structure and transitions in and out of resulting payload is presented on Figure 3.

[1]Code in Bareflank changes frequently. Some time ago we decided to freeze on one version, right after extended APIs were merged into main repository [14].

[2]This size depends on complexity of VMM, it was given for hypervisor built for integration tests, using instructions provided with Docker container. VMM will usually be bigger for feature-rich hypervisors.

[3]Some initialization is required after the hypervisor has started. For example, any memory mapped resources such as ACPI or VT-d need to be initialized using the VMM's CR3, and not the hosts [15]. For type 1 hypervisors they are usually the same, but Bareflank supports also type 2 hypervisors.

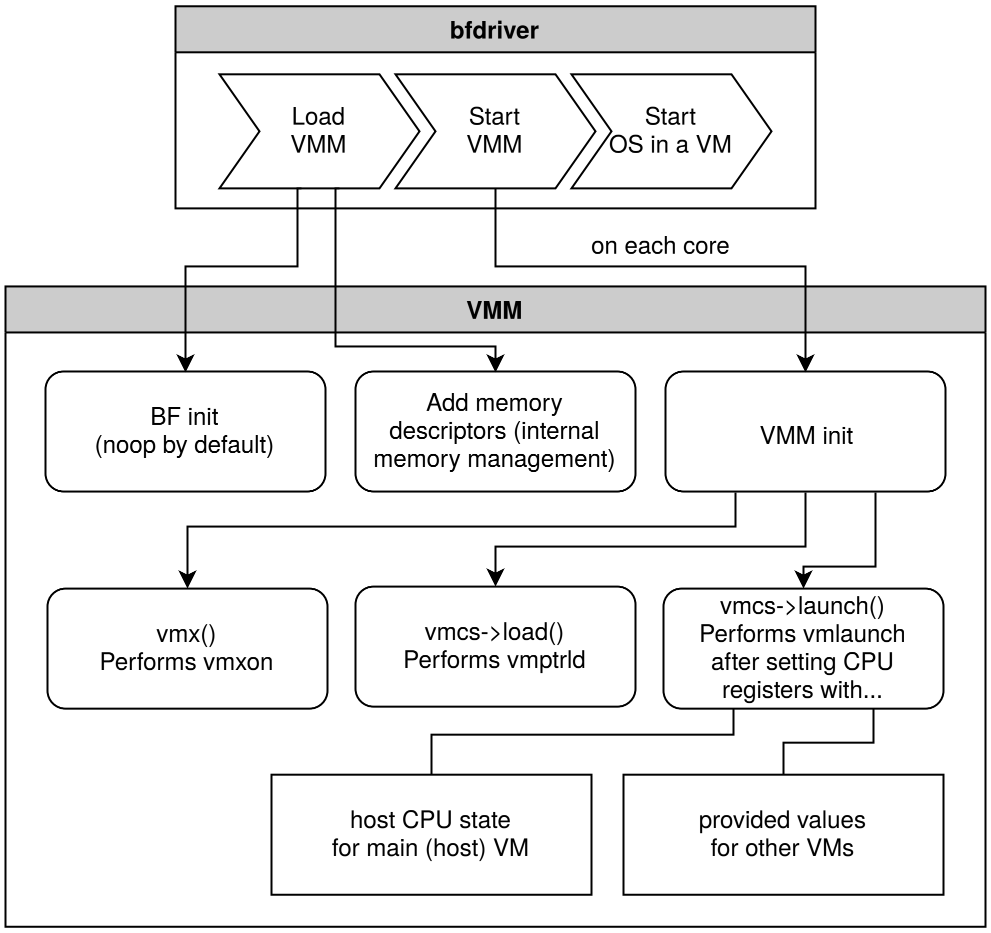
bfdriver
Load VMM
Start VMM
Start OS in a VM
on each core
VMM
BF init (noop by default)
Add memory descriptors (internal memory management)
VMM init
vmx() Performs vmxon
vmcs->load() Performs vmptrld
vmcs->launch() Performs vmlaunch after setting CPU registers with...
host CPU state for main (host) VM
provided values for other VMs

Figure 2: High-level overview of generic Bareflank initialization flow.

The first issue that had to be solved was that both Bareflank and SeaBIOS need to be built, and Bareflank becomes the main payload – the one that is loaded automatically by the last stage of coreboot (ramstage). This was resolved with small changes to the coreboot build system and won't be described in detail. More important part of this issue was the size of SPI memory. On the board we were working (MinnowBoard Turbot Quad-Core) that size is only 8 MB [19], but only half of it is available for BIOS section by default (the rest is reserved for ME, GbE and flash descriptor). Moreover, continuous free space is limited because BIOS section contains code that must be put at specific offsets. It can barely fit hypervisor, which is around 3.5 MB (uncompressed) before adding any custom handlers. We had to be careful and we left a little place to strip it eventually to have a way out, if during further development it would turn out that we have no space left. Payload compression is a must-have. Still, SPI contains SeaBIOS as well.

The second problem was that SeaBIOS obtains memory map from coreboot tables, which are *not* modified by memory allocation functions provided by libpayload. Payloads typically have limited lifetime, so the memory used by them is considered to be free as soon as that payload exits or starts another boot stage. As a result, SeaBIOS was free to write over the VMM memory, and even though it can be blocked by hypervisor, reporting this memory range as reserved is more elegant solution. It also makes it easier to deal with discrepancy between amount of memory available to VM and physical memory.

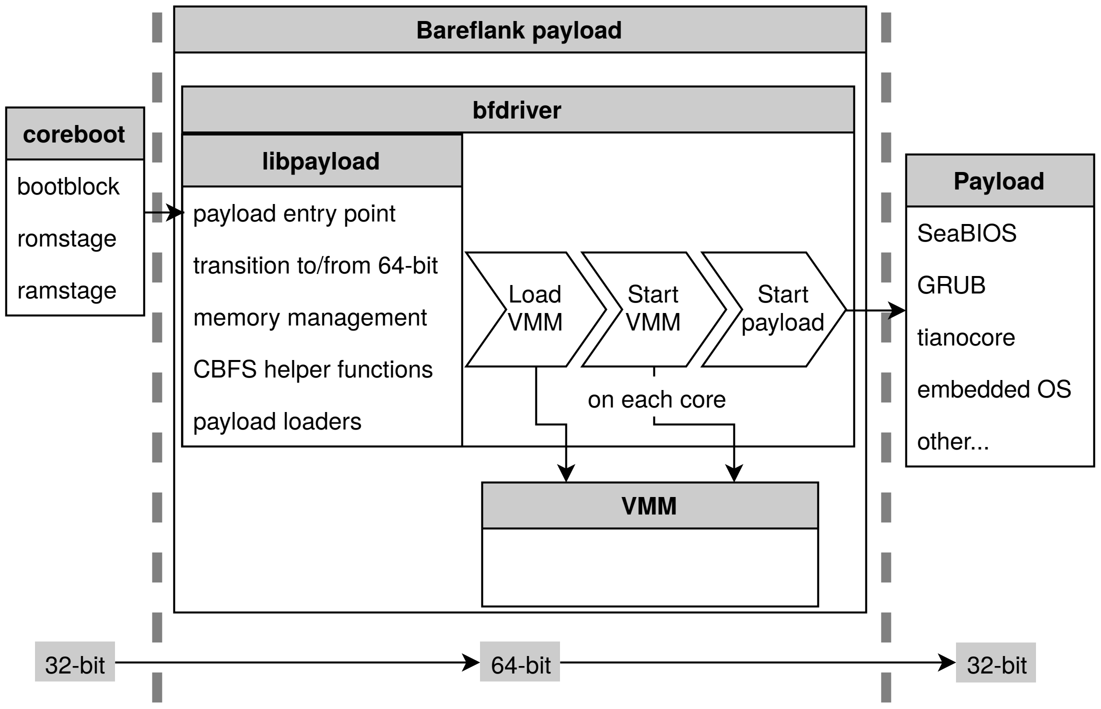

Figure 3: Structure of Bareflank payload.

Next problem is that coreboot works with 32-bit, Bareflank with 64-bit (while this is not a prerequisite for starting VMX, it is used in Bareflank because it simplifies its implementation), and SeaBIOS again with 32-bit architecture. We have achieved that by extending a default libpayload config for x86 32-bit architecture and fixing parts of code which were not ready for working in 64-bit environment. Soon after the Bareflank is loaded we switch to 64-bit mode and then execute our hypervisor, which in turn performs transition back to 32-bit mode before starting final payload.

Every core needs to set up VMX mode separately for multiple reasons, main being that each virtual CPU has its own state, which requires (at least) one VMCS per logical processor (which is either one core or one thread in systems with Intel Hyper-Threading)[4]. Code for executing code on all cores/threads is not part of libpayload, we had to implement our own, including transition to 64-bit mode.

[4]Apart from mentioned ability to save state of multiple vCPUs, multiple VMCSs can be used for asymmetric systems, where each core can work with different VMMs. While asymmetry can be achieved in code, checking for vCPU number on each exit would introduce unnecessary delays. Another reason is that each vCPU can have its own set of instructions and events resulting in VM exit. This is useful if some VMs require more access to physical hardware than others. The most permissive (and dangerous) option is to not start VMX on all processors, so some of them would have full access to the system – including ability to check for presence of other cores, so they must not be used by software running on unrestricted cores, as there would be no hardware restrictions of any kind. This core could also access memory assigned to VMM and communicate with all devices.

As a last step, to launch target applications or operating systems we start a virtual machine[5] which starts executing SeaBIOS as a bootloader and then loads an operating system. Instead of a heavy operating system it also might be a lightweight real-time operating system (RTOS) which may even fit into SPI flash entirely. The example architecture and execution model has been shown on Figure 4.

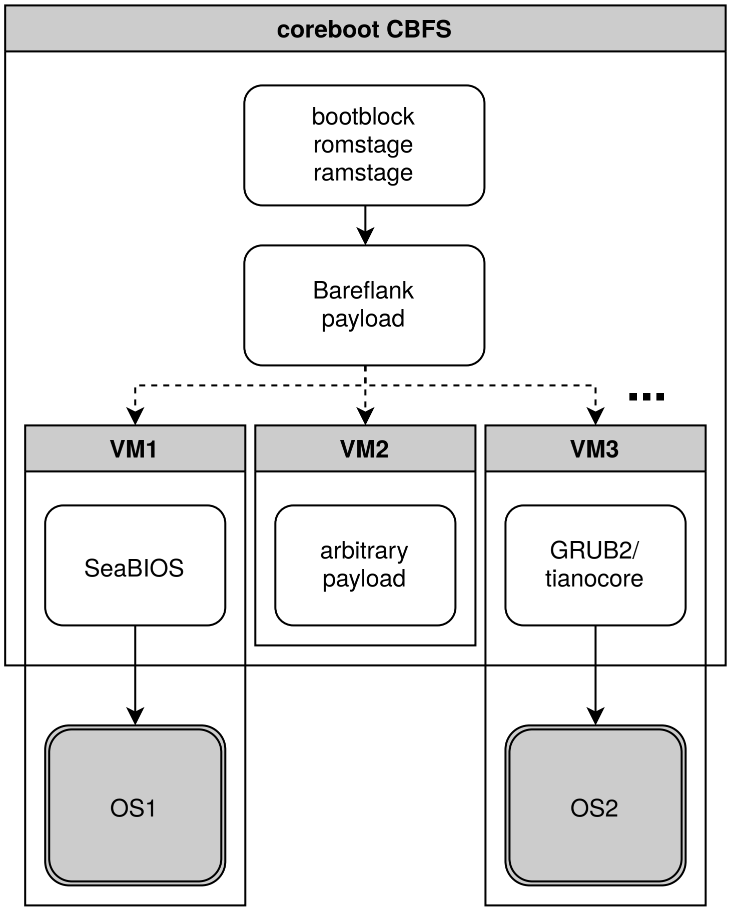

Figure 4: Example Bareflank hypervisor architecture.

6.1 Memory reservation

In the proposed architecture hypervisor wrapped with libpayload is launched. The memory allocation for the program is done by the libpayload's *malloc*. In order to pass the information to SeaBIOS launched in VM, we have marked the memory allocated for hypervisor as reserved. As a result SeaBIOS will create memory tables that mark hypervisor code as reserved. In such way the virtual machine knows which parts of RAM it should not access.

[5] Technically, the last part of Bareflank driver (see section 5) is executed in a VM already. This helps with parts of initialization that need to be performed after VMX is started. While this is important for developers, end users may safely assume that VM starts from entry point of payload.

Otherwise we would have to dynamically generate and update extended page tables (EPT)[6] at runtime, as well as implement our own memory management functions for allocating pages for virtual machines. Those functions would have to omit hypervisor memory and keep its own track of memory already in use. As sum of available memory sizes reported to all concurrently running VMs is usually larger than physical memory available in this case, some kind of swap memory would be required. While this is possible, it would have impact on both hypervisor size and runtime performance. It would be very hard, if not impossible, to use such approach with RTOS.

6.2 Virtualization enabling

In order to use virtualization features our hypervisor had to have a bit of multiprocessing (MP) code to be able to turn virtual machine extensions (VMX) on. The VMX has to be turned on for each core separately before any virtual machine on that core starts. The *vmxon* instruction serves the purpose.

One way to look at multi processor systems is that they are separate machines with shared memory. For implementing MP support we used APIC interrupts and a trampoline code that transits from 16-bit mode all the way towards 64-bit mode. Because it is used only once, during startup, there is no need to run it on multiple cores simultaneously, so all APs are started in turn.

6.3 Target virtual machines

In the Figure 4 we have presented example usages and target virtual machines that could be used with Bareflank. One can simply launch another payload entirely from CBFS to be running as virtual machine or run a payload/bootloader of the operating system. Examples of such payloads or bootloaders are:

- SeaBIOS [17] - open-source legacy BIOS implementation for booting conventional OS in legacy mode.
- GRUB [20] - Grand Unified Bootloader, a GNU project designed mainly to load he Linux kernel with initial ramdisk and pass the kernel command line parameters.
- tianocore [21] - open-source minimal Unified Extensible Firmware Interface (UEFI) implementation. Tianocore payload is a special coreboot payload built with EDK2 and coreboot packages. Can be used to load UEFI aware operating systems.

Summary

Apparently, there is a way to fit a hypervisor into an SPI flash, yet this solution has weak points. The biggest is the lack of memory. Our Bareflank hypervisor has very limited features, it doesn't handle multiprocessing like other hypervisors do. But it works. Similar concepts are simultaneously developed in other use cases: automotive, security and military, whenever any issue may be life or security critical.

[6]Extended page tables provide mapping from guest physical to host virtual address space. They also enable two new classes of exit reasons: EPT misconfiguration (tables are not valid, might be used for swapping memory) and EPT violation (VM tries to access memory that it is not supposed to, e.g. write to read-only memory, executing code from page that is not marked as executable). This is described in detail in SDM [16].

There are many possible future improvements. Firstly, Bareflank build system could be included in coreboot build system, so there would be no need for using both in turn. This would also improve header dependency issues. Another option could be to include whole ELF file in CBFS, instead of providing VMM part of hypervisor in the form of C include file. This way, when only VMM part is changed (i.e. behaviour of virtualized instructions or events, not the partitioning of resources) there is no need to rebuild whole coreboot image.

As for partitioning, it should be possible to provide some type of configuration possible after build, e.g. by reading a configuration file from CBFS. Also memory maps for next payloads can be reworked so reported total memory would be smaller, instead of marking a large part of it as reserved. Amount of memory can also be read from MSRs, they should also be emulated accordingly, but the exact details are dependent on CPU vendor and model.

Old code base (from beginning of 2019) was used. Developers of Bareflank have fixed bugs and changed API since.

Bareflank supports only Intel VMX. Support for other technologies (e.g. AMD-V, also named SVM) can provide additional value to this SDK. It seems that developers had this in mind when they were designing infrastructure for this repository, as all of the VMX-specific implementations are kept separate from generic ones.

There is no easy way to tell where bfdriver for coreboot should be located. It can be a part of coreboot, in which way it would be better integrated with libpayload. On the other hand, because of heavy use of Bareflank header files, it can be included in Bareflank repository. Best place for this driver is yet to be decided.

Trademarks

AMD and AMD-V are trademarks of Advanced Micro Devices, Inc.

Intel is a registered trademark of Intel Corporation.

References

[1] https://github.com/Bareflank/hypervisor

[2] https://www.coreboot.org/

[3] https://en.wikipedia.org/wiki/Hypervisor#/media/File:Hyperviseur.png

[4] https://xenproject.org/

[5] https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview

[6] https://www.vmware.com/products/esxi-and-esx.html

[7] https://www.qemu.org/

[8] https://www.virtualbox.org/

[9] https://www.vmware.com/products/workstation-player.html

[10] https://www.linux-kvm.org/page/Main_Page

[11] http://bhyve.org/

[12] https://zuhaib-shaikh.neocities.org/downloads/os/Lecture_Slide-Virtual_Machines_.pdf

[13] Edouard Bugnion, Jason Nieh, Dan Tsafrir, Synthesis Lectures on Computer Architecture, Hardware and Software Support for Virtualization, 2017

[14] https://github.com/Bareflank/hypervisor/commit/ba613e2c687f7042bac6886858cf6da3132a61d6

[15] https://github.com/Bareflank/hypervisor/blob/ba613e2c687f7042bac6886858cf6da3132a61d6/bfvmm/src/hve/arch/intel_x64/exit_handler.cpp#L789

[16] https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-3a-3b-3c-and-3d-system-programming-guide
[17] https://www.coreboot.org/SeaBIOS
[18] https://github.com/3mdeb/bareflank-docker
[19] https://minnowboard.org/minnowboard-turbot/technical-specs
[20] https://www.gnu.org/software/grub/
[21] https://www.tianocore.org/