AsiaBSDCon 2019
Proceedings

March 21-24, 2019

Tokyo, Japan

Copyright © 2019 BSD Research. All rights reserved.
Unauthorized republication is prohibited.

Published in Japan, March 2019

INDEX

Adventure in DRMland
Or how to write a FreeBSD ARM64 DRM Driver

Emmanuel Vadot, manu@freebsd.org

Abstract

DRM (Direct Rendering Manager) is today standard for applications like a display server, to talk the the graphical hardware present on a computer or System On a Chip (SoC). It consist of a kernel side API and a userland part (via IOCTLs) that application can use to talk the GPU or configure the display modes (resolution, refresh rates etc ...).

While support for X86 devices (intel or amd) are now correct on FreeBSD, arm and arm64 hardware support is still lacking. The only DRM driver is for Tegra based SoC, other hardware either have basic framebuffer support (like the RaspberryPi family) or will require the bootloader to have framebuffer support and will use it via EFI framebuffer.

While framebuffer might be enough for some use, having a DRM driver brings a lots of possibility, 2D acceleration, changing resolution, hotpluging another monitor etc ... And it is also mandatory if we want to support the 3D chip or the Video decoder usualy present on arm/arm64 SoCs.

In this paper the author will describe the DRM subsystem and the anatomy of a modern DRM driver, based on his work on the Allwinner Display Engine 2 present in many SoC of this semiconductor company.

1. Overview

1.1. DRM/KMS

DRM is an API first introduced to support 3D GPU in the Unix world. It only focused on GPU (command execution, textures etc ...) and not on the mode setting part. A userland program (most likely X) used to do the mode setting by talking to the hardware directly to setup VGA/HDMI and all the rendering pipeline. This required X to run as root and also cause problems if multiple program wanted to configure the hardware.
To solve this a second API was later introduced called KMS (Kernel Mode Setting). The userspace didn't need to talk to the hardware anymore and will only need to call a few IOCTLs to configure the display. A few iteration of the API added more and more concept like framebuffers, planes, encoder etc ...
This paper do not describe how to create a driver for a GPU but how to create a KMS driver. GPU on arm and arm64 are discrete ones and require a separate driver.

1.2. FreeBSD ARM{,64} Video Support

Support for video (in any possible form) on arm and arm64 on FreeBSD isn't available for a lot of platforms. The first driver was for the framebuffer on Raspberry Pi (r239922[i] by gonzo@ in August 2012), the bootloader/firmware on Rpi setup a display at startup and expose a framebuffer that the OS can use. To my knowledge it isn't possible to change the resolution or to setup the display after the OS is booted.
The second one was for the first generation of Allwinner Display Engine and supported only HDMI (r296064[ii] by jmcneill@ in February 2016). It worked by using modifed DTS files (file used to described the hardware on arm) and was later broken when we fully switched to the Linux DTS files.
In July of 2018 I added efifb support to ARM64 (r336520[iii]). As some bootloader supports video interface and EFI it allows us to have a working display with a simple driver. It has some limitation though, you cannot change the resolution or hotplug a monitor after that the OS is booted.

1.3. Chosen Hardware

To develop my first DRM driver I choose the Allwinner A64 SoC for multiple reasons :

1. This is a SoC that I know very well.
2. A lot of interesting hardware are available (or will be) like the PineBook and Pinetab ([iv])
3. Documentation, even if it isn't very explicit, exists. The only part not documented by Allwinner is the HDMI Transmitter but it is documented in NXP IMX.6 user manual.

2. KMS API and Object

2.1. Framebuffers and GEM Objects

2.1.1. Overview

Framebuffers are memory objects that holds the pixels needed to be rendered (or scanout) to the screen. They are composed of properties like width, height, pixels format etc ... and between one and four GEM objects.
GEM Objects are the memory objects holding the pixel data, this is the hardest part of DRM (atleast for me) as it is directly tied to the VM subsystem.

2.1.2. Implementation

For embedded devices where you don't have graphic memory, one should use the gem_cma implementation (CMA: Contigous Memory Allocator).
The problem is that this part is gplv2 only. A BSD implementation based on NVIDIA Tegra DRM driver by mmel@ is currently being written by the author.

2.2. Planes

2.2.1. Overview

Planes object are backed by drm_framebuffer objects and are tied to a CRTC (see 2.3).
Multiple types of plane exists :

1. Primary plane is used for the main framebuffer so it will likely contain the pixel for your hole desktop environment. There can be only one per CRTC.
2. Cursor plane is used for mouse cursor. Some hardware can do composition directly (blending and mixing multiple buffers) this avoid doing that in software and waste CPU cycle. Cursor plane are usually small, 64x64 pixels is a common size.
3. Overlay planes are just generic plane that can be used for anything. One common application of them is for video, the video player will have his own plane where it can render data directly in it and the compositing hardware will handle the blending/mixing part.

In the Allwinner DE2 there is two units (called mixer) that handle the planes/overlays. One have three planes that support RGB format and one plane that support YUV format (for Video purpose) while the other one only have one RGB plane and one YUV plane.

2.2.2. Implementation

A drm driver can register plane using the *drm_universal_plane_init* function.

```
int drm_universal_plane_init(struct drm_device *dev,
                             struct drm_plane *plane,
                             uint32_t possible_crtcs,
                             const struct drm_plane_funcs *funcs,
                             const uint32_t *formats,
                             unsigned int format_count,
                             const uint64_t *format_modifiers,
                             enum drm_plane_type type,
                             const char *name, ...);
```

The helper funcs are fully provided by the KMS framework and so a driver can simply use the default one.

```
static const struct drm_plane_funcs plane_funcs = {
  .atomic_destroy_state = drm_atomic_helper_plane_destroy_state,
 .atomic_duplicate_state = drm_atomic_helper_plane_duplicate_state,
 .destroy = drm_plane_cleanup,
 .disable_plane = drm_atomic_helper_disable_plane,
 .reset = drm_atomic_helper_plane_reset,
 .update_plane = drm_atomic_helper_update_plane,
};
```

What a driver only need to implements are the helper funcs for atomic mode setting.

```
int plane_atomic_check(struct drm_plane *plane,
                       struct drm_plane_state *state)
void plane_atomic_disable(struct drm_plane *plane,
                          struct drm_plane_state *old_state)
void plane_atomic_update(struct drm_plane *plane,
                         struct drm_plane_state *old_state)

static struct drm_plane_helper_funcs plane_helper_funcs = {
        .atomic_check           = plane_atomic_check,
        .atomic_disable         = plane_atomic_disable,
        .atomic_update          = plane_atomic_update,
};

drm_plane_helper_add(struct drm_plane *plane, &plane_helper_funcs);
```

The atomic_check function will return 0 if the plane can be drawn by the hardware. Most of the time just calling *drm_atomic_helper_check_plane_state* is sufficient.
The atomic_disable function will disable the plane from being rendered.
The atomic_update function will update all the plane information (address, format, size etc ...) in the hardware.

2.3. CRTCs

2.3.1. Overview

CRTC stand for Cathode Ray Tube Controller, this is an historical name and the KMS crtc object don't have anything to do with CRT monitor.
CRTC take the contents of the framebuffers and planes and output the final image on a physical bus. This can be an external bus (some RGB pins to drive a lcd panel for example) or an internal bus that goes into an HDMI or VGA encoder (see 2.4).

In Allwinner SoCs you have again two different units (called TCON), one can output pixel directly in RGB format, MIPI-DSI and LVDS while the other is directly tied to the HDMI transmitter. Both can take their inputs from either of the two mixers but the default configuration is that the mixer0 (The one with 4 planes) outputs to the TCON0 and mixer1 outputs to TCON1.

2.3.2. Implementation

The easiest way to register a crtc in the subsystem is with the function *drm_crtc_init_with_planes*.

```
int drm_crtc_init_with_planes(struct drm_device *dev,
                              struct drm_crtc *crtc,
                              struct drm_plane *primary,
                              struct drm_plane *cursor,
                              const struct drm_crtc_funcs *funcs,
                              const char *name, ...);
static const struct drm_crtc_funcs crtc_funcs = {
 .atomic_destroy_state  = drm_atomic_helper_crtc_destroy_state,
 .atomic_duplicate_state = drm_atomic_helper_crtc_duplicate_state,
 .destroy = drm_crtc_cleanup,
 .page_flip = drm_atomic_helper_page_flip,
 .reset = drm_atomic_helper_crtc_reset,
 .set_config = drm_atomic_helper_set_config,
 .enable_vblank = crtc_enable_vblank,
 .disable_vblank = crtc_disable_vblank,
};
```

In the crtc_funcs only two function for enabling/disabling vblank interrupt need to be implemented, for the others the helpers functions are enough.
Some helper functions for atomic mode setting are also needed :

```
static const struct drm_crtc_helper_funcs crtc_helper_funcs = {
        .atomic_check           = crtc_atomic_check,
        .atomic_begin           = crtc_atomic_begin,
        .atomic_flush           = crtc_atomic_flush,
        .atomic_enable          = crtc_atomic_enable,
        .atomic_disable         = crtc_atomic_disable,
        .mode_set_nofb          = crtc_mode_set_nofb,
};
drm_crtc_helper_add(crtc, &crtc_helper_funcs);
```

All of those functions need to be implemented but even if my current implementation seems to work I don't fully understand what they are really supposed to do. They deal with vblank and events and I am really not familiar with them enough.

2.4. Encoders

2.4.1. Overview

drm_encoder simply convert one pixel data bus format to another one. For example they can convert one internal format (such as between TCON1 and the HDMI transmitter) to TMDS, the signal format used in HDMI transmission.

2.4.2. Implementation

Only one helper function needs to be implemented for encoder : the mode_set one.
This is used to set the clock rate of the encoder at the same rate at the pixel clock for example.

Then using drm_encoder_helper_add and drm_encoder_init on can register the encoder in the DRM subsystem.

```
drm_encoder_helper_add(&sc->encoder,&encoder_helper_funcs);
sc->encoder.possible_crtcs = drm_crtc_mask(crtc);
drm_encoder_init(drm, &sc->encoder, &encoder_funcs,
DRM_MODE_ENCODER_TMDS, NULL);
```

2.5. Bridges/Connectors

2.5.1. Overview

drm_connector simply represent a physical connector on the card or single board computer.
drm_bridge sits between an encoder and a connector. They are used to enable/disable the display and configuring the display mode (resolution and timing).

2.5.2. Implementation

For drm_connector only one function need to be implemented : connector_detect. This isn't possible for every connector type but for HDMI it is.

```
static enum drm_connector_status
connector_detect(struct drm_connector *connector, bool force);

static const struct drm_connector_funcs dw_hdmi_connector_funcs = {
 .fill_modes = drm_helper_probe_single_connector_modes,
 .detect = connector_detect,
 .destroy = drm_connector_cleanup,
 .reset = drm_atomic_helper_connector_reset,
 .atomic_duplicate_state = drm_atomic_helper_connector_duplicate_state,
 .atomic_destroy_state = drm_atomic_helper_connector_destroy_state,
};
```

One helper function is also needed, the get_modes one, it is called for quering EDID from the connected monitor.

```
static const struct drm_connector_helper_funcs
connector_helper_funcs = {
                .get_modes = connector_get_modes,
};
```

For the bridges the following functions need to be implemented :

```
static int
bridge_attach(struct drm_bridge *bridge);
static enum drm_mode_status
bridge_mode_valid(struct drm_bridge *bridge, const struct
drm_display_mode *mode);
static void
bridge_mode_set(struct drm_bridge *bridge,
  struct drm_display_mode *orig_mode,
  struct drm_display_mode *mode);
static void
bridge_disable(struct drm_bridge *bridge);
static void
bridge_enable(struct drm_bridge *bridge);

static const struct drm_bridge_funcs bridge_funcs = {
                .attach = bridge_attach,
                .enable = bridge_enable,
                .disable = bridge_disable,
                .mode_set = bridge_mode_set,
                .mode_valid = bridge_mode_valid,
};
```

bridge_attach needs to init and attach the connector.
bridge_mode_valid need to filter the mode
bridge_mode_set just need to copy the desired mode that will be used in the enable function.

3. Current status and Future work

3.1. Current status

As of 20180225, my current implementation support the mixer1 and tcon1 IP block so only HDMI output is currently possible. There is still problem in the HDMI driver to do a full bring-up, it doesn't work yet if u-boot is configured without video support, this is probably just a few registers that aren't setup correctly.

3.2. Future work

Fixing all the bugs and testing different monitors (with differents supported resolutions) is my main priority.
Next I will finish the BSD implementation of the CMA function.
Adding support for other Allwinner SoC (such as the arm32 H3 and arm64 H5) is also planned in the near future.
LIMA (The reversed-engineered MALI driver) will be very interesting to have in FreeBSD.

i https://svnweb.freebsd.org/base?view=revision&revision=239922
ii https://svnweb.freebsd.org/base?view=revision&revision=296064
iii https://svnweb.freebsd.org/base?view=revision&revision=336520
iv https://www.pine64.org/

Removing ROP Gadgets from OpenBSD

Todd Mortimer
mortimer@openbsd.org

Abstract

Return Oriented Programming (ROP) is a common exploitation technique that reuses existing code fragments (gadgets) to construct shellcode in a compromised program. Recent changes in OpenBSD's compiler have started to reduce the number of gadgets in x86 and arm64 binaries, with the aim of making ROP exploitation more difficult or impossible. This paper will cover how ROP gadgets emerge from legitimate code, how OpenBSD's compiler removes these gadgets, and the effects on performance, code size, and ROP tool capabilities. We find that it is possible to meaningfully reduce the number of ROP gadgets in programs, and to effectively hinder ROP tool capabilities.

1 Background

Return oriented programming (ROP) [5] is an exploitation technique that uses fragments of existing programs in unintended ways to effect control over a compromised process. In contrast to traditional shellcode injection, ROP attacks inject a series of return addresses - a ROP Chain - into memory and which, when execution returns to the first address in the chain, cause execution to iterate through a series of small code fragments which have the same effect as traditional shellcode. ROP is a powerful technique in environments which disable simultaneous writable and executable memory (W$\oplus$X), since it does not rely on injecting executable code into program memory, but instead relies only on program fragments that already exist. These program fragments are called *gadgets*, and each gadget consists of a (typically small) sequence of instructions followed by a return. On aligned architectures, these returns are part of the intended instruction stream that make up the program, but on unaligned architectures such as x86, these returns can also emerge from jumping into the instruction stream at unintended offsets and causing the existing code to be interpreted differently from what was intended. ROP techniques have been used in attacks on real world systems, including recent attacks exploiting CVE-2018-5767[1], CVE-2018-7445[2] and CVE-2018-16865/6[3].

Numerous techniques have been proposed to mitigate against ROP exploits, including return address verification techniques [2] and control flow verification [1] which aim to prevent control flow being redirected towards a ROP chain. Attempts have also been made to attempt to remove or render unusable ROP gadgets themselves [4]. This paper describes ROP exploit mitigations in OpenBSD which are motivated by gadget reduction and removal, though some mitigations also verify return control flow through return address verification.

In order to mount a successful ROP attack against a vulnerable binary, the attacker must first catalogue all of the gadgets available in a given binary, then identify a sequence of gadgets which will result in their desired effect. This process of scanning binaries for gadgets and then constructing ROP chains which have a desired outcome is somewhat tedious and error prone, so numerous tools exist to make this easy, such as ROPGadget[4], ropper[5], angrop[6], or pwntools[7]. In this paper we will rely on the output from one of these tools, ROPGadget, to measure our effectiveness. Specifically, we will use the number of unique gadgets found by this tool to measure the effectiveness of gadget removal in the OpenBSD kernel and libc, which we have chosen because they are large and diverse binary objects, and are popular exploitation targets. ROPGadget also includes an option to generate a ROP chain that results in an exploited program executing a command shell. Obtaining a command shell is a common exploitation goal, since once an attacker has a command shell they can execute arbitrary other commands on the compromised system. We will use this feature to estimate the effectiveness of our efforts to impede mounting successful ROP attacks

[1] https://www.fidusinfosec.com/remote-code-execution-cve-2018-5767/
[2] https://www.secureauth.com/labs/advisories/mikrotik-routeros-smb-buffer-overflow
[3] https://www.openwall.com/lists/oss-security/2019/01/09/3
[4] https://github.com/JonathanSalwan/ROPgadget
[5] https://github.com/sashs/ropper
[6] https://github.com/salls/angrop
[7] http://docs.pwntools.com/en/stable/

against OpenBSD binaries. The output from the ROPGadget ropchain option is shown in Figure 1, and illustrates several important concepts in ROP attacks. First, the program scans the given binary and identifies all unique gadgets present - this output is shown first. Next, in order to successfully take control of the program and spawn a command shell, gadgets of certain *classes* must be found. These gadgets are listed after each [+] symbol, and show the address of the gadget and the instructions that will be executed when the program jumps to that address. The first class of gadget is a *write-what-where* gadget, which allows values to be moved between registers and memory locations. Next, a gadget must be found which can set the syscall number required for the *exec* system call - notice that these gadgets manipulate the value of the RAX register. The third class of gadget needed sets the arguments to the exec syscall, and these gadgets manipulate the RDI and RSI registers to set the arguments to the exec system call to be */bin/sh*. Finally, the last gadget type is the syscall gadget, which will execute the system call set up by the other gadgets. After identifying suitable gadgets in each class, ROPGadget will construct a ROP chain that will direct program flow through these gadgets in such a way to cause the program to execute a command shell. ROPGadget outputs the ROP chain as a python program which can easily be inserted into whatever exploit tool is being developed to target a specific program or binary. This level of ease and accessibility is typical of ROP tooling, and illustrates the ease with which ROP attacks can be mounted once a suitable vulnerability is identified which takes control of program execution.

We can see from this output that a variety of gadgets are required in order to mount successful ROP attacks on binaries, and that there are a large number of unique gadgets available in a typical binary. These observations motivate our approach of reducing the number of gadgets available in a typical binary - if we can reduce the number of unique gadgets enough then the remaining gadgets will be insufficient to mount a successful attack. In particular, if we can remove all gadgets of a particular class, then it may become impossible to mount some kinds of attacks against OpenBSD binaries, such as the *exec("/bin/sh")* attack shown in Figure 1. We therefore do not need to reduce the number of gadgets in a binary to zero in order to foil ROP attacks, we only need to remove enough gadgets, or enough types of gadgets, so that an attacker cannot cobble together a viable ROP chain.

2 Removing Gadgets

ROP gadgets depend on a sequence of instructions terminating on a return instruction. On aligned architectures, such as arm64, these return instructions are part of the intended instruction stream and are part of usual function epilogues. On unaligned architectures, such as x86/amd64, there are additional return instructions which arise when jumping into the instruction stream at offsets other than those corresponding to the intended stream of instructions. These *polymorphic gadgets* terminate on return instructions that are intended to be part of constants, multibyte instructions, or other artifacts in programs other than real return instructions. In this section, we discuss each kind of gadget and our techniques to remove or reduce them in compiled binaries.

2.1 Aligned Gadgets

Aligned gadgets terminate on intended return instructions as part of normal function epilogues. On aligned architectures these gadgets comprise some or all of the usual process of restoring register state before a function returns. On unaligned architectures these gadgets can also have entirely different effects depending on the offset where the gadget begins in the instruction stream. Examples of aligned gadgets on amd64 and arm64 are shown in Figures 2 and 3. Both of these examples are found in function epilogues, and we show both the bytes that make up the instruction and the instruction disassembly. Throughout this paper we will show both the bytes in the compiled binary and the disassembled instructions, since the interpretation of the bytes making up a compiled program is central to the concept of return oriented programming. For readers unaccustomed to inspecting program disassembly, it may be fruitful to use *objdump(1)* to disassemble and inspect various programs and libraries on their systems.

Figure 2: Aligned gadget on amd64

Bytes	Disassembly
`0f b6 c0`	`movzbl %al, %eax`
`5d`	`popq %rbp`
`c3`	`retq`

Figure 3: Aligned gadget on arm64

Bytes	Disassembly
`fe 03 05 aa`	`mov x30, x5`
`c0 03 5f d6`	`ret`

Since aligned gadgets terminate on function return instructions which are required for correct program operation, our strategy to prevent these return instructions being used in ROP gadgets will be to make them difficult to use outside of normal program flow. To this end, we insert interrupt instructions before the returns and then add instrumentation to the function that will allow normal program flow to jump over the interrupts. Program flow that starts at an offset other than the normal function entry point will fail to jump over the interrupts and abort.

Figure 1: ROPGadget ropchain against OpenBSD 6.3 libc

```
$ ROPgadget.py --ropchain --binary OpenBSD-6.3/libc.so.92.3

Unique gadgets found: 8453
ROP chain generation
===========================================================
- Step 1 -- Write-what-where gadgets
        [+] Gadget found: 0x1f532 mov qword ptr [rsi], rax ; pop rbp ; ret
        [+] Gadget found: 0x3b62e pop rax ; ret
- Step 2 -- Init syscall number gadgets
        [+] Gadget found: 0xfa0 xor rax, rax ; ret
        [+] Gadget found: 0x38fe inc rax ; ret
- Step 3 -- Init syscall arguments gadgets
        [+] Gadget found: 0x4cd pop rdi ; pop rbp ; ret
        [+] Gadget found: 0x905ee pop rsi ; ret
- Step 4 -- Syscall gadget
        [+] Gadget found: 0x9c8 syscall
- Step 5 -- Build the ROP chain
        p = ''
        p += pack('<Q', 0x00000000000905ee) # pop rsi ; ret
        p += pack('<Q', 0x00000000002cd000) # @ .data
        p += pack('<Q', 0x000000000003b62e) # pop rax ; ret
        p += '/bin//sh'
        p += pack('<Q', 0x000000000001f532) # mov qword ptr [rsi], rax ; pop rbp ; ret
        p += pack('<Q', 0x4141414141414141) # padding
        p += pack('<Q', 0x00000000000905ee) # pop rsi ; ret
        [elided ...]
        p += pack('<Q', 0x00000000000038fe) # inc rax ; ret
        p += pack('<Q', 0x00000000000009c8) # syscall
```

RETGUARD

RETGUARD is a mechanism that adds instrumentation to the prologue and epilogue of each function that terminates in a return instruction. In the prologue, we combine the function return address with a random cookie and store the resulting RETGUARD cookie in the stack frame. In the epilogue we verify that the return address is the same one we recorded on function entry. If the addresses match, then we jump over a sequence of interrupt instructions which precede the return. If not, then the program falls through into the interrupt instructions and aborts. By inserting interrupts before the return, we mitigate against gadgets which begin shortly before the return, and larger gadgets must pass the verification process in order to jump over the interrupts and reach the return.

The random cookies used in RETGUARD are drawn from the OpenBSD *.openbsd.randomdata* section. This special read-only ELF section is pre-filled with random byte values at load time by the kernel and dynamic loader (ld.so) whenever executables or shared library objects are loaded into memory. Programs needing high quality random data can allocate memory in this section and be guaranteed that the memory will be randomized when program execution begins. RETGUARD allocates one 8 byte random cookie per function, so the RETGUARD cookie is unique per function and per call.

amd64

The RETGUARD prologue and epilogue for amd64 are shown in Figures 4 and 5. In the prologue, we fetch the function's random cookie and combine it with the return address, then store the resulting RETGUARD cookie in the stack frame. The RETGUARD cookie is calculated before frame setup, and the cookie is stored in the frame along with any other callee saved registers. Unlike the stack protector cookie, the location of the retguard cookie in the stack frame is not important, so it can be stored anywhere in the frame.

The epilogue retrieves the retguard cookie from the frame, combines it with the address we are about to return to, and compares the result with the function's random cookie. If the values match, then the jump is taken over the interrupt instructions and the function returns normally. Otherwise, the program will fall through to the interrupts and the program will abort. A representative program epilogue is shown in Figure 5

By disassembling the epilogue from each offset leading up to the return, we can verify that for each possible offset the program must either pass the random cookie check or terminate on an interrupt. Since each disassembled 'gadget' contains an interrupt instruction, gadget tooling like ROPGadget will recognize the instruction sequence as unusable, with the consequent effect that these gadgets are effectively removed from the compiled binary. In the future, should ROP tooling become clever enough to recognize the jump before

Figure 4: RETGUARD Prologue (amd64)

Instruction	Description
`mov off(%rip),%r11`	`load random cookie`
`xor (%rsp),%r11`	`xor return addr`
`push %rbp`	
`mov %rsp,%rbp`	
`push %r11`	`save retguard cookie`

Figure 5: RETGUARD Epilogue (amd64)

Instruction	Description
`pop %r11`	`load retguard cookie`
`pop %rbp`	
`xor (%rsp),%r11`	`xor return addr`
`cmp off(%rip),%r11`	`compare random cookie`
`je 2`	`jump if equal`
`int3`	`interrupt`
`int3`	`interrupt`
`retq`	

the interrupts, then the random cookie comparison will still need to be passed before the jump can be taken. In this way, RETGUARD effectively removes aligned gadgets from programs.

arm64

For arm64, the function prologue and epilogue are similar, and are shown in Figures 6 and 7. The difference between the amd64 and arm64 versions is that because arm64 is an aligned architecture, we do not need to perform the disassembly exercise for each offset leading up to the return instruction - the only instructions available as ROP gadgets are the instructions as they were intended.

Again, we see that each possible gadget in the function epilogue contains an interrupt, and will therefore be ignored by ROP gadget tooling. Should an attacker attempt to use these gadgets anyway, then the return address verification step will still need to be passed in order to bypass the interrupt. Again, the RETGUARD instrumentation effectively removes these gadgets from the binary.

Stack Protection

Finally, although the intent of RETGUARD is to make it difficult to use function return instructions as ROP gadgets, the return address verification mechanism in the epilogue has the same effect as enforcing control flow on the program. If the return address is modified on the stack, then the program will abort. This is the same effect as the existing stack canary, which is placed on the stack immediately before the return address. RETGUARD improves upon the stack canary mechanism by allocating one random cookie per function instead

Figure 6: RETGUARD Prologue (arm64)

Instruction	Description
`adrp x15, #pageoff`	`load random cookie`
`ldr x15, [x15, #off]`	`load random cookie`
`eor x15, x15, x30`	`xor return addr`
`str x15, [sp, #-16]!`	`save retguard cookie`

Figure 7: RETGUARD Epilogue (arm64)

Instruction	Description
`ldr x15, [sp], #16`	`load retguard cookie`
`adrp x9, #pageoff`	`load random cookie`
`ldr x9, [x9, #off]`	`load random cookie`
`eor x15, x15, x30`	`xor return addr`
`subs x15, x15, x9`	`compare random cookie`
`cbz x15, #8`	`jump if equal`
`brk #0x1`	`interrupt`
`ret`	

of one cookie per object file, and directly verifying the return address instead of verifying the stack canary and inferring the integrity of the return address. In this way, RETGUARD provides a stronger stack protection mechanism than simple stack canaries.

2.2 Polymorphic Gadgets

Polymorphic gadgets terminate on unintended return instructions. These gadgets do not exist on aligned architectures, but on x86, there are four bytes which decode to a return [3]: *c2*, *c3*, *ca*, and *cb*. These are shown in Table 1 along with their meanings. The four kinds of return are divided between near (c2, c3) and far (ca, cb) returns, and returns that pop additional data off the stack (c2, ca) or not (c3, cb). The most common kind of return found in ordinary programs is the c3 return, which is also the easiest kind to employ in ROP gadgets since it is a near return that does not change the current code segment or adjust the stack.

Any time any of these bytes occur in the instruction stream, they represent a potential gadget. These bytes occur in three main parts of programs:

Instruction Encoding Instructions that encode with a return

Table 1: x86 Return Instructions

Byte	Meaning
c2 *imm16*	Near return to calling procedure and pop *imm16* bytes from stack.
c3	Near return to calling procedure.
ca *imm16*	Far return to calling procedure and pop *imm16* bytes from stack.
cb	Far return to calling procedure.

Table 2: ModR/M Byte Encodings

ModR/M Byte	1st Operand	2nd Operand
c2	rax, r8	rdx, r10
c3	rax, r8	rbx, r11
ca	rcx, r9	rdx, r10
cb	rcx, r9	rbx, r11

Table 3: SIB Byte Encodings

SIB Byte	Base	Index	Scale
c2	rdx, r10	rax, r8	8
c3	rbx, r11	rax, r8	8
ca	rdx, r10	rcx, r9	8
cb	rbx, r11	rcx, r9	8

byte as part of the instruction, either as part of the instruction directly, or through the encoding of the *ModR/M* or *SIB* byte.

Constants Instructions which use a numeric constant containing a return byte, such as loading a literal value onto a register. Since OpenBSD is compiled fully PIE (position independent executable), these are always value literals, since there are no address constants.

Relocation Addresses Instructions which reference a value located in another program object such as a shared library have the locations of these objects filled in at runtime. Sometimes the location value includes a return byte.

Examples of gadgets arising from each of these program parts are shown in Figure 8, which shows for each source of polymorphic gadgets the bytes making up the intended instruction(s), what the intended instruction was, and what the gadget instructions are. In each example, the gadget bytes are highlighted for easy identification. In the first example, the unintended return instruction comes from a ModR/M byte encoding the eax / ebx register pair. In the second example, the constant value loaded into rdi contains a c3 byte. In the last example, the address of the bcmp function happens to encode with a c3 byte.

In the case of instruction encodings, the majority of unintended return instructions originate from the *ModR/M* or *SIB* byte of instructions that operate on one or two registers. For some combinations of registers the ModR/M or SIB byte will be encoded as a c2, c3, ca or cb byte, and therefore constitute a possible return. These register combinations are shown in Tables 2 and 3, where the identified registers can also be referenced by their 8 (low), 16, 32 or 64 bit aliases (*eg*. al, ax, eax, rax).

We pursue two strategies for removing polymorphic gadgets from binaries. We first attempt to transform instructions containing return bytes into equivalent instructions that do not contain any. If this is not possible, either because there is no equivalent instruction, or because it is not safe to transform, then we prepend the instruction with an alignment sled. This alignment sled is a jump instruction followed by 2-9 interrupt instructions. The intent of the alignment sled is to limit the offsets from which a gadget may start and which will terminate on the unintended return byte. By inserting several interrupt instructions before the problematic instruction, we increase the likelihood that any execution starting from an unintended offset in the instruction stream will execute an interrupt and abort.

We implemented two approaches for transforming problematic instructions into safe alternatives: Alternate register selection; and Alternate code generation. These strategies are detailed below.

Alternate Register Selection

A survey of ROP gadgets present in the OpenBSD amd64 kernel revealed that many polymorphic gadgets result from c3 bytes which encode operations on the B series of registers (rbx, ebx, bx, bl). We can therefore reduce the number of gadgets by simply reducing the use of these registers. We have modified the register allocation preference in the clang compiler to place these registers after each of the other general purpose registers, so that the B registers are assigned last. As a consequence, many functions which do not need all of the available general purpose registers will never use the B registers, and will therefore not be at risk of encoding unintended c3 bytes when operating on those registers.

This change is straightforward to implement - we simply change a list:

Before RAX, RCX, RDX, RSI, RDI, R8, R9, R10, R11, **RBX**, R14, R15, R12, R13, RBP

After RAX, RCX, RDX, RSI, RDI, R8, R9, R10, R11, R14, R15, R12, R13, **RBX**, RBP

This change is entirely free. There is no additional compile time cost, and no additional runtime cost. Despite being free and trivial, we shall see in Section 3 that it has a measurable effect on the number of unique gadgets present in OpenBSD binaries.

Alternate Code Generation

For instructions which do use the B series registers or other pairs of registers which can encode a return byte (as per Tables 2 and 3), or which may use a problematic constant, we have modified the clang compiler to inspect each instruction before it is emitted and attempt to exchange these problematic instruction for safe alternatives. This is the *X86FixupGadgets* pass, which identifies potential ROP gadgets and attempts to mitigate against them. An initial implementation of this pass

Figure 8: Types of polymorphic gadgets

Gadget Source	Bytes	Intended Instruction	Gadget Instruction
Instruction Encoding	`83 e3 01 01 c3`	`andl $1, %ebx` `addl %eax, %ebx`	`add %eax, (%rcx)` `ret`
Constant	`48 c7 c7 a5 c3 84 81`	`movq 0x8184c3a5, %rdi`	`movsl (%rsi), (%rdi)` `ret`
Relocation Address	`e8 95 c3 3e 00`	`callq 4113301 <bcmp>`	`xchgl %ebp, %eax` `ret`

transformed a specific subset of instructions that encoded to include c3 return bytes, and was included in OpenBSD 6.4. A more general version of this pass is being prepared for OpenBSD 6.5 that targets all four kinds of return bytes and problematic constants. This pass uses two general strategies for gadget reduction: direct instruction modification and alignment sled padding.

For instructions which include a return byte because of their particular register operand encoding, we observe that the same instruction with the register operands reversed does not result in a return byte in the emitted instruction. For example, an instruction encoding a ModR/M byte using rax as the first operand and rbx as the second operand will emit a ModR/M byte of c3, according to Table 2, but if the operands were reversed, so the first operand was rbx and the second was rax, the ModR/M byte would not encode any of the four return bytes (it would instead encode to *d8*). This relationship holds for all of the problematic register pairs identified in Tables 2 and 3 - we can always reverse the operands and emit a safe instruction. Similarly, for instructions which use only a single register operand, we can safely substitute the equivalent A series register. Our strategy for fixing instructions which encode unintended return bytes through the ModR/M or SIB bytes is therefore to insert an exchange instruction before and after the problematic instruction which swaps the values of the operand registers, and then modify the instruction to reverse the order of the operands. The effect is to perform the exact same operation as the intended instruction, but do it with the operands reversed. The resulting instructions are free of unintended return bytes and cannot terminate ROP gadgets. An example of this transformation is shown in Figure 9, which shows the original instruction bytes and intended instruction and the transformed bytes and instructions. Notice that the transformed instructions do not contain any return bytes.

For instructions which cannot be modified by exchanging their operands, or which encode constants that include a problematic byte, we insert an alignment sled before the instruction in order to interfere with gadgets which terminate on the unintended return byte. There are many reasons why we may not be able to reverse the operands used in a given instruction, such as instances when one of the registers is not a general purpose register (such as the xmm registers), the byte value is non-optional (such as the VMRESUME instruction, which encodes as *'0f 01 c3')*, the instruction is a branch or other instruction that changes control flow, or the instruction implicitly uses a register that is also one of the operands. The alignment sled is a jump instruction followed by a series of 2-9 interrupt instructions. The effect of the interrupt instructions is to cause unaligned access to the instruction stream to result in the program aborting.

Figure 10 shows the effect of inserting an alignment sled before a problematic instruction that encodes a constant with a return byte. By placing the alignment sled before the instruction, any gadgets which would have used the c3 byte as a return are constrained to avoid executing any of the interrupt instructions which precede it, with the result that unaligned execution of the instruction stream is impractical, and the c3 byte cannot be used as a return and therefore cannot be used in a ROP gadget.

3 Results

OpenBSD has applied these mitigations to the amd64 and arm64 platforms. RETGUARD has been applied to both platforms for the 6.4 release, and mitigations targeting polymorphic gadgets have been applied on the amd64 platform over the 6.3 and 6.4 releases. An enhanced version of the alternate code generation mitigation is planned for the 6.5 release.

3.1 arm64

RETGUARD was applied to the arm64 platform during the OpenBSD 6.4 release cycle. Compared to the OpenBSD 6.3 release, the number of gadgets found by ROPGadget decreased from 69935 to 46, as shown in Table 4. This decrease is attributable to the arm64 platform requiring instruction alignment, so each function protected by RETGUARD becomes effectively gadget free. The remaining 46 gadgets are all from assembly level boot code functions, which are unmapped after boot. Consequently, the OpenBSD kernel on arm64 is effectively gadget free after boot. The results in userland are much the same, with only a small number of assembly level functions contributing gadgets to userland executables and libraries. These small numbers of gadgets are generally insufficient for constructing arbitrary ROP chains,

Figure 9: Instruction transformation

Original Bytes	Original Instruction	Transform Bytes	Transform Instruction
48 89 c3	mov %rax,%rbx	48 87 d8 48 89 d8 48 87 d8	xchg %rbx,%rax mov %rbx,%rax xchg %rbx,%rax

Figure 10: Alignment sled

Original Bytes	Original Instruction	Transform Bytes	Transform Instruction
49 bc c3 f5 28 5c 8f c2 f5 28	mov $0x28f5c28f5c28f5c3,%r12	eb 06 cc cc cc cc cc cc 49 bc c3 f5 28 5c 8f c2 f5 28	jmp 6 int3 int3 int3 int3 int3 int3 mov $0x28f5c28f5c28f5c3,%r12

Table 4: Number of Kernel Gadgets (arm64)

OpenBSD Version	# Unique Gadgets
6.3	69935
6.4	46

and so executing ROP attacks on OpenBSD binaries on the arm64 platform is generally more difficult or impossible.

3.2 amd64

ROP mitigations have been applied to the amd64 platform over several release cycles. The alternate register selection mitigation was applied for the 6.3 release. An implementation of the alternate code generation mitigation targeting some common gadget forms was applied for the 6.4 release, in addition to RETGUARD.

The alternate register selection mitigation removed approximately 6% of unique gadgets from the kernel, with negligible impact on code size and performance. The alternate code implementation removed an additional 5% of unique gadgets, at the cost of 6 bytes of additional code per transformation (which yielded an approximately 0.15% larger kernel). Since the xchg instruction is inexpensive to execute, the performance impact of this mitigation was negligible.

At the time it was applied, RETGUARD removed approximately 50% of total gadgets from the OpenBSD kernel, and around 20% of unique gadgets. The RETGUARD instrumentation adds 31 bytes per function, and increased the size of the kernel by approximately 7%. Additionally, each function reserves 8 bytes of space in the *.openbsd.randomdata* section for its random cookie, with the consequence that the random data section grows significantly compared to OpenBSD 6.3, and takes more time to fill when an executable launches. Performance overhead of RETGUARD is divided between startup cost, which is dominated by generating the random cookies for each function, and the runtime cost of executing the instrumentation in each function. For a typical system build workload, the runtime cost of RETGUARD is approximately 2%. Results summarizing the number of unique gadgets found in the OpenBSD amd64 kernel across releases is shown in Table 5. This table shows the OpenBSD version, number of unique gadgets, and kernel size for successive OpenBSD releases, with preliminary numbers shown for OpenBSD 6.5, which is currently in development. When reading this table, it is important to point out that many things were added to the kernel during each release cycle which contributed to the overall size of the kernel and the number of gadgets. For example, in the 6.4 release, RETGUARD accounted for approximately 7% of the additional code size, but the kernel grew by approximately 17%. The remaining 10% of code was new drivers and other enhancements, and this code also contributed to the overall gadget count. With this in mind, we introduce a new metric for measuring *gadget density*: unique gadgets per kilobyte. With this metric, we can estimate the effect of ROP gadget mitigations independent of the code size and more easily compare the effectiveness of gadget reduction over releases. Figure 11 shows kernel gadget density over several OpenBSD releases, including an estimate of new alternate code generation mitigations planned for OpenBSD 6.5.

In userland we found similar results, with the number of unique gadgets declining in successive OpenBSD releases. Figure 12 shows the total number of unique gadgets in the popular *sshd* binary and all linked libraries. This figure shows a re-

Figure 11: Kernel gadget density (amd64)

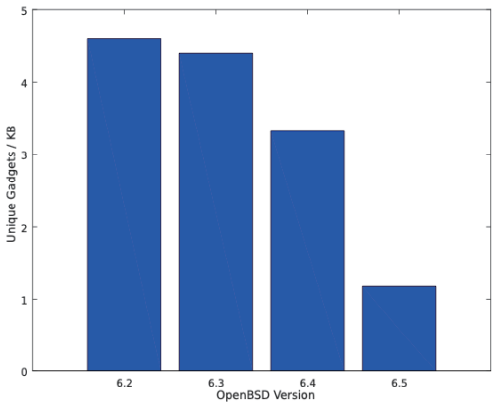


Figure 12: Number of gadgets in sshd and libraries (amd64)

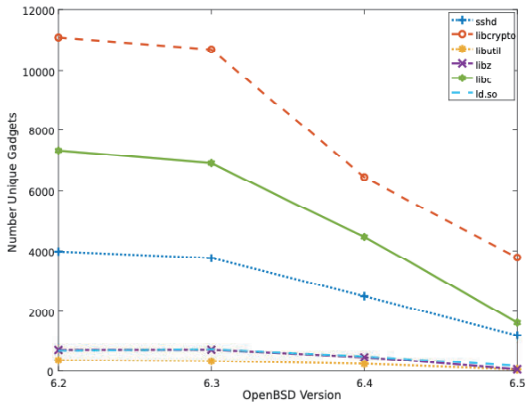


Table 5: Number of Unique Kernel Gadgets (amd64)

Version	# Unique Gadgets	Size (kB)
6.2	60589	13167
6.3	57980	13190
6.4	51229	15438
6.5	21807	15852

Figure 13: ROPGadget ropchain against OpenBSD 6.4 libc

```
$ ROPgadget.py --ropchain
               --binary OpenBSD-6.4/libc.so.92.5

Unique gadgets found: 5994
ROP chain generation
===========================================================
- Step 1 -- Write-what-where gadgets
  [-] Can't find 'mov qword ptr [r64], r64' gadget
```

duction of over 70% in the number of unique gadgets present in a typical running sshd executable between OpenBSD 6.2 and the upcoming OpenBSD 6.5.

3.3 Effect on ROP Tooling

Figure 1 showed the output from ROPGadget against the OpenBSD 6.3 libc, where we saw the tool successfully generated a ROP chain that executed a command shell using gadgets found in the libc binary. When we run the same tool against the OpenBSD 6.4 libc, we find that the tool fails to find a ROP chain that will result in the program executing a command shell. This is shown in Figure 13, where we see that after applying the mitigations described in this paper, the ROP tool is incapable of finding a write-what-where gadget, and therefore unable to construct a ROP chain that will execute a command shell. This result is true of many of the binaries and libraries included in OpenBSD 6.4, including sshd and all of it's linked libaries.

4 Conclusion

In this paper we have described a series of ROP mitigations applied in OpenBSD for both aligned and polymorphic (unaligned) ROP gadgets. For aligned gadgets, we have deployed RETGUARD on both amd64 and arm64 platforms, which resulted in a significant reduction of unique gadgets on amd64, and an almost total elimination of gadgets on arm64. For polymorphic gadgets, we have deployed a series of mitigations that eliminate gadgets either through trivial changes to register selection preferences, direct instruction modification, or forcing alignment in the instruction stream. These mitigations have resulted in a significantly reduced number of gadgets in OpenBSD binaries, both in raw numbers and in gadget density. We have shown that, as a result of these efforts, constructing ROP chains on OpenBSD is more difficult than before, and specifically shown that common ROP tools are now unable to construct ROP chains that execute a command shell against OpenBSD's libc and other binaries and libraries.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[2] Thurston HY Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566. ACM, 2015.

[3] Intel®. 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[4] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58. ACM, 2010.

[5] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.

powerpc64 architecture support in FreeBSD ports

Piotr Kubaj, Bsc

1. Abstract

IBM POWER processors are 64-bit CPU's designed primarily for server market. With POWER9, there has been renewed interest in them, due to the use of open-source firmware and focus on security and control of the hardware. There are also new desktop boards with POWER9. Because of that, support for them has been recently greatly improved in FreeBSD with increased driver compatibility and more 3rd party software having available. For my project, I build the whole ports tree using Poudriere and fix the compilation errors I meet. In this paper, I specify challenges met during porting software to work on POWER processors on FreeBSD and show how most problems can be solved. FreeBSD on POWER architecture runs in big-endian variant only and uses old toolchain – with GCC 4.2 and binutils 2.17. This is why many problems are related to fixing bugs in big-endian variants of code and solving issues related to the old toolchain that the operating system uses.

Keywords: freebsd, powerpc64, ibm power, ports, endianness, big endian, little endian.

2. Introduction

2.1. POWER architecture

This paper describes the effort for running FreeBSD ports on powerpc64 architecture. IBM POWER are 64-bit RISC processors designed specifically for the server market, although there are also boards for desktops and workstations using those processors.

Before POWER9, last POWER processors released for desktop computers, were PowerPC 970MP (based on POWER4) available in Apple PowerMacs G5.[1]

With the release of POWER9, there appeared new desktop computers with POWER processors, making this architecture again interesting for desktop users.[2]

IBM also made the firmware fully free and open source, making this platform owner-controllable, in contrary to other CPU vendors. This aspect makes it more appealing to some users than its competitors.[3]

2.2. Endianness

Endianness is the order in which bytes are read in larger numerical values.
In big-endian architectures, the most significant byte is stored the first (has the lowest address). This is similar to the order used commonly by people.

Little-endian architectures are the opposite. The first number is the least significant byte. The most significant byte is the last one in a given number and has the highest addres. This is in contrary to number ordering used in everyday life.[4]

POWER8 and POWER9 processors are bi-endian – they can run in both modes. However, older POWER CPU's, like PowerPC used in Apple Macintoshes, can only run in big-endian mode (there are some older POWER CPU's able to run little-endian mode but they are not common). Because of that, FreeBSD currently only supports big-endian mode. It is also important that some buses may be little-endian, even though the CPU is big-endian – e.g. PCI bus. That means device drivers need to be endianness-aware to work with both little-endian and big-endian processors.

AMD64 architecture is little-endian only. This causes more problems, because some applications are not properly tested on big-endian architectures.

Some examples of how endianness works in practice are below. They come from the hexdump of /bin/sh on amd64 and powerpc64.

0x7f454c46 is the magic number of ELF format (marked as yellow in the Figure 1) (0x7f followed by ELF in ASCII). In little-endian variant the values are reverted (offset 0x00). [5]

After the magic number and the number telling whether the architecture is 32-bit or 64-bit, follows the number which says whether the given architecture is little- or big-endian (1 meaning little-, 2 meaning big-, marked in blue).[6]

Then, in offset 0x20, there is a value which keeps the address of program header table. In 64-bit architectures, it is at offset 0x40 (marked in red).

```
0000000 457f 464c 0102 0901 0000 0000 0000 0000
0000010 0003 003e 0001 0000 f000 0000 0000 0000
0000020 0040 0000 0000 0000 11c0 0004 0000 0000

0000000 7f45 4c46 0202 0109 0000 0000 0000 0000
0000010 0002 0015 0000 0001 0000 0000 1003 6598
0000020 0000 0000 0000 0040 0000 0000 0002 aa28
```

Figure 1: Screenshot of hexdump /bin/sh from little-endian (upperside) and big-endian (downside)

FreeBSD gained support for POWER9 and the previous POWER8 in 12.0-RELEASE.[7]

3. Purpose of this work

My goal is to make powerpc64 platform equal to amd64 in terms of software support. This includes having usual desktop-class software like web browsers or desktop environments available. It also includes having server-class software like database servers.

4. Used hardware and methods

All tests were carried out using Talos Lite board with 4 core IBM POWER9 processor and 64GB RAM.

I used FreeBSD 12.0-RELEASE system with manually merged patches related to POWER from CURRENT branch and ran bulk builds of the whole ports tree using Poudriere.

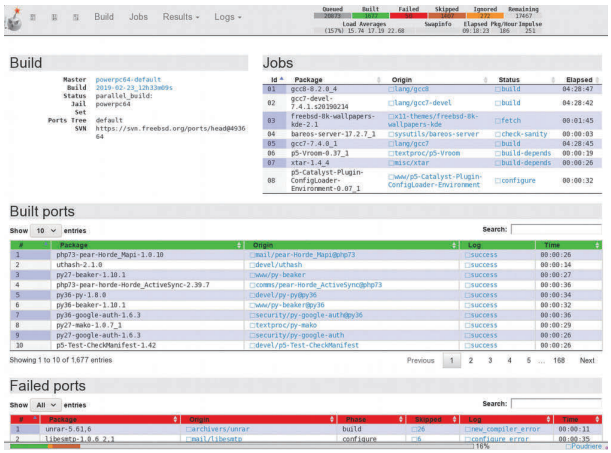


Figure 2: Screenshot of Poudriere web frontend

This screenshot shows the frontend of Poudriere accessible via web browser. I am able to easily see which ports fail to build and which ports are set not to build. It is also possible to browse specific compilation logs.

That allowed me to identify existing problems, fix them one by one, then launch next Poudriere run.

5. Results

Summary of identified problems

There are three main reasons why powerpc64 support in ports tree needs more work:

1. powerpc64 is big-endian on FreeBSD, while all Tier 1 architectures are little-endian.
2. powerpc64 uses (as do all other big-endian architectures on FreeBSD) outdated toolchain in the base system. The default compiler is GCC 4.2. Binutils 2.17 is also used.
3. FreeBSD uses powerpc64 name for 64-bit POWER port, but Linux uses ppc64 name.

1. Endianness problem is cultural and social. The most popular and widely available architectures, amd64 and arm, are little-endian. Developers often do not write software with endianness compatibility in mind.

   FreeBSD/powerpc64 little-endian port is in planning, but this is out-of-scope for this article. However, this will not solve all the problems, because:

   a) As a general rule, POWER processors older than POWER8 can only run big-endian (there are exceptions to that).

   b) Some people may prefer big-endian for various reasons.

   c) That would only solve powerpc64's problems, other big-endian architectures will be left with the same problem.

   d) Creating powerpc64le port will divide already small community of FreeBSD/powerpc* users to two groups – big-endian (powerpc, powerpc64 and powerpcspe) and little-endian (powerpc64le).

Endianness issues cause programs written for little-endian architectures to require byte-swapping functions. Common issues for little-endian-only software are mixed colors in graphic applications, because the hexadecimal value used for a given color is different when read in big-endian mode.

Example of such function is given below:

```
static inline int16_t
ORCT_Swapi16(int16_t x)
{
    return
    static_cast<uint16_t>((x << 8)
    | (x >> 8));
}
```

This code makes byte ordered in little-endian mode to be swapped to big-endian mode.

Unfortunately, the only solution to this problem is a political one – exposing big-endian architectures more and raise awareness that FreeBSD works on many different architectures. This is also troublesome for GNU/Linux ppc64 users and patches for many ports can be adapted to FreeBSD from GNU/Linux. There could also be a little-endian POWER architecture port of FreeBSD, but that would divide the (already small) FreeBSD community of POWER-users.

2. Old toolchain is a technical problem and is specific to FreeBSD.

When FreeBSD migrated from GNU toolchain to LLVM, it started with i386 and amd64 architectures. Soon after, arm platforms followed. This incidentally makes all little-endian architectures use LLVM.

However, all big-endian architectures are left with GCC 4.2 and Binutils 2.17.

The most common issues resulting from using outdated toolchain are:

a) plenty of software nowadays require C11 or C++11 compatibility. Because LLVM on all supported FreeBSD releases supports both, there are hundreds of example where the necessary `USES=compiler` is not put to given port's Makefile.

b) when the port defines `USES=compiler` properly, but its library dependency does not, this creates a linking issue. This happens because the port uses new GCC (and its new ABI), but the linked library is built with base GCC (and old GCC ABI). This problem requires adding `USES=compiler` to the library dependency, even though it compiles with base GCC. Example of such linking error:

```
/usr/local/lib/
libIlmThread.sodefined
reference t to
`std::__cxx11::basic_string
stream<char,
std::char_traits<char>,
std::allocator<char>
>::basic_stringstream(std::_
Ios_Openmode)@GLIBCXX_3.4.21
'
```

c) sometimes a LLVM-specific CXXFLAGS is put to Makefile. GCC in such situation can't compile such software and throws error.

The most prominent error is:

```
CXXFLAGS+=      -Wno-c++11-
narrowing
```

In above case, it is usually enough to instead force compilation in c++98 mode – LLVM compiles by default in c++11. This can be achieved by the following directive:

```
USE_CXXSTD=     c++98
```

d) a common error in software is redefining typedefs. This happens with code in example:

```
typedef struct _Example {
int a;
char b;
} Example;
```

```
typedef struct _Example
Example;
```

Code from the first fragment is put to a file that is included in another file, which has code from the former fragment.
In this case, it is enough to remove the typedef from the former fragment.

Alternatively, one could use appropriate USES directive to force GCC from ports to be used. New GCC allows typedefs to be redefined

The issue of outdated toolchain can be worked around by using newer GNU compiler from ports tree, but such workaround creates other issues:

e) programs do not respect `CXXFLAGS`, linking to base libstdc++, instead of libstdc++ from ports' GCC,

Example for devel/protobuf:

```
--- src/Makefile.am.bak
2018-10-27
21:56:16.784704000 +0200

+++ src/Makefile.am
2018-10-27
22:01:47.564751000 +0200

@@ -518,7 +518,7 @@

 # to build the js_embed
binary using $
```

```
(CXX_FOR_BUILD) so that it
is executable

 # on the build machine in a
cross-compilation setup.

 js_embed$(EXEEXT): $
(srcdir)/google/protobuf/com
piler/js/embed.cc

-       $(CXX_FOR_BUILD) -o
$@ $<

+       $(CXX_FOR_BUILD) $
{CXXFLAGS} -o $@ $<


js_well_known_types_sources
=                              \
google/protobuf/compiler/js/
well_known_types/any.js
\
google/protobuf/compiler/js/
well_known_types/struct.js
\
```

This specific issue alone made protobuf fail to build, resulting in over 400 ports to be skipped.

f) software developers believe that using FreeBSD implies having libc++ in base and add `-stdlib=libc++` to `CXXFLAGS`. This is not necessary, Clang uses libc++ by default and adding it breaks build with GCC.

g) GCC and LLVM include by default slightly different set of headers, resulting in some headers needed to be included manually when using GCC. The most common example is using `sys/types.h`, which contains commonly used typedefs (like uint).

3. There is also a third problem, which is that Linux uses ppc and ppc64 names for POWER architecture ports, while FreeBSD uses powerpc, powerpc64 and powerpcspe names. This issue is also present on amd64 and i386 architectures and is easily fixed:

```
ANT_ARCH=       $
{ARCH:S/amd64/x86-64/:S/i386
/x86/:S/powerpc64/ppc64/}
```

6. Conclusions

Switching to LLVM in base will allow focus only on architecture-related differences with amd64. This is already work-in-progress by some members of the community.[8] Having modern toolchain with C++17 support in FreeBSD base will fix most toolchain issues in ports.

With POWER9 available, there is a renewed interest in big-endian systems. Linux users port more and more software to big-endian architectures and FreeBSD/powerpc* will also be able to benefit from that.

7. References

1. Forever Mac, https://web.archive.org/web/20120930005749/http://www.forevermac.com/2005/10/apple-power-macintosh-g5-quad-core-2-5-ghz/, access from 22.02.2019,
2. Raptor Computing Systems, https://www.raptorcs.com/, access from 22.02.2019,
3. OpenPOWER Foundation, https://github.com/openbmc and https://github.com/open-power, access from 22.02.2019,
4. Mozilla, https://developer.mozilla.org/en-US/docs/Glossary/Endianness, access from 22.02.2019,
5. Unix System Laboratories, http://www.skyfree.org/linux/references/ELF_Format.pdf, page 1-5, access from 23.02.2019,
6. Unix System Laboratories, http://www.skyfree.org/linux/references/ELF_Format.pdf page 1-6 access from 23.02.2019,
7. The FreeBSD Documentation Project, https://www.freebsd.org/releases/12.0R/relnotes.html#hardware-support, access from 22.02.2019,
8. Alfredo Dal Ava, https://wiki.freebsd.org/powerpc/llvm-elfv2, access from 23.02.2019.

Design and Implementation of NetBSD Base System Package Distribution Service

Ken'ichi Fukamachi
Chitose Institute of Science and Technology
k-fukama@photon.chitose.ac.jp

Yuuki Enomoto
Cybertrust Japan Co., Ltd.
yuki.enomoto@cybertrust.co.jp

Abstract

We consider that Unix operating system should be built on fine granular small parts (packages) to improve the system maintenance. It is expected that it enables speedy security update, system update tracking in detail, easy replacement and rollback of specific parts.

We have implemented and run a new service to distribute modular base system userland for NetBSD. We generate the least amount of modular base packages by using `basepkg.sh`. It splits NetBSD daily binaries into 1000 over packages based on `syspkgs` meta-data and ident comparison within the binaries. This scheme drastically reduces the processing time to realize operations within practical time.

Our system have shown that granular update system and service can be implemented and operational under breakdown approach. NetBSD users can maintain NetBSD base system in more granular way with fine update history and build an arbitrary system from the NetBSD minimal installation.

1 Introduction

Historically, before the use of Internet leased lines was popular in 1990s, operating system (OS) had been managed on one source tree and the source tree set has been distributed. The typical example is BSD Unix. It has been developed in its own source tree including kernel, general commands, configuration files, and manuals. BSD Unix distinguishes between the official distribution and 3rd party software.

Another example is Linux distribution. It does not distinct its own base system from third-party software. It assembles a lot of small packages which are created and maintained by many different authors. To manage the whole system, it is inevitable to develop software such as `apt` for Debian GNU/Linux and `yum` (`dnf` in the future) for Red Hat Enterprise Linux.

Aside from the origin of development styles, OS built on fine granular small parts must be preferable to improve the system maintenance. It is expected that it enables speedy security update, system update tracking in detail, easy replacement and rollback of specific parts.

To reconstitute NetBSD base system to be comprised of small parts, we have implemented software (Chapter 4) to dispose the base system to 1000 over parts and run a service (Chapter 5) to distribute them with our experimental client (Chapter 6). In this paper we call our strategy `breakdown` approach in contrast to the bottom up one of Linux distribution.

The rest of this paper is organized as follows. We define terms in Chapter 2. We introduce components of the whole service in Chapter 3. The details of each component are described in Chapter 4, 5 and 6. We discuss several remaining issues in Chapter 7.

2 Terms

The term "package" implies both 3rd party software management and a kind of a container. The usage differs from OS to OS. We need to clarify the terms "base system" and "package". In this paper, we use

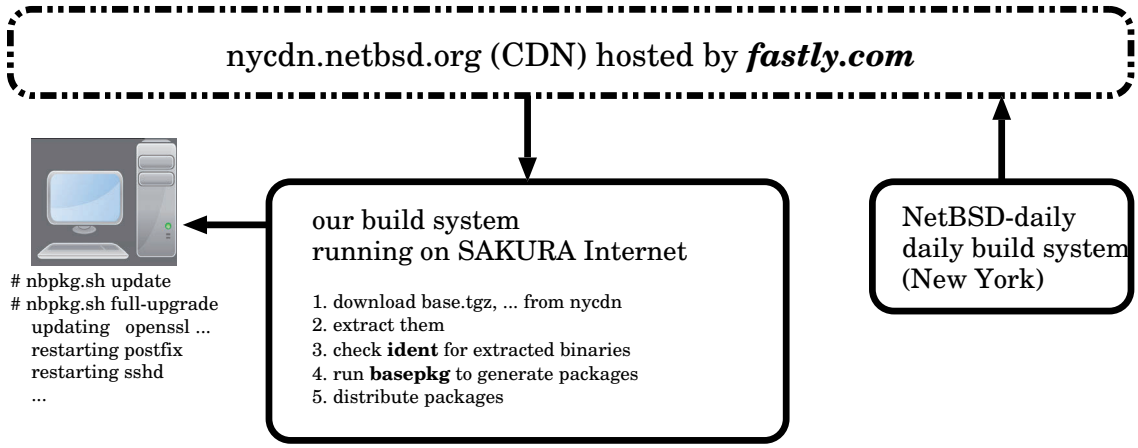


Figure 1: Overview of NetBSD base system package distribution service. It generates base packages by using `basepkg` and distributes them. `nbpkg.sh` client demonstrates updating and restarting.

the term "package" as a container by default.

Linux distributions consider the whole system consists of packages but BSD Unix(s) distinguish between the base system and 3rd party software. BSD Unix(s) consider that the whole system consists of the base system and 3rd party software.

"base system" implies a set of programs officially maintained and distributed by the project. In almost cases, the OS base system distribution is divided by roles to a set of tarballs (which extension is known as ".tgz") such as "base.tgz" (mandatory for the operating system), "comp.tgz" (compiler tools), "man.tgz" (manuals) and so on. BSD Unix base system is composed of a set of 10 or more tarballs.

In the BSD Unix, we manage each 3rd party software as a "package". "package" itself implies a container which consists of software, documentation, configuration files and this package's meta data required to operate in installation and de-installation. We also call the 3rd party software system "package". Each BSD Unix project provides the package system such as `pkgsrc` (NetBSD), `ports` (FreeBSD and OpenBSD) and so on. Users can easily handle the package by using the management system.

3 Components of NetBSD Base System Package Distribution Service

We have implemented and been running a new service to distribute modular base system userland for NetBSD (Figure 1). This distribution system consists of three components: (1) `basepkg`[1, 2] (2) `nbpkg-build.sh`[3] (3) `nbpkg.sh`[3].

`basepkg` splits NetBSD base system into 1000 over packages (we call them `base packages`). `basepkg` is a simple almost POSIX compliant shell script built on `pkgsrc`[4] framework and `syspkgs`[5] meta-data. Hence the naming convention of `base package` is same as `syspkgs` one such as `base-crypto-shlib` (shared libraries for cryptography, classified as a mandatory system).

`nbpkg-build.sh` is the top level dispatcher to run `basepkg` for NetBSD binaries downloaded from `nycdn.netbsd.org`. We generate base packages which changes are detected based on ident (RCS Id) comparison. Though community based development does not have powerful computer resources, those measures reduce the work, as a result, our build sys-

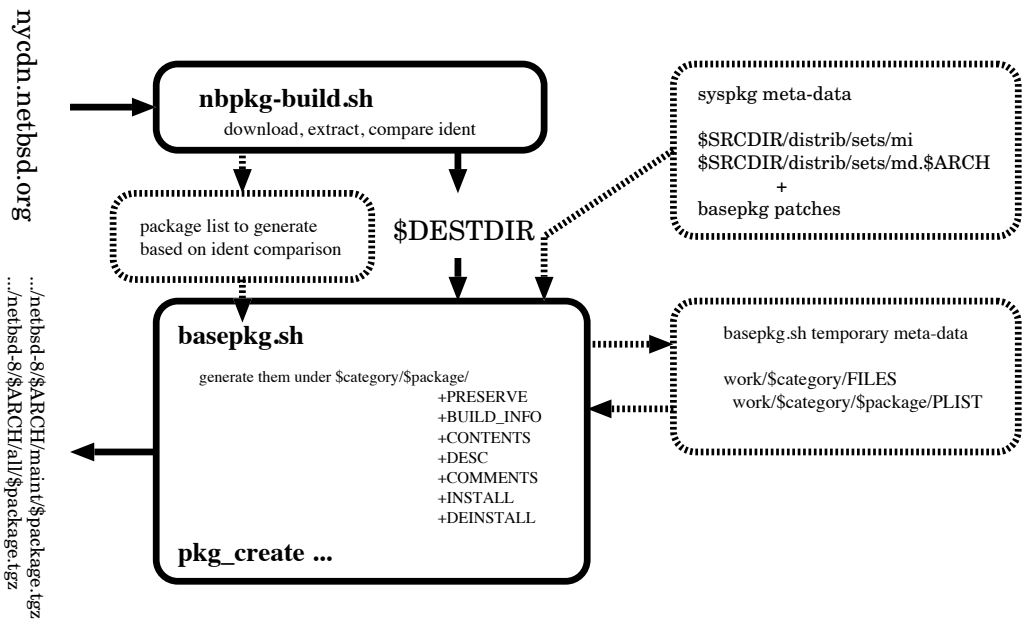


Figure 2: `basepkg` internals

tem, running on low spec VPS[1] , works out daily to provide base packages for NetBSD 8.0 stable branch (62 targets).

`nbpkg.sh` is an experimental client to show our operation model. `basepkg` is built on `pkgsrc` framework so that we can use `pkgsrc` functions as could as possible. `nbpkg.sh` is an extension to `pkgin`[6] (`pkgsrc/pkgtools/pkgin`) which provides `apt/yum/dnf` like functions to maintain the base system more systematically.

4 Basepkg

`basepkg` is a 1200 lines Bourne shell script to split NetBSD base system into 1000 over packages. It consists of meta-data and package build system. The `basepkg` processing (Figure 2) is briefly described below. See the reference[1] for more details.

The meta-data is derived from NetBSD source tree but modified and enhanced by us. The `basepkg` meta-data is based on `syspkgs` one, files in `/usr/src/distrib/sets/lists/`. Each line of the meta-data file contains a set of information (path, syspkgs package name, comments) such as

```
./bin/ls       base-util-root
./bin/rcorder  base-obsolete  obsolete
./bin/rump.dd  base-util-root rump
./usr/bin/cpp  base-util-bin  gcccmds
```

It has been maintained but is inconsistent and contains several bugs. `basepkg` imports the `syspkgs` meta-data and modifies it to fix several bugs and enhances it to support X11.

The actual build process of `basepkg` is running `pkg_*` utilities (`pkgsrc/pkgtools/pkg_install`) to split the base system according to the meta-data.

[1] `bytebench`(`pkgsrc/benchmarks/bytebench`) shows our VPS power is considered to be almost same as a popular home server such as NEC S70 (its CPU is Intel Pentium G6950) on sale in 2011.

```
#   list_arch = amd64 evbarm evbmips evbppc hpcarm i386 sparc64 xen
#        arch = amd64
#      branch = netbsd-8
#    url_base = http://nycdn.netbsd.org/pub/NetBSD-daily/netbsd-8/
#  build_nyid = 201811180430Z
#  build_date = 20181118
#  build_url  = $url_base/$build_nyid/$arch/binary/sets/
for arch in $list_arch
do
    nbdist_download $arch $build_url
    nbdist_extract  $arch
    nbdist_check_ident_changes ...
    if "X$is_ident_changes_found" = "Xyes"; then
        nbpkg_build_gen_basepkg_conf    $arch $branch ...

        # (1) maint mode
        nbpkg_build_run_basepkg         $arch $branch "maint"
        nbpkg_release_basepkg_packages  $arch $branch "maint"

        # (2) all mode
        nbpkg_build_run_basepkg         $arch $branch   "all"
        nbpkg_release_basepkg_packages  $arch $branch   "all"
    fi
done
```

Figure 3: Concept of the `nbpkg-build.sh` main loop

`basepkg` is built on `pkgsrc` framework. It is good to avoid `reinventing the wheel`. `basepkg`-generated package format is same as pkgsrc one, so that we can use features `pkgsrc` provides. For example, we can use `pkg_add/pkg_delete` to add/remove base packages. Moreover, we can use more powerful utility such as `pkgin` which provides `apt/yum/dnf` like functions for pkgsrc. By using `pkg_summary(5)`, `pkgin` resolves associated dependencies among base packages and provides smart operations for installation, removal and upgrade of base packages.

5 Base Package Generation and Distribution System

5.1 Overview

`basepkg` is just a script to split NetBSD base system to 1000 over packages. To provide the NetBSD base system upgrade service, we need to design, implement a package generation system (`nbpkg-build.sh`) described in this section and run the web service at `http://basepkg.netbsd.fml.org`. `nbpkg-build.sh` is the top level dispatcher to run `basepkg` to split NetBSD binaries. Both `nbpkg-build.sh` and the web runs on **SAKURA**

```
# tar -C $DEST_DIR -zxpf $DIST_DIR/*.tgz
# find $DEST_DIR -exec /usr/bin/ident {} \;        > $IDENT_NEW
# diff $IDENT_OLD $IDENT_NEW                       |
  sh $CONVERT_TO_PACKAGE_NAME_VIA_SYSPKGS_DATA     > $TARGETS
```

Figure 4: Brief description of logic on how to compare ident information: NetBSD binaries e.g. base.tgz, etc.tgz, ... are downloaded to $DIST_DIR and are extracted at $DEST_DIR. The final output $TARGETS file contains only `syspkgs` names e.g. base-sysutil-bin. $TARGETS file is passed to `basepkg.sh` as an optional argument.

Internet VPS(v3)[7][2].

`nbpkg-build.sh` (1) downloads binaries from `nycdn.netbsd.org` (fastly CDN) (2) extracts them (3) checks ident within the extracted NetBSD base system binaries and (4) runs `basepkg` to generate base packages.

5.2 Download and Extract NetBSD Binaries

Firstly we need to prepare NetBSD base system binaries to split.

It is preferable to build the binary from the source code to avoid versioning problem (see the Section 7) but it requires a lot of machine resources (both CPU power and storage). To process them within practical time, we decided not to build NetBSD from the source code but download NetBSD binaries provided as NetBSD-daily (what we call **daily build**). The daily build system runs at Columbia University in New York[8] but we can download the binaries via `nycdn.netbsd.org` hosted on `fastly.com` CDN (contents delivery network). Also, our build machine (`basepkg.netbsd.fml.org` running on SAKURA Internet VPS[7]) has enough bandwidth to Internet. Hence we can download binaries enough fast.

It is observed that the time to download and extract them requires not more than 300 seconds (the average is 200 seconds for downloading and 70 seconds for extraction) per one distribution e.g. amd64 on netbsd-8 branch. If we build NetBSD from the source on this host, it requires at least 1200 seconds even in the case of running "build.sh -u release" (18000 seconds in the case of running "build.sh release"). This process contributes to the processing time reduction and does not consume the storage temporarily required for build process.

[2] 3 CORES, 3GB MEMORY, 200GB HDD

5.3 Base Package Generation Strategy

We have implemented the following two plans to consider which is proper in operating NetBSD upgrade service using base packages.

The plan A proposed in AsiaBSDCon2018[1] is that we generate all available base packages daily and determine which base package should be installed according to a given configuration based on NetBSD security advisory. In this case, the implementation of the package generation system is simple but we need huge machine resources. To make matters worse, it is not automatic since somebody needs to edit the configuration.

The plan B is opposite to the plan A. We have only to generate the least amount of base packages having the change after the major release. We can find changes by tracing the ident information in binaries, To implement it, we need to modify both `nbpkg-build.sh` and `basepkg` but this modification can drastically reduce the amount of work in generation base packages. In addition it is important that this mechanism runs automatically.

Hence we decided to use and run the service based

on the plan B. We define the basis of the ident comparison is the most recent major release, NetBSD 8.0 release now.

5.4 Ident Based Comparison

The ident based comparison is straightforward (Figure 4). We check all files on the extracted NetBSD base system and compare the ident information with the previous one to determine which base packages are changed and so should be re-generated. The list of base packages which should be re-generated is passed to the `basepkg` as the configuration.

In this case, `basepkg` has only to build a minimum of packages. Both the saving of targets and downloading via CDN (Section 5.2) drastically reduces the processing time. Currently our processing time per `arch` per `branch` is about 1300 seconds on average. It includes downloading, extraction and `basepkg` which runs two times to generate packages for two modes described below (Section 5.5). It is estimated that we can process 62 targets of netbsd-8 branch within one day. We try to process Tier 1[9] targets twice a day to keep the base package up-to-date as could as possible, but Tier 2 ones once a day.

5.5 Package Dependency Problems

Our breakdown approach introduces a new kind of difficulty on that the basis of comparison is arbitrary. Consider the following cases: (1) keep the system up-to-date mainly for security update (2) build up an arbitrary system from the minimum one. They are similar but the dependencies among base packages differ in essential. In the case 1, we need only packages having the change after the major release. In the case 2, we need to prepare both all base packages for the major release and packages having the change after the major release. The package dependencies in two cases differ since packages in the case 2 demands major release packages if needed. Since the coexistence of different dependency within one package is difficult, our system distributes two kinds of base packages with different dependency at different URLs.

```
[at .../netbsd-8/$ARCH/maint/]
SHA512
base-cron-bin-8.0.20181129.tgz
base-ext2fs-root-8.0.20181129.tgz
base-ipf-bin-8.0.20190119.tgz
...

[at .../netbsd-8/$ARCH/all/]
SHA512
base-adosfs-root-8.0.20180717.tgz
base-amd-bin-8.0.20180717.tgz
base-amd-examples-8.0.20180717.tgz
base-amd-shlib-8.0.20180717.tgz
...
base-cron-bin-8.0.20180717.tgz
base-cron-bin-8.0.20181129.tgz
base-cron-root-8.0.20180717.tgz
...
```

Figure 5: Examples of packaegs for both maint and all modes. The path of "all" mode contains base packages for 8.0 release (suffix .20180717) and packages (e.g. suffix .20181129 and .20190119) which changed after 8.0 release. Howeve the path of "maint" mode contains only changed packages.

In the case 1, we call it `maint` mode, we assume a scenario that users have full-installed NetBSD (e.g. NetBSD 8.0) initially and keep it as the latest NetBSD 8.0 stable (netbsd-8 branch). In this case, we have only to install (overwrite) all base packages having the change after the major release. This update process is able to work automatically. It is possible that each user can install the specific package manually if needed.

In the case 2, we call it `all` mode, it is necessary to think about the possibility that we need to install packages which do not exist on the system. Consider the minimal NetBSD which consists of only base.tgz and etc.tgz. If we newly want to install a C compiler (e.g. `/usr/bin/cc`), which does not exist now, we only have to run ``pkg_add comp-c-bin''. If `comp-c-bin` has no difference from 8.0 release, there is no `comp-c-bin` package in the `maint` mode (case

1). In the case 2, we need to obtain and install 8.0 release `comp-c-bin` base package which name is `comp-c-bin-8.0.20180717.tgz`. Hence in this case, for the possibility building any system from a scratch, we need to prepare both all base packages for the major release and packages having the change after the major release.

The details of package dependency are as follows. For example, let autopsy the dependency of `base-sysutil-bin-8.0.20190119.tgz` tarball. `+CONTENTS` file (`pkgsrc` meta-data) in the tarball in `maint` mode is defined as

```
@pkgdep base-sys-shlib>=8.0.20181129
```

but `+CONTENTS` in `all` mode contains

```
@pkgdep base-sys-root>=8.0.20180717
@pkgdep base-sys-shlib>=8.0.20181129
@pkgdep base-sys-usr>=8.0.20180717
```

where the suffix `8.0.20180717` implies the NetBSD 8.0 release we define artificially since NetBSD 8.0 was released on July 17, 2018. Due to this artificial dependency

```
@pkgdep $package>=8.0.20180717
```

a package which does not exist can be installed automatically.

6 Experimental Client

6.1 Overview

We provide an experimental client `nbpkg.sh` to show our operation model. As mentioned above, `basepkg`-generated package format is same as pkgsrc one, so that we can use full features `pkgsrc` provides. We design our client as a wrapper of `pkgin` to provide integrated service like apt/yum/dnf on Linux distribution.

Our current experimental client uses `/var/db/pkg/` directory for package registration which `pkgsrc` uses originally. Hence `/var/db/pkg/` holds installed package data for both `pkgsrc` and `basepkg`. It is considered to be easy to distinguish `basepkg` packages from `pkgsrc` ones by names, since the naming convention for `pkgsrc` and `basepkg` are very different.

6.2 Usage

`nbpkg.sh` usage is similar to `apt` command. See a demonstration running `update` and `full-upgrade` commands in Figure 6.

At the first time, `nbpkg.sh` checks the environment and install mandatory package management utilities (`pkgsrc/pkgtools/`) such as `pkg_install` (`pkgsrc/pkgtools/pkg_install`) and `pkgin` by default if they are not found.

‘‘`nbpkg.sh update`’’ updates the database (`pkg_summary(5)`) from a remote repository defined by the environmental variable `PKG_PATH` which is hard-coded in `nbpkg.sh`.

‘‘`nbpkg.sh upgrade`’’ is reserved, not recommended currently.

We can use ‘‘`nbpkg.sh full-upgrade`’’ to keep the system up-to-date, the latest one on stable branch, automatically. By default we assume `maint` mode operation described above (Section 5.5). ‘‘`nbpkg.sh full-upgrade`’’ upgrades packages specified by a file `pkg_list2upgrade` in `PKG_PATH` e.g. `http://basepkg.netbsd.fml.org/pub/NetBSD/basepkg/netbsd-8/$ARCH/maint/pkg_list2upgrade`. `pkg_list2upgrade` describes a list of all packages having the change after the major release. It is prepared and updated by our system `nbpkg-build.sh`.

We can install or remove arbitrary packages manually. We assume `all` mode for such operation. For example, to install a C compiler `/usr/bin/cc`, we run ‘‘`nbpkg.sh -a install comp-c-bin`’’ where `-a` option implies all mode (Section 5.5). It resolves the dependecies so that it installs all packages required to use a C compiler. The packages to install are the latest stable one if exists but the last major release (8.0 release now) ones if the package has no change after release.

6.3 Extension

Our client `nbpkg.sh` is not just a wrapper of `pkgin`. It is extended to support (1) fool-proof function not

```
# nbpkg.sh full-upgrade

Running install with PRE-INSTALL for pkg_install-20180425.
man/man1/pkg_add.1
   ...
Package pkg_install-20180425 registered in /var/db/pkg/pkg_install-20180425
   ...

Running install with PRE-INSTALL for pkgin-0.11.6.
bin/pkgin
man/man1/pkgin.1
   ...
Package pkgin-0.11.6 registered in /var/db/pkg/pkgin-0.11.6
   ...
Requesting http://basepkg.netbsd.fml.org/.../maint/pkg_list2upgrade
100% |**********************************| 435 967.66 KiB/s 00:00 ETA
435 bytes retrieved in 00:00 (608.60 KiB/s)
pkgin import /var/db/nbpkg/pkg_list2upgrade
reading local summary...
processing local summary...
processing remote summary (http://basepkg.netbsd.fml.org/.../maint)
  ... snip ...
downloading pkg_summary.gz: ...
calculating dependencies...done.

29 packages to install:
  base-cron-bin-8.0.20181123 base-ext2fs-root-8.0.20181123
  ... snip ...
  xetc-sys-etc-8.0.20181123

0 to refresh, 0 to upgrade, 29 to install
62M to download, 221M to install

proceed ? [Y/n] y
downloading base-cron-bin-8.0.20181123.tgz ...
    ...
installing base-cron-bin-8.0.20181123...
    ...
pkg_install warnings: 0, errors: 0
reading local summary...
processing local summary...
    ...
marking xetc-sys-etc-8.0.20181123 as non auto-removable
```

Figure 6: Example of upgrading the system

to overwrite /etc by accident (2) alias function for us to handle user friendly package names.

To implement the fool proof, `etc-*` packages was removed from `pkg_list2upgrade`. Hence automatic upgrade for `etc-*` packages does not work but you can use explicitly running `nbpkg.sh install etc-*` to update files under `/etc/` (We trust each user for such critical actions).

We support `nbpkg.sh` alias function since `syspkgs` naming convention is too far from the usual convension. For example, `syspkgs` name `base-crypto-shlib` contains `libcrypto.a` and `libssl.a` but generally we call them `openssl shared libraries`. `base-crypto-bin` includes `openssl` command. By our alias support, we can use ‘‘`nbpgk.sh upgrade openssl`’’ to update both openssl libraries and commands. Currently the following aliases are defined.

alias	syspkgs-package-name
libcrypto.so	base-crypto-shlib
libssl.so	base-crypto-shlib
openssl	base-crypto-shlib
openssl	base-crypto-bin
openssh	base-secsh-bin
named	base-bind-bin
bind	base-bind-bin
postfix	base-postfix-bin

We can install `openssh` by running ‘‘`nbpkg.sh install openssh`’’ instead of ‘‘`nbpkg.sh install base-secsh-bin`’’. This definition is hard-coded currently, it is an issue to resolve in the future.

Theoretically we can rollback the specific package manually. After checking package registration logs in `/var/db/pkg`, we run `nbpkg.sh` to remove the installed package and enforce the installation of the previous one.

7 Discussion

We have resolved several issues addressed in AsiaBSDCon2018[1]. Especially the use of `nycdn.netbsd.org` and ident based comparison contributes to the huge reduction of processing time. However it remains a few difficult issues such as base package versioning and dependency discussed in the Section 5.5. We do not discuss `syspkgs` meta-data maintenance problem such as validation of the granularity since these topics are beyond our 3rd-party development scheme.

The versioning problem implies we distinguish base packages by the date suffix not semantic versioning[10] x.y.z e.g. 1.0.0, 1.0.1 and so on.

In the case of the bottom up approach such as Linux distribution, OS assembles a lot of small packages which are created and maintained by many different authors. Each package has each author and versioning e.g. etc-passwd-1.0.1, bin-ls-2.0 and timezone-20190101. The versioning are inconsistent but meaningful in some sense.

In the case of BSD Unix, the whole system is maintained uniformly so that the version is **NetBSD 8.0** for not only the whole system but also all parts in it. Paradoxically we cannot determine the precise version for each granular base package since the author or maintainer is ambiguous for each small parts. For example, consider `etc-sys-etc` package which contains password related files such as `/etc/passwd`, `/etc/master.passwd` and so on. We can assign etc-sys-etc-8.0.0 for NetBSD 8.0 release, but we cannot automatically assign etc-sys-etc-8.0.1 when a part of the `etc-sys-etc` package e.g. `/etc/passwd.conf` is updated on netbsd-8 branch. In addition, NetBSD daily build does not consider the update details and generates the whole NetBSD system on a daily basis. Hence, especially in the case of our system, it is practical to assign etc-sys-etc-8.0.YYYYMMDD e.g. etc-sys-etc-8.0.20180717 not etc-sys-etc-8.0.0.

The date based naming convention may introduce the following problem. If the base package generation does not end within one day, package names for the same source changes may be different among architectures e.g.

```
amd64/all/base-sys-shlib-8.0.20190101.tgz
   ...
zaurus/all/base-sys-shlib-8.0.20190102.tgz
```

However it is considered to be enough practical in order to keep the system up-to-date. We always have

only to install the latest base packages irrespective of the precise package version name since the versioning is consistent within each branch and architecture. Hence the versioning problem must be trivial from the point of practical or operational view.

8 Conclusion

We reorganize NetBSD userland by breakdown approach and run the base package distribution service for NetBSD users.

Our build scheme is based on the use of NetBSD daily binaries via CDN and ident based comparison to generate the least number of base packages. It enables practical time operations.

Our client is useful to maintain the base system in more granular way and build an arbitrary NetBSD system from the minimum. Most importantly, we can upgrade NetBSD with detailed history data stored under `/var/db/pkg/`. We can trace the update details on a daily basis so that we can rollback if needed.

Our approach introduces another package dependency problem but it is trivial from the point of practical view. Our system must be beneficial for NetBSD users.

References

[1] Yuuki Enomoto and Ken'ichi Fukamachi. Design, Implementation and Operation of NetBSD Base System Packaging. *AsiaBSDCon2018 Proceedings*, pp. 21–31, 2018.

[2] Yuuki Enomoto. `basepkg`. `https://github.com/user340/basepkg`. (accessed 2018-12-31).

[3] Ken'ichi Fukamachi. NetBSD modular userland. `https://github.com/fmlorg/netbsd-modular-userland`. (accessed 2018-12-31).

[4] The NetBSD Project. pkgsrc. `https://www.pkgsrc.org`, 1998. (accessed 2019-02-22).

[5] The NetBSD Project. syspkgs. `http://wiki.netbsd.org/projects/project/syspkgs`, 2015. (accessed 2018-12-31).

[6] Emile 'iMil' Heitor. pkgin, a binary package manager for pkgsrc. `https://pkgin.net`. (accessed 2019-02-22).

[7] Sakura Intenet. Sakura Internet VPS Service. `https://vps.sakura.ad.jp/`. (accessed 2019-02-22).

[8] The NetBSD Project. admin. `https://wiki.netbsd.org/users/spz/admin/`. (accessed 2019-02-22).

[9] The NetBSD Project. Platforms Supported by NetBSD. `https://www.netbsd.org/ports/`. (accessed 2019-02-22).

[10] Tom Preston-Werner. Semantic Versioning 2.0.0. `https://semver.org/`. (accessed 2019-02-22).

LLVM and the state of sanitizers on BSD

David Carlier

- French software engineer living in Ireland.
- Contribute to various opensource projects directly or indirectly related to FreeBSD and OpenBSD mainly, from enterprise solutions to more entertaining ones like video games.
- Contributor LLVM since end of 2017, committer since May 2018.
- Used to write for BSDMag.

Status on FreeBSD and OpenBSD

How it had started ?

- It often starts comes with initial frustration.
- Indeed, after having tried fuzzer under Linux and quite pleased with the outcomes, I realized it was not supported under FreeBSD.
- From this point, you can try to solve this “issue” by contributing.
- After a certain time, patience and relentlessness, they might give you the commit accesses.

What is implemented in FreeBSD ?

- address sanitizer (asan).
- thread sanitizer (tsan).
- memory sanitizer for the x86/64 architecture (msan).
- cache frag part of the efficiency sanitizer (esan).
- undefined behavior sanitizer (ubsan).

Also the following components

- libFuzzer.
- X-Ray instrumentation.
- Safestack.
- libCFI

What could be ported

- the working sets part of esan.

Difficult to port

- lsan due to lack of the stop the world feature in the kernel.

What is implemented in OpenBSD ?

- ubsan
- libFuzzer
- X-Ray instrumentation.

What won't be ported

- other sanitizers “due” to ASLR not possible to disable it also not possible to map large regions of memory for the shadow mappings.
- Safestack, even if it was possible, has not much of interest as OpenBSD has similar features out of the box.

What are the key roles of the sanitizers ?

They allow to detect at runtime some well know bugs especially the ones which are difficult to detect in production environments.

- msan is mainly all about detecting not initialized pointers.
- asan is more for memory handling as double and use after free ; heap and stack overflows ... minus the memory leaks detection under BSD as leak sanitizer is not doable at the moment. asan is mutually exclusive with msan.
- ubsan for errors about integer overflows (typical cases are the shift operations), misaligned pointers due to casts with different alignements.
- tsan to detect race conditions in multithread contexts.
- esan basically helps to pack the data structures efficiently to avoid cache fragmentation, as you would try to do manually with tools like phole for example.

What is fuzzing all about ?

- It is a testing technique, “invented” in late 80's by Barton Miller, when basically you try to give random data to your software and its dependencies included.

- Inputs source come from what call “corpus”.

- It is good to find particular set of bugs, based on input handling basically while trying to cover as much as possible code paths by mutating these inputs.

- Ideally running long, as the data will undergo some “mutation” in the process, as necessary until it crashes eventually.

- Completes traditional unit tests set too.

What corpus means ?

- Place holder for inputs, inputs which suit the particular software.

- Let's imagine an image library reader which relies on specific binary format header to recognize if it is png/jpeg and so on.

- The corpus will then contain hand crafted data for a particular test.

- There is possibly more than one corpus but those corpus could possibly be merged, keeping only the relevant mutation results.

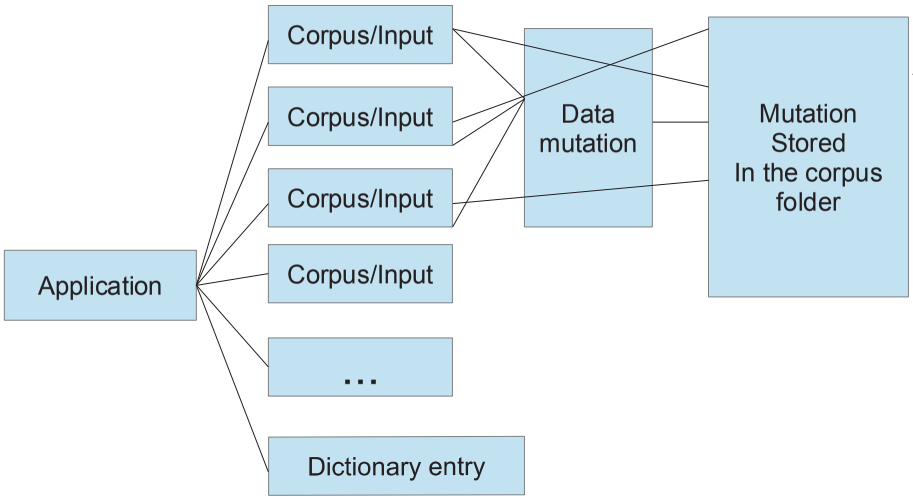
- Those mutations will be then stored (and to be reused) in this corpus.

What mutation means in libFuzzer context ?

- Mutation simply means some bytes are deleted, some others are inserted, got shuffled at random offsets. A dictionary (format key=value) can be used too.

- Software developers might start to have concerns ... That would trigger a segfault, stack overflows or SIGBUS throwing.

Fuzzer workflow



How does it works under LLVM ?

- Basic flag is -fsanitize=fuzzer to gives to either the C or C++ frontend.
- The code in question needs to have a specific entry point at minimum to receive the input data, main is already present. Saying that, there is an optional startup function to implement which catch the arguments.
- You can also implement the mutation and combination part.
- Can be combined with another sanitizer flag as ubsan, asan, msan, even lsan ...
- Once the binary built, has plethora options as parallel jobs, enabling/disabling signal interceptions, memory usage limit, ... some options are dependent to other sanitizers for example detecting memory leaks.

We said options ?

- A fuzzed library run each time the whole process, we can limit the number of runs. Limits the max length of the incoming inputs.
- Can be ran with parallel jobs.
- Enable/disable certain signal interruption.
- Limit the memory usage.
- Control the degree of mutation.

Culprits of the fuzzing ...

- There are certain type of softwares more difficult to fuzz.
- For instance, softwares listening on a socket !
- Example with h2o which is a web server library and uses libFuzzer for its tests.
- Indeed, in this case there is a need to create a whole handcrafted configuration, creating a client request from this fuzzed data. Fortunately, h2o is mainly a shared library, things get complicated if it s a monolithic binary.

Xray Instrumentation

Is a run time call tracing facility. Mainly made for function timing measurements.

Can be refined by explicitly tracing or not tracing certain functions via clang attributes, configuration files or at least by function thresholds.

Can be enabled/disabled at runtime.

When disabled, the performance overhead is usually non existent but has a more noticeable performance difference when enabled but somehow suited to be ran in production.

But usually only to be run for a certain time and for a subset of functions in order to collect enough data dependent also on the function threshold and the memory usage limit wish for the in memory buffer data collection.

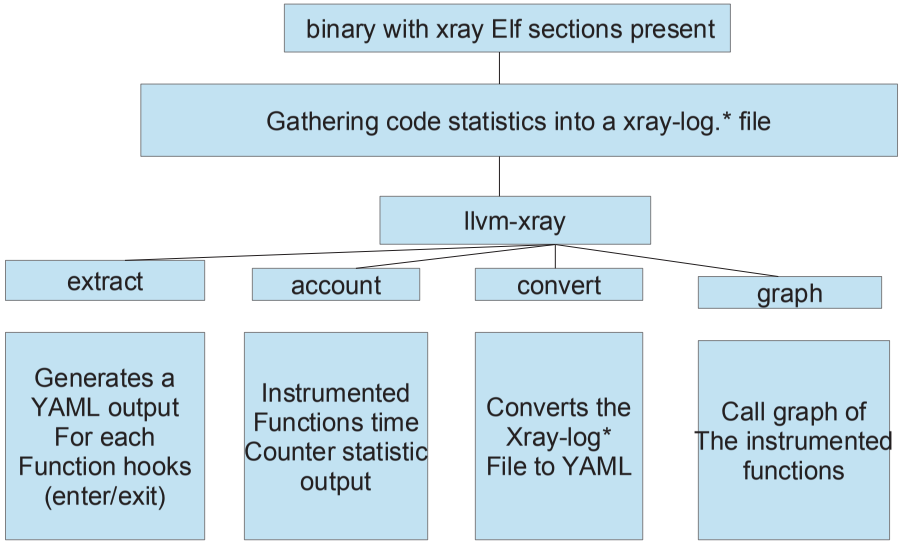
How does it work ?

- Xray injects instrumentation hooks at function entry and exits.
- Empty hooks until xray is enabled so as runtime thus replaced by cycle counter, function identifier, thread id, base address metadata.
- The basic flag is -fxray-instrument
- Our binaries contains now xray_instr_map and xray_fn_idx sections within data segments.
- We need means to extract those data from the Elf binary to generate our call graphs => llvm-xray.
- Accounting is also a feature to display where the code spends most of the time.
- There is logging options settable via XRAY_OPTIONS.

Accounting is not what you think !

- But Is more about to display the code used and the cumulative time spent for each.
- If it is a multithread program, data can possibly be aggregated.
- Can be sorted by any column, formatted as csv
- Can give a good idea of the possible bottlenecks.

Xray workflow



Monitoring FreeBSD Systems
What to (Not) Monitor

Andrew Fengler
ScaleEngine Inc.
andrew.fengler@scaleengine.com

Abstract

Operators of computer systems need to be aware of the state of their systems. As systems and networks become more intricate, the need for this information increases. This increase in complexity also leads to an increase in failure modes, often creating modes unique to the environment.

This means that any monitoring setup must be as unique as the environment it operates on if it is going to be of use. As a result, it is frequently not possible to simply run an off-the-shelf solution, and an understanding of the principles behind monitoring systems must be applied to the implementation of your solution.

This paper will cover many of the basic areas for monitoring, and how they can be applied to FreeBSD systems.

1 Introduction

Monitoring of large numbers of systems is a complex balance between being sensitive to abnormal conditions that indicate problems, and filtering out false positives. Modern operating systems offer a large number of metrics to monitor, but not all are useful, and many of the things that are useful to monitor are not immediately apparent, or are difficult to find, especially for someone new to monitoring servers. The same error can have a different meaning or cause depending on other factors on the system, and without proper monitoring, it will be difficult to track down the source.

1.1 Reasoning for Monitoring

It is important to understand why we are interested in monitoring a computer system before we implement it. A computer or the FreeBSD operating system is a very intricate thing. To monitor all aspects of the system would be an incredible undertaking. So we need to reduce what we monitor down to an achieveable quantity.

The reason for running a computer at all is because they perform useful work. Parts of the computer or operating system that is needed to perform that work are important and need to be monitored. Conversely, any part that is not needed for this work is unimportant and can be ignored. Attempting to collect and monitor unimportant information simply increases the difficulty of determining whether the system is working as intended.

1.2 An Abridged History of Monitoring Technologies

In the beginning, all observation of a computer's state was done manually by a human operator. Whether this was lights on a status panel, or messages printed to `STDERR`, this would go directly to a human's attention. Daemons that are not run interactively log any problems to `syslog`, and any log entries with a high enough error level would be shown to the operator.

This methodology has persisted in many ways. Although on most servers `TTY1` is is not watched by anyone, all log messages of level `err`[1] or higher are printed there by default in FreeBSD. Many daemons will log errors, or print them to `STDERR`, and promptly forget about them, making the error impossible to spot unless you were filtering through the log file. Although logging of errors is useful for diagnosing an issue and determining the cause, is is significantly less useful for detecting the problem in the first place.

Technologies such as `SNMP`[2] solve some of this problem by allowing information on the system's state to be procedurally retrieved by a remote system. Unfortunately this is limited to information about the operating system, as few programs have support for providing data through `SNMP`.

Many programs now offer some form of status output that can be externally retrieved. BIND dumps statistics to a file, Nginx has a status page, and Varnish offers statistics through `varnishstat`. Although this allows each program to offer information tailored to how the program operates, this has the limitation that each one is unique, and any monitoring

[1] LOG_ERR is the 4th highest level in syslog. LOG_EMERG, the highest, is normally broadcast to all users terminals

[2] SNMP - Simple Network Management Protocol

must be custom-built for the combination of your monitoring system and the program being monitored.

2 Availability

The simplest form of monitoring is availability checks; i.e. "does it work?". These tests ensure that at some higher level, the system works as intended. This could be as simple as sending a `ping` and seeing if the system responds. If the system answers the ping, then we can infer many things: the system is running, it has power, it has a working network connection, and so on. There are advantages and disadvantages to high-level tests. They simplify the monitoring by not requiring a separate check at each level, but if the check fails, the reason why might take some digging to solve.

One check of this sort is to use `netcat` to check if a service is listening on a given port. Figure 1 shows a simple check to see if a webserver is listening on port 80 on IPv4. This can be useful for testing services that don't have an easy way to do a functionality check, but this way you can at least tell that something is listening.

Often, a program can listen on multiple ports or addresses. A typical web server will be listening on ports 80 and 443, on both IPv4 and IPv6. Doing 4 seperate checks of the web-server's status page would be redundant and incur pointless overheat. If you only are checking one port, you might not notice a problem on the other ports, such as the server failing to listen on IPv6, but otherwise working normally on IPv4. A simple TCP connection check is enough to catch that failure case.

Other examples of availability checks include loading a web page, logging in with SSH, and sending an email to a monitored mailbox.

3 System Resources

A computer offers resources, and services running on the computer will use some of those resources. Ensuring that those resources are available is one of the main tasks for monitoring at the operating system level. Some of these resources include CPU time, memory, disk space, and network bandwidth.

3.1 CPU

SNMP offers 2 ways to get your CPU usage. One is to use the `UCD-SNMP-MIB::ssCpuRaw*` values to get a counter of the CPU time spent and average it on your monitoring interval. The other is to use the `UCD-SNMP-MIB::ssCPU*` values which are an integer, pre averaged over 1 minute, divided down by the number of processors, and rebased as a percentage. See figure 2.

These 2 collection methods represent 2 different types of measurement: the rolling average, and the snapshot. A rolling average shows an average usage over a given window, in this case the window is the measurement interval. Rolling averages will account for all usage in that window, but as a side effect will smooth out spikes shorter than the window size. A snapshot shows the exact level at the time of the measurement. This shows a more exact level, but only at the time of the measurement. Any spikes outside of the time of the measurement will be missed.[3]

Neither is necessarily better, both have their strengths and weaknesses. If you have a cron job running every 5 minutes that is using all available CPU for 30 seconds, and you check CPU usage every 5 minutes, a rolling average will show only 10% usage, and a snapshot might show either 100%, or 0%.

3.2 Load Average

Although it seems like the easiest thing to check, load average has a number of issues. It fails to account for multiple cores, and the distinction between cores and hyperthreads is lost on it. It varies wildly between workloads: our monitoring servers will show loadavg of 16 on a 4 core machine, with only 30% CPU usage, and video server with its CPU fully used might only show a load average of ˜8 on an 8 core machine.

3.3 Memory

We need to ensure that there is free memory available for programs that need it. However not all memory usage is equal.

[3] While the `UCD-SNMP-MIB::ssCPU*` values are strictly a rolling average over a window of 1 minute, for checks with an interval > 1 minute it is functionally a snapshot.

```
$ nc -z -4 host 80
Connection to host 80 port [tcp/http] succeeded!
```

Figure 1: A simple check with netcat to see if something is listening on port 80

```
$ snmpget -c public -v 2c server.example.com UCD-SNMP-MIB::ssCpuIdle.0
UCD-SNMP-MIB::ssCpuIdle.0 = INTEGER: 83
$ snmpget -c public -v 2c server.example.com UCD-SNMP-MIB::ssCpuRawIdle.0
UCD-SNMP-MIB::ssCpuRawIdle.0 = Counter32: 1653347551
```

Figure 2: Getting CPU data out of SNMP

FreeBSD will put unused memory to use, by using it for cache and for the ZFS ARC.[4] ARC and cache will frequently consume most free memory, but the system will free up memory being used for cache when needed. This creates a measurement problem, as memory being used for ARC is counted as wired. You can see an example of this in figure 3. To get useful numbers, you have to count the used memory minus the ARC, count the ARC and cache as memory that can become available, and the memory that is truly free.

All of the statistics from the arc and regular memory usage can be found in `sysctl: kstat.zfs.misc.arcstats`, `vm.stats`, and `vm.stats.vm.v_page_size`.[5] You can even get a count of how many times the ARC has been throttled due to memory pressure from `kstat.zfs.misc.arcstats.memory_throttle_count`

3.4 Network

Because packets are routed from network provider to network provider on their route, a problem in the middle of the route can cause connection issues. Frequently, these problems will cause packets to be lost, or to be routed on sub-optimal connections and cause the packets to take longer from source to destination. We can use ICMP between remote systems to detect loss and high latency along the route.

The status of the network connection itself deserves checking. Ethernet link speeds are typically autonegotiated, and negotiations can sometimes return a less than optimal result. Some service providers will throttle your connection speed if you're over your usage quota. In any case, it's good to know if the connection is operating at the speed we want it to be. FreeBSD's `ifconfig` will show you the media settings for an interface, which includes the speed:

```
$ ifconfig | grep media
media: Ethernet autoselect (1000baseT )
```

[4] ARC: Adaptive Replacement Cache

[5] Note that ARC stats are in bytes, and memory stats are in pages, so you need to multiply memory stats by page size to get an even comparison.

3.5 Bandwidth

Checking the network utilization is pretty straightforward with `SNMP`. `Net-SNMP` implements the IF-MIB::ifXTable, which shows statistics for each interface. We can get the number of octets[6] sent and received on the second interface in the table, as in figure 4. Usage is stored as a counter of total octets sent since system boot. If we store the value and subtract the old value from the new one, we get a delta of the total bytes sent over that time interval.

Note that we are using `ifXTable` which uses 64 bit counters. `SNMP` also has `ifTable`, which only uses 32 bit counters. A 32 bit counter of bytes will roll over at ~4 GB, which can be sent in less than 5 minutes on a 10 Gb/s interface.

For those who rent their servers, the total data transfered can also be a limited resource. Many providers limit the amount of data you can send and receive, usually some number of terrabytes per month. Since the method we've discussed for monitoring network usage measures in deltas of bytes, if we store those deltas, we can tally them up and measure usage over the interval of interest. ScaleEngine uses a program called `RTG` [1] for this, which logs the deltas to a MySQL database, where we get the data for usage information.

3.6 Disk Space

Free space on disk is important if anything needs to write to the disk. Most filesystems, ZFS included, will perform poorly if they do not have some free space.

A common way to watch your disk space is to use `SNMP` to check free space on `/`. That will not work as expected with ZFS. `Net-SNMP`'s disk checks are not ZFS aware, and see each dataset as a partition, and any data stored on a different "partition" will cause the root partition to shrink, rather than show more space used. A stock FreeBSD installation will only have ~5 GB on the root dataset, meaning the free space on the / "partition" won't drop below 50% free space until there is only 5 GB free for the root dataset. Since most programs will not be writing to the root dataset, this will only ensure that there is free space on that one dataset. If you set a reservation on the

[6] octets = bytes

```
Mem: 9588K Active, 103M Inact, 2786M Wired, 4992K Cache, 55M Free
ARC: 1935M Total, 603M MFU, 1202M MRU, 560K Anon, 19M Header, 111M Other
```

Figure 3: Memory usage from `top`

```
$ snmpget -c public -v 2c server IF-MIB::ifHCOutOctets.2
IF-MIB::ifHCOutOctets.2 = Counter64: 26790537050371
$ snmpget -c public -v 2c server IF-MIB::ifHCInOctets.2
IF-MIB::ifHCInOctets.2 = Counter64: 17901892810225
```

Figure 4: Getting counters for the second NIC

root dataset, that will be seen as free space, even though other datasets that do not have reservations will be out of space.

It's better to use ZFS to check ZFS. ZFS has some useful tools, and also has good output handling for automatic parsing:

```
$ zfs list -pH -o name,used,avail
mjolnir 46128820224 67251605504
mjolnir/ROOT 12187820032 67251605504
mjolnir/ROOT/default 12187729920 67251605504
mjolnir/tmp 157851648 67251605504
mjolnir/usr 29228777472 67251605504
mjolnir/usr/home 27944427520 67251605504
```

ZFS offers a number of advantages over `SNMP` for disk space checks, making it worth the exta labour to implement the checks. ZFS has per-dataset granularity, quota information is readily available, and jail usage is easy to break down. If you're only able to work with `SNMP`, then make sure you're checking on all datasets you care about, not just `/`.

3.7 Disk Health

Disks are consumables, whether they are hard disks or solid state drives. Staying on top of the health of the drives will allow you to have some chance of anticipating a failure. We can get the disk information using `smartctl`.

A study by Google [2] found that the only SMART errors that correllate to drive failure is reallocated sectors, weakly correlating for online reallocations, and more strongly for offline reallocations. SMART value 5 is online reallocation, and value 198 is offline reallocations. Value 197 is the same as 198 for most manufacturers, Toshiba being the only exception we've encountered. We can monitor these 3 values along with the overall health check using `smartctl`. Figure 5 shows an example of these values.

Also of interest is the SSD wear values, but these vary between manufacturers and we have yet to find a way to measure them that gives usable data.

SMART will not predict all failures, and there are failures external to the drive that can make it unusable. This could be a cable error, data corruption, or some other error that SMART doesn't catch. Checking zpool health is pretty simple: To see if the drive is in a working state, we can check if ZFS sees the drive as usable or not.

```
$ zpool list -o name,health
NAME     HEALTH
mjolnir  ONLINE
```

The health column shows the state of the pool, `ONLINE` is ok, other states indicate a problem.[7] If you want more details, you can also parse through `zpool status` to find the exact drive that has the problems.

4 Precursor Metrics

In addition to system resources, there are many things that while they do not directly affect the system's operation, they can cause problems in some situations.

The temperature of the system can be monitored to watch for overheating. FreeBSD does not make it obvious when thermal throttling kicks in, other than some entries in `syslog`. You can get the CPU temperatures out of `sysctl`.

```
dev.cpu.#.temperature
```

NTP is important for keeping your clock accurate. If you drift out of sync, this will cause problems with anything time-sensitive, including a lot of cryptography. You can check your offset against an NTP server with `ntpdate`. Use a server that is different than the one you're syncronizing against in order to catch problems with that server.

```
$ ntpdate -q 0.pool.ntp.org
... offset -0.007518, ...
```

While it might not seem useful, monitoring the uptime of a server can be a good way to catch unexpected reboots. Check if the uptime is less than twice the alert interval to have an alert whenever a server reboots. The advantage to using uptime rather than a cron entry for `@reboot` is that checking the uptime with `SNMP` works on switches, allowing you to catch reboots that might just apear as a brief period of packet loss otherwise.[8]

5 Jails

Jails offer a lot of challenges in obtaining meaningful statistics, especially if you care about isolating to the jail. CPU

[7] A state of OFFLINE indicates the device was taken offline. While this is not an error, it is probably not a desired state.

[8] Switches can reboot really fast, and if it happens between checks there might not be anything else to give it away

```
SMART overall-health self-assessment test result: PASSED
  5 Retired_Block_Count     0x0033   100   100   003    Pre-fail  Always       -       0
197 Current_Pending_Sector  0x0022   100   100   000    Old_age   Always       -       0
198 Offline_Uncorrectable   0x0008   100   100   000    Old_age   Offline      -       0
```

Figure 5: SMART values of interest

usage information will include usage from processes outside the jail. Isolated network metrics are only possible if you are using vnet, which only became the default in FreeBSD 12. If you want to run `SNMP` in the jail, you have to build `Net-SNMP` with special options so it works at all. Since much of the resource usage information will only be available on the host, make sure you can corellate services in jails to the host they are running on to be able to examine the state of resource availability for that jail.

6 Different Types of Measurements

Regardless of the source of the measurement, the data that can be retrieved will fit into 2 broad categories.

State measurements are of something that fits into a limited set of states. This could be a service that is up or down, whether a connection succeeds or fails, or an HTTP status code. Each state is typically tied to some stable meaning: a degraded zpool is a degraded zpool, whether it's on a supercomputer or your Raspberry Pi.

Metrics are a measurement in some range of possible values. CPU usage, ICMP RTA[9] times, and a count of logged in users are all examples of this. The administrator's judgement is required to establish thresholds for mapping values in this range to a stable interpretation of the state it represents. These thresholds will frequently vary from system to system. 600 Mb/s of network traffic might be unremarkable on a router, but could be worrisome on a server that's supposed to only be serving DNS. Tools such as graphs are useful to track historical values, as the definition of typical may change over time and as the system itself grows.

7 Conclusions

While the information shown here is by no means exhaustive, this paper covers a selection of key methods and services for FreeBSD systems in a server environment. Each environment is unique, and this paper was based off of only one. However, the basic principles will remain the same, and the techniques here can be applied to far more than what the examples hold.

Availability

Most of the monitoring described here has been implemented by the author, and is available on ScaleEngine's Github: https://github.com/scaleengine/se-nagios-plugins

Acknowledgments

Thanks to every sysadmin who solved a problem they faced, or wrote some script, and provided their work to the world. The author has been saved many wheel reinventions by simple blog posts.

Thanks to Allan Jude, for humoring my many experiments that never panned out.

[9] Round trip average, or the average time it takes a packet to travel to the other system and back

References

[1] Robert Beverly. Rtg. http://rtg.sourceforge.net/.

[2] Eduardo Pinheiro, Wolf-Deitrich Weber, and Luiz Andre Barroso. Failure trends in a large disk drive population. In *USENIX Conference on File and Storage Technologies (FAST'07)*, 2007. https://research.google.com/archive/disk_failures.pdf.

Intel HAXM – a hardware-assisted acceleration engine in the NetBSD kernel

AsiaBSDCon 2019
Tokyo, Japan

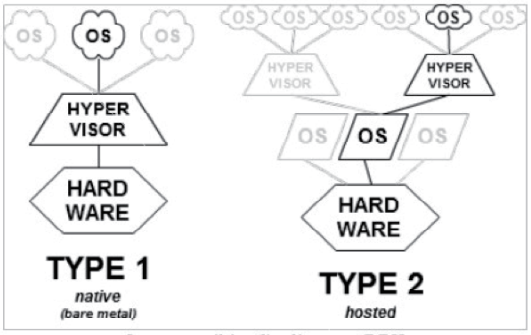
Kamil Rytarowski
The NetBSD Foundation
kamil@NetBSD.org

Abstract

Intel HAXM (Hardware Accelerated Execution Manager) is a hypervisor that works as a loadable kernel module in multiple kernel environments. HAXM uses the Intel Virtualization Technology (VTx) and thus requires an Intel hardware architecture. A hypervisor (on a so called host machine) is a piece of software or hardware (sometimes both) that can create and run a virtual machine (called a guest machine). Usage of virtual machines reduces the need for using each software or product on dedicated hardware and can fully isolate it from the environment, which makes it suitable for data migration, reduction of costs of running multiple instances of guest machines on a host machine.

1. Introduction

There are two types of hypervisors: Type 1 and Type 2. The first type requires booting directly into a hypervisor that is a standalone program and it can be utilized to spawn guest machines, while one of them typically is a master guest that can orchestrate the hypervisor. The second type allows to boot into the default kernel of the host and the hypervisor runs as a kernel module in kernel space, creating dedicated guest-machines in a privileged kernel-space on demand.



[source: wikipedia; license: CC0]

2. Xen

The NetBSD Operating System traditionally uses Xen virtualization (NetBSD/xen). Xen is a Virtual Machine Monitor that was integrated with the distribution sources in 2004. The Xen technology is supported by NetBSD/i386 (x86 32-bit) and NetBSD/amd64 (x86_64). Over the years the set of features and hardware facilities have evolved and are still maintained by the NetBSD developers, although there is no full feature parity with other kernels.

A shortcoming of Xen/NetBSD, besides the lacking parity in features is that it's not a convenient solution for typical desktop usage. There are reported conflicts with the X Window system, which is nowadays a mandatory graphical system on UNIX systems.

Another property that makes the usage of XEN more difficult is the need to redefine the booting process and directly start a standalone hypervisor that manages all guests. This is a Type 1 hypervisor. On a desktop computer a user usually prefers by far to enable or disable features without the need to reboot and especially without the need to reinstall the Operating System.

3. Type 2 (hosted) virtualization

Nowadays desktop setups tend to use Type 2 virtualization more often which is less intrusive for the host system, however there wasn't any available solution so far that would be supported by NetBSD. A popular front-end to hypervisors of this type is Qemu – a GPLv2 software maintained by the Open-Source community. The primary purpose of Qemu is to utilize software-assisted emulation (softemu) or hardware-assisted emulation. The software-assisted emulation is required for emulating a different CPU than the underlying host CPU, however for the use cases of emulating the same type of architecture there is room for hardware-assisted emulation that can offer a massive boost in speed of code execution inside guest machines.

4. Qemu with hardware-acceleration

There is a number of hypervisor solutions for mainstream Operating Systems that work as plugins for qemu. The most important ones are:

- KVM for Linux,
- HVF for Darwin,
- WHPX for Windows.

Until the process of making the HAXM project available to the Open-Source community there wasn’t a single solution for the mentioned systems, basically due to the fact that each of these three systems weren’t from the same kernel family and each requires specific knowledge. This also resulted in the solution of each of the mentioned systems that were closely tied to each of the specific kernels and hardly reusable by someone else.

5. HAXM

Intel HAXM (Hardware Accelerated Execution Manager) is a hypervisor that works as a loadable kernel module in multiple kernel environments. HAXM uses the Intel Virtualization Technology (VTx) and thus requires an Intel hardware architecture.

Until the process of making the HAXM project available to the Open-Source community there wasn’t a single solution for the mentioned systems, basically due to the fact that each of these three systems weren’t from the same kernel family and each requires specific knowledge. This also resulted in the solution of each of the mentioned systems that were closely tied to each of the specific kernels and hardly reusable by someone else.

I decided to give a HAXM port to NetBSD a try as it was already available for two kernel families: Windows and Darwin, which is rather an unusual pair of supported kernels by an open-source project. The process of porting HAXM to NetBSD however was still difficult as semantics of internals of these systems were alien to NetBSD internals. This has suspended the porting process for a while, until the moment when Linux was supported as host, which exposed internals that are well documented in the OpenSource resources and much closer to the model of the NetBSD kernel, and thus more easily mappable into native code. A HAXM port to the NetBSD kernel has been ported, which involves the kernel workaround patch and qemu patching. The final deliverable is that the HAXM engine is successfully emulating NetBSD, Linux and Windows as guests, while support for other systems is either functional or close to be so.

With the advent of Intel's open-sourcing of their HAXM engine, we now have access to an important set of features:

- A BSD-style license.
- Support for multiple platforms: Windows, Darwin, Linux, and now NetBSD .
- HAXM is an Intel hardware assisted virtualization for their CPUs (VTx and EPT needed).
- Support for an arbitrary number of concurrent VMs. For simplicity's sake, NetBSD only supports 8, whereas Windows/Darwin/Linux support 64.
- An arbitrary number of supported VCPUS per VM. All OSs support up to 16 VCPUs.
- ioctl(2) based API (/dev/HAX, /dev/haxm_vm/vmXX, /dev/haxm_vmXX/vcpuYY).
- Implement non-intrusively as an out-of-tree, standalone executable kernel module.
- Default compatibility with qemu as a front-end.
- Active upstream support from Intel, which is driven by commercial needs.
- Optimized for desktop scenarios.
- Probably the only open-source cross-OS virtualization engine.
- An active and passionate community that is dedicated to improving it.

As well as a few of HAXM's downsides:

- No AMD (SVM) support (although there are community plans to implement it).
- No support for non-x86 architectures.
- Need for a relatively recent Intel CPU (EPT required).
- Not as flexible as KVM-like solutions for embedded use-cases or servers.
- Not as quick as KVM (probably 80% as fast as KVM).

The NetBSD support has been upstreamed directly to the Intel HAXM repository at GitHub and packaged in pkgsrc.

6. Supported guest OSs

HAXM on NetBSD has been verified to successfully boot a number of guest Operating Systems, such as NetBSD/amd64 and (in alphabetical order):

- DarwinX86
- DragonflyBSD
- FREEDOS
- FreeBSD/i386
- Haiku
- KolibriOS
- Linux/amd64 (noapic)
- Minix3
- OpenBSD/amd64
- Plan9
- ToaruOS
- Windows 7 32-bit

A selection of OSs is still known to be not handled appropriately. An important target is NetBSD/i386 and (in alphabetical order):

- Android X86
- FreeBSD/amd64
- Icaros (AROS)
- OpenBSD/i386
- QNX
- ReactOS
- Solaris
- Windows in other versions

There are still issues specific to the NetBSD host, especially APIC and RDTSC calibration bugs and generic ones such as handling Guest CPU Registers and missing instructions in the embedded instruction emulator (needed for MMIO).

7. Conclusions

HAXM has been verified to be a portable cross-platform loadable kernel module in the NetBSD kernel context. Utility of the software has been verified with a number of Guest Operating Systems.

8. Acknowledgments

The HAXM community for helping to address generic bugs and openness for new platform support.

The NetBSD community for thorough testing and feedback in the porting process.

9. References

[1] The Hardware-Assisted Virtualization Challenge
http://blog.netbsd.org/tnf/entry/the_hardware_assisted_virtualization_challenge

[2] HAXM in pkgsrc
http://mail-index.netbsd.org/netbsd-users/2019/02/13/msg022207.html

[3] HAXM official repository
https://github.com/intel/haxm

Managing System Images with ZFS

Allan Jude, Klara Inc
allan@klarasystems.com

Abstract

The author describes existing procedures, tools, and ongoing development to improve the process of updating appliances, remote systems, and individual computers using ZFS. This paper describes a mechanism for replacing the operating system image with a newer image in a safe and atomic fashion. This system allows for fail-safe unattended upgrades of remote appliances and machines with a built in automatic recovery mechanism in the event of failure. Current and planned enhancements to 'poudriere image' are described as well as improvements to support tools including bectl and zfsbootcfg.

1. Motivation

FreeBSD is a popular choice for building appliances because of its liberal license and composable base system, but each vendor is tasked with building their own upgrading mechanism for both minor security updates, and major OS upgrades. FreeBSD would benefit from a standardized upgrading mechanism that is adaptable to each vendors requirements. We examined the existing updating frameworks in FreeBSD and then set out to build a better mousetrap. Leveraging experience gained while managing 100s of remote servers we describe how upgrades can be handled in an automatic, yet safe fashion, without merging configuration files or requiring manual intervention.

2. Prior Art

The problems of building system images and keeping systems up to date are not new. FreeBSD has evolved a number of systems over time to try to address these issues, and many of them worked quite well for a time. However, each has developed shortcomings as the state-of-the-art has advanced.

2.1. NanoBSD

NanoBSD is a build system first introduced to FreeBSD in 2004 by Poul-Henning Kamp[1] for the purpose of making it easier to generate compact-flash disk images for embedded systems running FreeBSD. The resulting image would consist of three partitions, the 3rd held persistent configuration data, and was usually quite small, and the remaining space was divided evenly to form two system image partitions. The current operating system would be written to both of these partitions, and in the future when it came time to upgrade, the inactive partition would be overwritten with a newer image, and a flag set to boot from that partition, one time only. If that boot succeeded, then the upgraded partition would become the new default. If boot failed, power cycling the device would automatically revert to booting from the image that had been running before the attempted upgrade. Upgrades could continue like this, alternating back and forth between the two partitions, always leaving the system with a known good image to fall back to. The filesystems were also mounted read-only, so power loss or other issues could not corrupt the filesystem or require a fsck(8) at boot. Configuration data was copied from the dedicated configuration partition to a memory-backed filesystem at boot, and optionally saved off to the dedicated partition when necessary.

NanoBSD allowed easy customization of FreeBSD, including and excluding features as needed to make a functional but minimum system image. NanoBSD was adopted by many FreeBSD based appliances, including pfSense and FreeNAS.

NanoBSD is limited to UFS, and both FreeNAS and pfSense have since switched to ZFS boot environment based solutions instead.

2.2. freebsd-update

Binary updates for FreeBSD were first introduced in 2001 as experimental accompaniments to some security advisories in the summer of 2001 by Colin Percival[2]. In this timeframe, the freebsd-update client was available via the ports tree.

Version 2.0 of freebsd-update was merged into the FreeBSD base system in 2006, and the building of the updates was handed off to the FreeBSD Security Officer, rather than Colin personally building the updates.

freebsd-update binary updates are created by comparing the build output of the unmodified source code (build 1), to the same source code built with the clock changed backwards by 400 days (build 2), to detect changes in files that are related to “build stamps”, rather than changes to the code. The offsets that contain “build stamps” are identified and recorded. Then a 3rd build is done of the patched source code, and it is compared to the original release binaries, excluding the offsets of the “build stamps”. Any files that still differ have

been modified by the patches to the source code, and need to be distributed as part of the binary update. Lastly, a 4th build is done, again on the patched source code, to locate the new offsets of the "build stamps", as changes to the source code are likely to have moved the build stamps. In some cases, the first of the four builds can be replaced with downloading the original -RELEASE ISO instead.

PC-BSD previously attempted to use freebsd-update to provide updates to its users, but found the freebsd-update-server to be too slow (requiring 3 or 4 full buildworlds) and fragile. They also found that freebsd-update did not support updating stable branches or head, only releases. Users also complained about the way merging of configuration files was handled (manually, labour intensive), especially the fact that freebsd-update does not ignore changes to VCS ID lines, and the chance of leaving merge markers in configuration errors by mistake is too high.

Currently freebsd-update is x86 only (i386 and amd64) and still requires a lot of manual intervention by secteam when trying to release security advisories. Support for other architectures is likely quite feasible, but the read/write access pattern of the freebsd-update client is not very conducive to SD cards and other low speed, low endurance flash.

Recent work on reproducible builds should reduce the number of "build stamps" that exist in the official releases of FreeBSD, but this is unlikely to be able to eliminate an entire build cycle due to the way freebsd-update is structured.

2.3. pkg base

The pkg(8) package manager has been used, since 2014, to install, manage, and update third party software, and is one of the most familiar aspects of the FreeBSD Operating System for end users. Naturally, this argues for using pkg to distribute and update the base system itself. There have been discussions and some development along these lines, but a complete solution has not yet emerged.

Currently development seems stalled and there is a lack of an overall design for what a packaged base system will look like. Trying to balance flexibility, and allowing a user to opt out of individual pieces of the base system has resulted in the system being split into many 100s of packages. This will make updates smaller, but makes dependency tracking more difficult and results in a very verbose package listing. However, pkg(8) has better automated 3-way merge handling for configuration files. Conflicts are left for the user to deal with later, leaving the original file in place, and installing the new version with the suffix ".pkgnew".

PC-BSD/TrueOS attempted to use the pkg-base system as it existed, but found a number of limitations, especially around upgrades. TrueOS is currently starting fresh with trying to have the base packages built via the ports tree, rather than the main OS build infrastructure, with a target of approximately 10 packages, rather than the current 800 or so packages.

3. ZFS

ZFS is an advanced filesystem that combines the role of volume manager and filesystem together to make managing storage easier for the administrator. ZFS is a CoW (copy-on-write) filesystem, so blocks are not overwritten in place, but written to a new location, and then the old location becomes free again later. ZFS is transactional like a database, so each group of writes (transaction) either fully completes, or is rolled back. This allows ZFS to move from consistent state to consistent state, without ever requiring intervention from tools like fsck. So, if the system crashes or unexpectedly loses power, the filesystem does not require any recovery steps or consistency checks. It finds the more recently completed transactions and mounts the filesystem from that point. This obviates the need to have the filesystem be read-only, as it is in NanoBSD, to avoid being inconsistent.

A traditional filesystem can only operate upon a single disk, so volume managers were created that would allow multiple disks to be combined into a single logical disk that could be presented to the filesystem. The volume managers also grew features like parity and redundancy (RAID), to protect the filesystem from the failure of one of the disks that makes up the volume. By combining these roles into a single system, the ZFS filesystem has direct knowledge of the fact that the filesystem is backed by multiple disks, allowing for dynamic stripe sizes and other optimizations.

The biggest advantage to this approach, is that ZFS filesystems each share all of the free space from the "pool" of available storage. Whereas in a traditional filesystem, the volume must be partitioned into fixed-size chunks at the time of filesystem creation. While most filesystems support growing, this requires there be contiguous free space available at the end of the existing filesystem. In addition to the inflexibility of this arrangement, it can lead to the available free space being fragmented across multiple partitions, where the total amount of free space is sufficient for an upcoming task, but no individual partition has enough free space to meet the demand. With ZFS, as files are written to one of the filesystems, the free space is taken from the pool and allocated to that filesystem. Each filesystem does not have a fixed size, but instead can take available space from the pool as needed, in a thin-provisioned manor. It is also

possible to reserve space for a specific filesystem, or limit a filesystem with a quota.

4. ZFS Boot Environments

With ZFS, creating additional filesystems a low cost operation. This allows the NanoBSD concept to be taken much further. Instead of two fixed sized partitions for system images, ZFS allows a number of system images limited only by the available space in the pool. Additionally, the copy-on-write nature of ZFS can used to share unchanged blocks between system images to save space.

The concept of ZFS boot environments, originally from Solaris (as is ZFS), is having multiple root filesystems that the operator can choose from at boot time, using the loader menu. A snapshot of the working system can be taken at any time, and then cloned to create a writable filesystem, with the contents of the system image as it existed at the time of the snapshot. This allows the operator to easily revert to a previously working image, without the storage cost of an entire system image. Additionally, instead a single dedicated configuration partition, every filesystem other than the root filesystem is retained as the boot environment is changed. User data, such as home directories, log files, configuration, databases, etc are all retained as long as they live in a filesystem other than the root. The system image can be build such that packages (/usr/local) are included in the system image (so a failed package upgrade can be reverted), or be kept in a separate filesystem, so they are not impacted by changes to the system image.

To provide an equivalent to the fail-safe upgrade mechanism that NanoBSD has using GPT partition flags, FreeBSD has the zfsbootcfg utility. This writes the name of the selected boot environment into the on-disk filesystem label, where boot1 (gptzfsboot) reads it, and then overwrites it with zero bytes. It then sets it as an environment variable when invoking the loader. The loader then uses it to load the kernel from that filesystem, and pass that filesystem as the root to the kernel when it boots. This provides the same “next boot, and next boot only” use this newly upgraded filesystem instead of the default. If the boot is successful, the default can be changed and the upgraded system image will now boot by default. If not, power cycling the device will revert to the previous system image, and the upgrade can be retried, or debugged.

ZFS features a replication protocol, that can serialize a filesystem or hierarchy of filesystems into a stream that can be stored for later recreation of the filesystem. This protocol also includes support for incremental replication between snapshots of a filesystem or series of filesystems. Using this system it is possible to incrementally update a clone of a system image, reducing the size of the binary updates. In 2018 the replication protocol was enhanced to take advantage of ZFS's transparent compression features. If data is compressed on-disk in the filesystem, it can optionally remain compressed during replication, further reducing the size of the incremental system image updates.

5. Building System Images

How does one build a clean system image using ZFS? The FreeBSD package building framework, poudriere, includes a system image building feature.

5.1 Poudriere

In the previous generation of packaging on FreeBSD, before the pkg tool, packages were built in a distributed fashion, using a number of machines, but it was often fragile, and duplicated a lot of effort. As work was distributed, it often came to pass that the same common dependencies were build by each worker, and the effort is coordinating the distributed system, and maintaining the control software was to onerous.

Thus, poudriere (powderkeg in french) was invented. Rather than trying to build one package at a time, as quickly as possible, using all of the CPUs, it instead creates a number of jails (by default equal to the number of CPU cores), and in each one builds a single package using only a single thread. This tends to make much more efficent use of the available CPU cores, since parts of the build that are inherently single threaded (configure scripts and the like) do not result in all other CPUs going idle. An additional advantage to using a jail for each builder, is that the jail can be reverted to a clean environment between each build, ensuring there is no contamination of the build environment. Before building each port, a clean copy of the operating system is setup, the binary packages of the dependencies are installed, and then the build is run. The build jail also has no access to the internet, ensuring that the build process cannot reach out to the internet and modify its behaviour. While not required, using ZFS for these build jails makes the cleanup process nearly instantaneous, as the filesystem is just reverted to a clean snapshot between each build.

The system image used for these build environments can be created by downloading the official releases of FreeBSD (no compiling required), or by building from source, optionally with an external patch applied before the build process. Poudriere supports both svn and git, as well as tar and copying an existing source tree. The former options include support for incremental updates (svn update, git pull). Existing

system images can also be updated with freebsd-update. Poudriere can also cross-build for different architectures, making it possible to produce system images and packages for arm, arm64, risc-v, etc, from a standard amd64 machine.

These basic system images must be created in order to build packages, so when the need arose for Gandi.net to build virtual machine images of FreeBSD for their public cloud, it was a logical extension to poudriere, and the “image” subcommand was born. The image command takes an existing poudriere jail (which can be compiled, or created from an official release and binary updates), excludes listed files you do not want or need, adds an overlay (your own custom files that are added on top of the system image), installs a list of packages, and then generates an image in the specified format. Supported formats include ISO (cd/dvd image), optionally using an memory filesystem which itself is optionally compressed, or the same for a USB stick image, a raw disk image which is suitable to be written directly to a disk or converted to various hypervisor image formats, an embedded or firmware image, a UFS dump, or a ZFS replication stream. The ZFS replication stream comes in two flavours, the entire pool (a compound stream containing all filesystems), or a single stream containing just the root filesystem (the boot environment).

The author added the support for ZFS replication as an output format for poudriere image, and is continuing to work on enhancements. Currently building incremental replication streams is still a work-in-progress. The existing ZFS support defaults to a pool layout identical to that created by bsdinstall, but is easily customizable. This controls what filesystems are created for the whole pool image, or what files are included in the root filesystem versus having their own filesystem in the boot environment mode. This can be used to control if packages are part of the system image, or if they are managed independently.

Distribution of a boot environment system image is just a matter of feeding the contents of the image file to the `zfs recv poolname/ROOT/environment_name` command. For a full pool image, create a pool if one does not exist, and then `zfs recv -F poolname`. Beware, this will irrevocably overwrite the pool with the contents of the new system image.

5.2 Customizing Images

In order to achieve our requirement to retain system configuration through upgrades, while avoiding the need to merge every change to /etc, we must make some small customizations to the system image as part of its creation. We create an additional filesystem, /cfg, to store the subset of files from /etc we wish to have been persistent. We then replace the versions in /etc with a symbolic link to /cfg. Our configuration moves the following files to /cfg: fstab (for swap configuration, ZFS does not use fstab by default), hostid (so the hostid persists), rc.conf.d (directory), rc.conf.local, resolv.conf, ssh (directory), sysctl.conf.local.

The remaining files in /etc are replaced with the latest version during each system image upgrade. Of course, since these files now reside on a separate filesystem, they will not be available during the boot process, until the additional filesystems are mounted. This is doubly true since fstab is one of the files we have relocated, so the OS won't have a list of other filesystems to mount. To address this, we abuse a little known feature of FreeBSD's init process. Designed to allow booting into a chroot environment, the loader.conf variable init_script runs a script to prepare the chroot, and init_chroot sets the directory to chroot the system into. We use the init_script feature to run a small script that finds the poolname/cfg filesystem and mounts it early in the boot process, before /etc/rc is run, so that our replacement configuration files will be readable.

6. Upgrading Systems

The process of upgrading a system is straightforward. First build the new system image, as a ZFS boot environment, then receive it under a new name to the pool on the target device. Use the zfsbootcfg utility to configure the system to boot into the new environment only once. Configuration such as hostname, network settings, SSH keys, will be retained via the /cfg filesystem. Attempt a reboot. As the system boots, it will erase the zfsbootcfg temporary configuration. If there are no problems, the system will now be running from the new environment. It is left as an exercise to the reader to develop a procedure to determine that the new system image is working as expected, and set the upgraded system image as the boot default. If this is not done, or the system fails to boot correctly, rebooting will use the previous system image.

Whether packages are included in the boot environment will depend on the type of deployment. In an appliance type configuration, it makes sense to bundle the packages into the system image, so updating the system image updates the packages as well. This also avoids any potential dependancy solving issues, as the new system image always contains a freshly installed set of packages. In the case of a server deployment, or an appliance where the set of packages may be customized, it may make sense to have /usr/local as its own filesystem and managing packages separately from the system image. Reducing what is included in the system image makes the incremental updates smaller and less risky. Deploying OS security updates does not need to involve changing the versions of the packages that are installed on the system. Excluding the packages from the system image allows the two to be updated separately. They could still both be

managed using the same ZFS update mechanism, just as separate filesystems.

In the case of a laptop or workstation, it makes sense to include the packages in the boot environment, so that a package upgrade can be undone if it causes issues. The advantage to boot environment is that rolling back to an older system image won't roll back the user data, so no work is lost even if rolling back to a weeks old system image.

7. Further Enhancements

There is still much to be done to improve this mechanism and provide a flexible but fail-safe upgrading mechanism. Creating a upgrading mechanism that is adaptable enough to serve the majority of use cases is challenging, but the result will ultimately be beneficial to the entire FreeBSD community.

7.1 Poudriere Image

Currently the layout of ZFS datasets in poudriere image is very basic, and defaults to the same configuration used by the FreeBSD installer. The author envisions a series of different templates for different use cases, and a more expressive configuration syntax that requires less ZFS expertise than the current system.

In addition to further enhancements to the overlay system to make it easier and more performant to include additional material in the system image, it is desirable to have support for a post-built chroot script to perform operations on the image. This is complicated in the cross-platform image case.

Poudriere image has a relatively wide selection of output formats, but the level of flexibility is still rather limited. Increasing the options available to the user without creating a combinatorial testing nightmare is a delicate balancing act. Even just the selection of disk layout and bootcode combinations: MBR, BSD, or GPT partition table, Legacy BIOS, UEFI, or dual mode, regular or advanced bootstrap code, encrypted with GELI or not, UFS (one or many partitions) or ZFS, etc quickly leads to over a 100 possible combinations to test, let along try to support. Then each of those can be packed as a raw disk image, CD/DVD ISO, vmdk, vhd, qcow2, Amazon EC2 image, and more. Finding what is more useful and most supportable and trying to limit creep will be challenging.

7.2 Bootstrap and Boot Code

Updating the bootstrap, the tiny bits of assembly that load the more complicated and feature rich loader, can be complicated and error prone. Worst of all, there is currently no fallback mechanism if this goes wrong. In the legacy (BIOS) boot case, the bootstrap is a complicated process (see “Booting from Encrypted Disks on FreeBSD”[3], proceedings of AsiaBSDCon 2016) consisting of multiple phases. While the boot0 and boot1 rarely change, boot1 is usually combined with boot2, so is updated when boot2 changes. With ZFS, changes to boot2 are often required when new features are introduced, as boot2 contains a minimal, read only implementation of ZFS that is used to read the boot loader from the ZFS root filesystem. This is also where the zfsbootcfg logic happens. There is currently no provision for recovering from a failed boot2. If the newly installed version doesn't boot, the zfsbootcfg logic may never happen, and its effects are only on the later stages of the boot anyway. In the common GPT case, boot1 and boot2 are combined and live in the partition with the “freebsd-boot” type. It may be possible to have have two “freebsd-boot” partitions (they are limited to 536 KB each) and teach the boot0 step (stand/i386/pmbr/pmbr.S) to use the GPT flags similar to NanoBSD to mark the freebsd-boot partition as failed, and on a successive boot, use the backup bootstrap code. Then have the boot failed flag removed by the loader when it runs successfully, assuming it can detect from which of the two freebsd-boot partitions it was spawned.

For the loader itself, the build world process keeps the previous version of the loader as /boot/loader.old, but relies on an operator interupting the boot2 phase and manually specifying that the alternative loader should be used. Luckily, the loader exists in the boot environment, so a failure here should be solved by zfsbootcfg's boot-once feature.

However, in the UEFI boot case, things are a bit different. Previously, the EFI System Partition (ESP) contained boot1.efi, which had just enough of a filesystem driver to read /boot/loader.efi from the target filesystem and hand off the boot process to it. However, the duplication of features and code complexity have resulted in this approach being replaced with placing loader.efi directly into the ESP. Since boot1 was going to have to be updated to support new ZFS features on a regular basis, it no longer made sense as originally envisioned, a rarely changing shim that would load the more featureful loader.efi. So now we may need to develop a fail-safe mechanism for updating the EFI loader. This can likely be implemented with the EFI Variables service provided by the system firmware as managed by FreeBSD's efibootmgr. The new loader would be installed to the ESP in a different location, and use the EFI nextboot feature to use the new loader only once. If this is successful, the regular loader can be replaced.

Additionally, there are plans to extend zfsbootcfg to have a more expressive configuration. Currently it writes a raw string to an reserved area of the ZFS on-disk label, containing the pool and dataset name to mount the root filesystem for. Another ZFS user, Delphix, has a similar system, except

theirs keeps only a counter, of the number of boot attempts. When a system manages to stay up for 10 minutes, it resets this counter to 0. If the boot attempts counter reaches 3, the system instead boots a rescue image that calls home for a technician to intervene. This came out of the constraints of the Amazon cloud environment, where there is no out-of-band console access to resolve boot issues. The author envisions a more structured data store (a packed nvlist) that could contain multiple parameters, combining the existing boot environment selection, a failure counter, a specific rescue environment, and even arbitrary environment variables to set to impact other parts of the boot process.

7.3 ZFS

There are a number of features that could be implemented in ZFS to make an upgrade system easier to implement and more powerful. The first is actively being discussed on the monthly ZFS leadership calls, which is a more controlled zpool upgrade process. Rather than upgrading to all of the latest features, this would allow the user to specify a level to upgrade the pool to, such as “compatible-2019”, which is the lowest common denominator of features supported by all main stream ZFS ports as of January 1, 2019. This ensures that a pool can be upgraded to get new features, but won't upgrade past what might be supported by the FreeBSD boot code, or what can be imported on OS X.

A zpool bootcode command to take care of updating the boot code on all disks in the pool could make this procedure much safer. Currently on FreeBSD, when you upgrade a pool, the output of the upgrade command includes an example command to upgrade the bootcode on your disk. However, this example command assumes you are booting in legacy BIOS mode, on a GPT partitioned disk. If you are booting in UEFI mode, or the default configuration the installer uses, where both types of bootcode are installed, this command could end up overwriting the EFI bootcode partition with the legacy bootcode, resulting in an unbootable system. Since it will not be possible for ZFS to accurately predict where the bootcode might need to go, the proposal is to relocate the bootcode to the location specified by the ZFS on-disk format. There is a 3.5 MB reserved area after the first two labels before the start of the filesystem. This is already used for the bootcode if the partition table type is MBR instead of GPT. This would then just involve the zpool command writing the data to space owned by ZFS. It would require a different version of boot0 for GPT, that found the freebsd-zfs partition and read boot1 and boot2 from the correct offset, rather than looking for boot1 and boot2 in the freebsd-boot partition. However, what should it do in the case where there are multiple freebsd-zfs partitions?

In the case of UEFI boot, things get sticky as well. Currently FreeBSD acts a bit different than most other ZFS ports when given an entire disk, rather than a partition. FreeBSD just writes the ZFS on-disk format directly to the device, with no partition table, similar to the old “dangerously dedicated” BSD partition table scheme. In illumos , where ZFS originated, when given an entire disk, ZFS writes a GPT partition table, and creates a single large partition and puts the ZFS contents in that. Recently, the zpool create command has been extended to have an optional parameter to create an ESP partition of a defined size to store the EFI loader, and then use the rest of the space for the ZFS partition. With this configuration, the ESP partition is somewhat controlled by ZFS, and could be overwritten with the newer loader as part of the zpool bootcode command. However, unlike the freebsd-boot parttion, of the reserved area in the ZFS label for other legacy bootcode, the ESP is not unformatted space, it is a FAT filesystem, and designed to container the loaders for multiple operating systems if the machine has multiple operating systems installed. It can also contain EFI native applications for diagnostics. As such, it may not be advisable to overwrite the entire partition with an image that contains only the FreeBSD bootcode. Mounting the ESP and copying files to it doesn't seem like something that should be done by the zpool bootcode command.

Lastly, another ZFS feature in the works is support for a newer compression algorithm, ZStandard (ZSTD). On a default installation of FreeBSD, the current default compression algorithm, LZ4, compresses the image approximately 2:1, but ZSTD can compress as much as 3:1, making the system image, and incremental updates to it, smaller.

7.4 bectl and libbe

bectl and libbe are the recently introduced base system utilities for managing boot environments on FreeBSD. Started as a Google Summer of Code project in 2017, the work was later completed by Kyle Evans and merged into FreeBSD 12.0.

In addition to replacing the previous shell script, beadm, bectl also provides a C library, libbe, that can be integrated into appliance control planes and used by other tools, like package managers. Kyle is also in the process of adding “deep boot environment” support, which will handle children under the root filesystem. One of the popular examples for this is having a /usr/src for each boot environment, that is a separate database, but /etc/rc.d/zfsbe takes care to mount the usr/src child for the current boot environment. This way the source tree that corresponds to the running kernel is always mounted. This could also allow things like /usr/local (for packages) to be a separate filesystem, but still be swapped in sync with the boot environment.

Additionally, libbe makes it easier to integrate a customized mechanism for deciding that the boot-once of a system image is working as expected, and make it the new default.

8. Additional Considerations

There are still many more factors to consider when designing an upgrading mechanism. Most of what is described here would work in an air-gapped environment, where the updated system image was delivered on read-only media such as a DVD, but some minor changes may need to be made to accommodate this.

A lot more consideration needs to be paid to security and authenticity. An update mechanism needs to be secure from a number of different threat vectors. The updated system images should likely be signed to prove their authenticity, which would require distributing the trusted keys with the original system image. If updates are fetched from the internet, not only do you need to consider the identity of the host you are fetching from, but also the integrity of the new system image. Additionally, there may be a need to hide the details of the update (what previous version the system is being upgraded from), which may require randomizing parts of the request to ensure the resulting response size hides the details of the update.

There is also the considerations around encryption. FreeBSD supports high performance full disk encryption using GELI. There is currently some support for GELI in the FreeBSD bootcode and loader, but it is geared towards interactive use on a laptop. There are some design documents, but no current work in progress to support headless operation of GELI by storing encryption keys on dedicated removable devices (such as a USB stick or smart card). However, since GELI operates at the block layer, below ZFS, it doesn't have much direct impact on the upgrading mechanisms. There is however a forthcoming feature in ZFS to provide filesystem native encryption. This does not encrypt all data on the disk, but provides for confidentiality of the user data and metadata of the individual filesystem. This allows only a subset of the filesystems to be encrypted, possibly with different keys. The main advantage to this design is that individual keys can be unloaded when they data is not needed, placing the data safely at rest. Importantly, the scrub (data integrity check) and resilver operations (recovering from failed disks) can operate without requiring the encrypt keys be loaded. ZFS replication can optionally send the ciphertext rather than the plain text version of the filesystem in the replication stream. This would allow the distribution of encrypted system images and updates. However, FreeBSD does not yet support this feature, and once it does, support for encrypting boot environments would likely be futher behind if it is judged useful to have at all.

9. Conclusion

FreeBSD has the tools to build a fail-safe updating mechanism, and with a bit more time to polish it, we hope to arrive at a flexible, yet easy to use solution that works for the majority of FreeBSD machines, be they appliances, embedded devices, servers, or laptops.

References

[1] Poul-Henning Kamp. *Building a FreeBSD Appliance With NanoBSD*, 2005.

https://papers.freebsd.org/2005/phk-nanobsd/

[2] Colin Percival. *An Automated Binary Security Update System for FreeBSD*, 2003.

http://www.daemonology.net/freebsd-update/binup.html

[3] Allan Jude. *Booting from Encrypted Disks on FreeBSD*, 2016.

http://allanjude.com/bsd/AsiaBSDCon2016_geliboot_pdf1a.pdf

bhyvearm64: Generic Interrupt Controller Version 3 Virtualization

Alexandru Elisei
University Politehnica of Bucharest
Bucharest, Romania
alexandru.elisei@gmail.com

Mihai Carabas
University Politehnica of Bucharest
Bucharest, Romania
mihai.carabas@upb.ro

***Abstract*—Traditionally associated with low-power, mobile computing, Arm is now seeking to enter the PC and the server markets. Virtualization is especially used in these areas, and current hypervisors rely on various hardware features to achieve efficient virtualization. To this end, the Armv8 architecture introduces a series of hardware mechanisms to reduce or eliminate some of the overhead associated with running virtual machines. Modern computers rely on hardware interrupts to communicate with peripherals, and this aspect of virtualization has seen a series of architectural optimizations from Arm. We will present our experience emulating the Generic Interrupt Controller version 3, the interrupt controller designed by Arm. We have used a mix of virtualization techniques: trap-and-emulate for the memory-mapped regions of the controller, which are accessed less frequently, and hardware accelerated virtualization where possible. To validate our approach, we have created a virtualized timer which is used to deliver timer interrupts to the virtual machines. Timers are essential for modern operating systems, and the virtualized timer is an abstraction over the Arm architectural timer, the Generic Timer. As with the interrupt controller, we have taken special care to take advantage of the available hardware mechanisms to reduce the cost of virtualization. The end result is a fully functioning hypervisor which is able to create, run and destroy virtual machines on Armv8.0-A and later processors.**

***Index Terms*—Arm, Armv8, virtualization, hypervisor, interrupts, timer**

I. Introduction

Arm is the dominant architecture in the mobile space and banking on its expertise in efficient computing, Arm is now looking to enter the PC [1], [2], [3] and the server markets [4], [5]. Virtualization is popular in these areas, especially in the server room, where more than 75% of the servers use this technology [6]. As a result, efficient virtualization solutions are necessary in order for Arm's CPU ambitions to come to fruition.

Virtualization makes it possible for an operating system to run in an environment that is indistinguishable from the real hardware [7]. One of the characteristics that makes virtual machines so appealing is resource control: the virtual machine manager is always in control of the underlying hardware resources: CPU, memory and input/output (I/O) devices. [8]. Early CPUs were slow, with speeds matching the I/O devices of the day, and a simple method of communication was used, called programmed I/O [9]. As CPUs became faster than the devices with which they interacted, a new method of communication was developed, which uses interrupts. Interrupts are electrical signals sent directly to the processor. Today, all processors use interrupts, and virtually every operating has support for such a communication mechanism. As a consequence, a virtualization solution must also provide a way for delivering interrupts to a virtual machine, while still retaining control over the hardware.

bhyvearm64 [10] is a type 2 hypervisor for the FreeBSD operating system and it implements the virtual interrupt controller on top of the Arm interrupt controller. Arm calls its implementation of the interrupt controller the Generic Interrupt Controller (GIC), and we have focused on version 3 (GICv3) of the controller. As with other components of the architecture, Arm has taken care to design the controller with features that make virtualization more efficient.

GICv3 has three separate components: the Distributor, the Redistributor and the CPU interface [11]. The Distributor and Redistributor are memory-mapped and are accessed usually only during the kernel's boot process. Accessing these two components is not performance critical and we have opted for a trap-and-emulate approach to virtualization, where the I/O registers are emulated and stored in memory as part of the virtual machine context. On the other hand, the CPU interface is used frequently, each time an interrupt is handled, and has been designed as part of the processor, accessible via fast hardware registers. For the CPU interface we have taken advantage of various hardware features designed to achieve fast virtualization.

Interrupt controllers are used by input/output devices to communicate with the processor, and we have emulated the system timer as the first device to use our virtual interrupt controller. Timers are essential to the operation of any system for a variety of reasons [9]. Among other things, they are a key part of process scheduling. Arm's implementation of the system timer is called the Generic Timer, and it actually consists of several timers [12]: the physical and the virtual timers, which are always present; the secure physical timer, present when the CPU implements the secure mode of operation; and the physical Exception Level 2 (EL2) timer, part of the virtualization extensions. An operating system is free to choose between the physical and the virtual timer, and we

This work has been sponsored by the FreeBSD Foundation.

have chosen to emulate both of them in order to not restrict a guest operating system to one of the timers. While emulating the timers we also had to take into account the fact that the host operating system must have exclusive access to one of the timers for its own uses.

With the interrupt controller and the timer both emulated, bhyvearm64 is capable of successfully booting and running a FreeBSD guest, albeit with some limitations due to the project being in its early stages: a virtual machine can only run on one virtual CPU, and besides virtio support there is little user space device emulation support.

The structure of the paper is as follows: the next section will cover present hardware and software approaches to interrupt controller and timer virtualization. In section 3 we will describe the Armv8.0 virtualization model that bhyvearm64 implements. Section 4 is dedicated to the Arm interrupt controller: first the GICv3 architecture will be explained, followed by our approach to virtualization. Then we will proceed to presenting the emulated timer in Section 6. Section 7 will cover what we believe the immediate goals for bhyvearm64 should be in order to reach a level of functionality similar to bhyve on x86. Finally, we will present our conclusions regarding virtualization on the Arm platform.

II. Related Work

Virtualization is not new. It appeared in the 1960s [9], and gained momentum in the late 1990s when VMware launched the first virtualization solution for the x86 platform [13], [9]. At that time, the x86 CPU architecture was unvirtualizable according to the Popek and Goldberg definition and interrupt injection and handling was done entirely in software. The current version of the Intel interrupt controller, called the x2APIC, has advanced support for virtualization and virtual interrupts can now be asserted by the hardware, without hypervisor intervention [14].

On the Arm side, the first hypervisor to take advantage of the Arm virtualization extensions was KVM [15], [16], which is part of the Linux kernel. KVM uses a similar approach to bhyvearm64 for virtualizing the interrupt controller, taking full advantage of hardware virtualization where available, and resorting to a trap-and-emulate technique when that is not possible.

Operating systems require timers to perform basic operations like scheduling or measuring the passing of time, but when running inside a virtual environment it is impossible to have the same precision as the physical machine because of the inherent virtualization costs.

For historical reasons, the x86 architecture implements several timers with various and sometimes overlapping uses: some of them are used by software to measure the frequency of another timer; and several are capable of generating a periodic interrupt suitable for timekeeping. This adds to the software complexity of offering a virtualized view of the passage of time, as multiple timers need to be emulated and kept in sync with each other [13].

By contrast, timer virtualization is relatively simple on the Arm platform, as the General Timer is mandated by the architecture and is adequate for timekeeping usage by an operating system. KVM uses an approach similar to bhyvearm64: the virtual timer is made available to the virtual machine, with the caveat that the interrupts generated by the timer still have to injected by the hypervisor, The physical timer is emulated in software because it is used by the host [15].

Arm has created a new virtualization model in version 8.1 of the architecture called Virtual Host Extensions (VHE) [12]. When this feature is enabled, the host operating system executes in a different CPU execution mode than the virtual machine, with access to its own separate hardware timer. The host can then assign the physical timer as well as the virtual timer to the virtual machine, eliminating the need for software emulation. KVM is working towards removing timer emulatation in this scenario [17]. Currently bhyvearm64 doesn't take advantage of VHE, but support is planned in the near future.

III. Background

bhyvearm64 is a type 2 hypervisor and virtual machine manager for the FreeBSD operating system. It is based on the existing bhyve virtualization solution for the x86 architecture. Fig. 1 shows the main components of bhyvearm64, which can be broadly categorized into user space programs and kernel code. The user space programs are bhyveload, bhyve and bhyvectl which an user employs to create, run and destroy a virtual machine. Communication with the kernel is facilitated by the library libvmmapi, which serves as a wrapper over ioctl calls to a special device which uniquely identifies the virtual machine. On the kernel side, the hypervisor is implemented as a loadable kernel module name vmm.ko. The virtual interrupt controller is abstracted as the software component named VGIC and the virtualized timer as vtimer. Perhaps contrary to its name, the virtual timer is a physical, hardware timer and not a software abstraction, as is our virtualized timer. Both the VGIC and the vtimer are emulated in kernel space to achieve better performance and less overhead.

Arm introduced the virtualization extensions with version 7 of the Arm architecture, Armv7, and Armv8.0 follows the same virtualization model. For CPUs that support virtualization, the necessary hardware support is implemented as a distinct processor execution mode, called Exception Level 2 (EL2). The hypervisor architecture is similar to KVM [15], the Linux hypervisor. EL2 was created with a type 1 hypervisor in mind. A type 1 hypervisor [9] runs directly on the hardware and its functionality is centered around managing virtual machines, as opposed to both virtual machines and user space programs as is the case with a type 2 hypervisor plus host kernel. This design decision unfortunately makes it impractical to run the host operating system in EL2, and instead has forced us to split the hypervisor code to run across two different CPU execution modes.

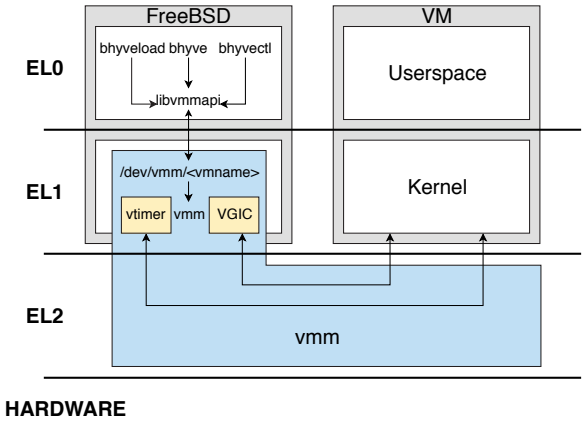


Fig. 1. bhyvearm64 architecture

There are several versions of the Arm Generic Interrupt Controller. The most common are GICv2 and GICv3. GICv2 has two major drawbacks:

- Supports at most eight processors, which is limiting for today's platforms, especially when it comes to server hardware.
- It is entirely memory-mapped, which makes accessing the registers expensive.

Version 3 of the controller addresses these drawbacks by not putting a strict limit on the number of CPUs and implementing hardware registers for the most frequently used operations, registers which have virtualization support. Version 4 of the interrupt controller is identical to version 3 from a software perspective, the only difference being added support for virtual message-based interrupts. We expect adding support for GICv4 in the future will be relatively painless.

IV. GICv3 Architecture

Before describing the process of emulating interrupts, it is worth getting familiar with the inner workings of the Arm Generic Interrupt Controller version 3 (GICv3). An interrupt is an asynchronous, external electrical signal delivered to the processor [9]. The GICv3 controller implements four different types of interrupts:

- Software Generated Interrupts (SGI). These are generated by the operating system and used for inter-processor communication. On other architectures, they are known as Inter-Processor Interrupts (IPIs).
- Private Peripheral Interupts (PPI). This type of interrupts are generated by devices that communicate with only one CPU core, which is always the target for the interrupt. Timer interrupts are PPIs.
- Shared Peripheral Interrupts (SPI). These are interrupts that originate from I/O devices and can target any core in the system.
- Locality-specific Peripheral Interupts (LPI). These are message based interrupts and can be used by PCI Express devices or other devices. The PCI Express specification calls them Message-Signaled Interrupts (MSI) [18].

Besides their type, interrupts have other attributes that can play a major role in deciding when and how they are delivered to the processor: interrupt group, which can be group 0, non-secure group 1 or secure group 1, and interrupt priority. Interrupts can be delivered to the CPU as either an IRQ or a FIQ (distinguished by their offset in the interrupt vector). The security state (secure or non-secure) and their group are the deciding factors in asserting an interrupt [19]. Group 0 interrupts are always delivered as FIQ interrupts. FreeBSD configures all interrupts as non-secure group 1 interrupts, which are always delivered as IRQs.

Interrupts can be masked based on their priority. Interrupt priority also serves to arbitrate between multiple interrupts: the interrupt with the highest priority will be asserted first. There are attempts to use this priority mechanism to allow for pseudo Non-Maskable Interrupts (NMI) on the Arm architecture: interrupts designated as NMI will have a higher priority associated with them, and when disabling interrupts, instead of setting the PSTATE.I bit, a priority mask is used that will block all "regular" interrupts, while NMIs can still be asserted [20].

Fig. 2 is an overview of the GICv3 architecture. There are three main components: a single Distributor per system, one Redistributor and one CPU interface per core. The Distributor is responsible for the configuration of the global interrupts (SPIs) and the Redistributor is used to configure various properties of the interrupts that are private to the core (PPIs and SGIs). The CPU interface is responsible for advancing the state machine associated with handling an interrupt. The Distributor and the Redistributor are expected to be used sporadically, typically at boot to configure the interrupts, as opposed to the CPU interface, which is used each time an interrupt is handled. Their implementation mirrors this usage pattern: the Distributor and the Redistributor are memory-mapped, and accessing them is slower, but that is acceptable because it is rarely done; the CPU interface is implemented as hardware registers and that means faster accesses.

There is one optional component that is missing from the figure. That component is the Interrupt Translation Service (ITS) and it is responsible for message-based interrupts. It is optional because system integrators can choose to implement equivalent functionality in the Redistributor. bhyvearm64 doesn't support LPI virtualization and for this reason it has been omitted.

V. GICv3 Virtualization

The virtual interrupt controller has been implemented taking into account the specifics of the GIC components and how FreeBSD uses the interrupt controller. Interrupts can be configured as group 0, non-secure group 1 or secure group 1 interrupts. Secure group 1 interrupts are always delivered to the firmware running in the secure world, and those type of interrupts haven't been implemented in bhyvearm64, as the hypervisor runs in the non-secure world.

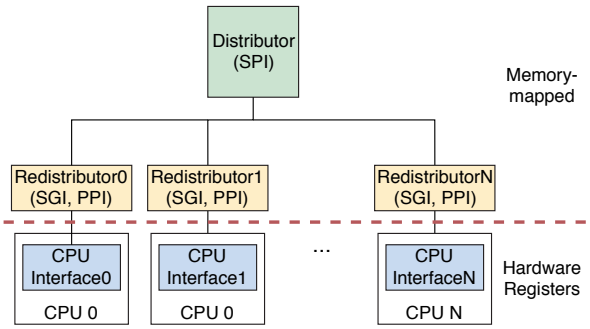


Fig. 2. GICv3 architecture

Group 0 interrupts are always delivered as FIQ interrupts and the firmware can configure the hardware to deliver those interrupts to the secure world, similar to how secure group 1 interrupts work. For this reason, FreeBSD and Linux choose to configure all interrupts as non-secure group 1 interrupts and bhyvearm64 has support for only this use case.

The interrupt controller is emulated entirely in kernel space. Interrupts are time sensitive events, and switching execution to user space to handle the emulation, and then switching again to the kernel would have proven too costly. However, in order to make it possible for the virtual machine manager to emulate various devices, we have provided an API for asserting and retiring interrupts.

A. Distributor and Redistributor emulation

Because the Distributor and the Redistributors are memory-mapped and are seldom accessed, we have chosen a trap-and-emulate approach for virtualization. This approach takes advantage of how memory virtualization works: the guest physical addresses that correspond to the Distributor and Redistributor registers aren't mapped in the Stage 2 translation tables. The Stage 2 tables are responsible for translating a guest physical address generated by the virtual machine into a real address in physical memory. When the guest address isn't present in the tables, an exception occurs. With the information associated with the exception, the hypervisor is able to reconstruct the guest instruction and emulate it accordingly without the need to propagate the fault to user space.

The virtual Distributor and Redistributors are purely software constructs that exist in the host's memory as part of the virtual machine context. Each time the guest tries to access these virtual registers, the hypervisor is able to extract the address from the exception syndrome. For each such memory region we maintain an array sorted by the start address, and using binary search we are able to quickly determine which virtual register the virtual machine is accessing. Emulation consists mainly in saving the value written by the guest and returning that value on a read. The register values are also used for determing which interrupt can be presented to the virtual machine, in a manner similar to how the hardware works.

A complete list of interrupt controller registers that are part of the virtual machine context can be found in Table I.

B. CPU Interface virtualization

The CPU Interface is used every time an interrupt is handled, therefore it makes sense to make read and writes fast. The CPU Interface is implemented as registers that are part of the CPU and has support for hardware virtualization. Virtualization is activated when the hypervisor configures EL2 to route all physical group 0 and group 1 IRQs to EL2 by setting the HCR_EL2.IMO and HCR_EL2.FMO bits. The purpose of these settings is twofold:

- All physical interrupts will be routed to the host, therefore enforcing the separation between the hardware and the guest.
- All accesses to the CPU Interface registers are transparently redirected to a separate set of registers with identical functionality, but which control the handling of virtual interrupts instead of physical interrupts.

Because the virtual CPU Interface registers are used when advancing the state machine of a virtual interrupt in exactly the same way that the non-virtual registers are used for physical interrupts, they are not writable by the hypervisor and are not considered part of the virtual CPU context. However, additional registers are used for asserting a virtual interrupt, and these registers are only accessible at EL2.

C. Virtual interrupt injection

Virtual interrupt injection and handling is done mostly in hardware. The hypervisor is responsible for choosing which interrupt to inject in the guest. After the interrupt is injected, its state becomes pending. When the guest is resumed, the interrupt is asserted to the guest. The rest of the state transitions are handled by the virtual CPU interface and no intervention from the hypervisor is necessary.

TABLE I
VIRTUAL GIC REGISTERS

Component	Type	Register	Description
Distributor	uint32_t	GICD_CTLR	Distributor Control
	uint32_t	GICD_TYPER	Distributor Type
	uint32_t	GICD_PIDR2	Peripheral ID2
	uint32_t *	GICD_ICFGR	Interrupt Config
	uint32_t *	GICD_IPRIORITYR	Interrupt Priority
	uint32_t *	GICD_IXENABLER[a]	Interrupt Enable
	uint64_t *	GICD_IROUTER	Interrupt Routing
Redistributor	uint32_t	GICR_CTLR	Redistributor Control
	uint32_t	GICR_TYPER	Redistributor Type
	uint32_t	GICR_IXENABLER0[b]	Interrupt Enable
	uint32_t	GICR_ICFGR0	Interrupt Config 0
	uint32_t	GICR_ICFGR1	Interrupt Config 1
	uint32_t[]	GICR_IPRIORITYR	Interrupt Priority
System Registers	uint32_t	ICH_EISR_EL2	EOI Status
	uint32_t	ICH_ELRSR_EL2	Empty LRs
	uint32_t	ICH_HCR_EL2	Hypervisor Control
	uint32_t	ICH_MISR_EL2	Maintenance Status
	uint32_t	ICH_VMCR_EL2	VM Status
	uint64_t[]	ICH_LR_EL2	List Registers

[a]Combination of GICD_ICENABLER and GICD_ISENABLER.
[b]Combination of GICR_ICENABLER0 and GICR_ISENABLER0.

To inject an interrupt, the CPU provides the hypervisor with a series of registers, called List Registers. Each List Register contains information about one virtual interrupt that will be handled by the guest: the interrupt group, state, priority, interrupt number and if the virtual interrupt maps directly to a physical interrupt. A virtual interrupt can shadow a physical interrupt, and in this case, when the guest deactivates the virtual interrupt, the corresponding physical interrupt is also deactivated.

The number of List Registers is limited and hardware-dependent. The maximum number is 16 and it is possible to have more pending interrupts for the virtual machine than the number of List Registers. To get around this limitation we keep our own buffer for the pending interrupts. Each time the guest is resumed, we check this buffer and select the highest pending interrupts to be injected in the guest.

The interrupts that will be asserted are selected based on the guest interrupt configuration and it takes into account:

- The group, type and interrupt number: the interrupt must be enabled in the Distributor and the Redistributor.
- If the target CPU for the interrupt is the current CPU.
- The priority of the interrupt relative to the other pending interrupts.
- If two interrupts are equal in terms of priority, the hypervisor keeps an extra field for each interrupt for additional information. For example, a clock interrupt should always have higher priority because this is how the guest operating system does its timekeeping.

Another hardware features that is designed to help with virtualization is the presence of a special interrupt, called the maintenance interrupt. The purpose of this interrupt is to address scenarios where the hypervisor wants to inject more interrupts than available list registers or when a special action needs to be performed when a certain virtual interrupt is handled. In such situations, the hypervisor enables the maintenance interrupt which when asserted will trigger a world switch to the host. The hypervisor is then free to execute the action it deems appropriate.

VI. Timer Virtualization

The timer is essential for any operating system: without it, there could be no process scheduling in the context of preemptive scheduling. Operating systems also use a timer for periodic tasks, either as a functionality offered to user space processes, or for internal purposes. It is necessary for a virtual machine to have access to a virtualized timer in order for the guest operating system to function properly.

A. The Generic Timer

The timer provided by the Armv8 architecture is called the Generic Timer. The implementation actually consists of at least two different timers, up to seven [12]. A system can have a secure physical timer, a non-secure physical timer, which we will call simply the physical timer, a virtual timer, physical and virtual non-secure EL2 timers, and physical and virtual secure EL2 timers. For the purpose of virtualization, we will focus our attention on the timers that a regular operating system uses, the physical timer, which counts the passing of real time, and the virtual timer, which counts the passing of time from a fixed offset.

The host operating system needs to use a timer exclusively; it is not desirable for a virtual machine to slow down the host. bhyvearm64 assigns the physical timer to the host and the virtual timer to the virtual machine currently running on the CPU core for the following reasons:

- Because the virtual timer counts time from a fixed offset, a guest running inside a virtual machine can be tricked into thinking that the timer started at the same time as the virtual machine.
- FreeBSD [21] and Linux [22] prefer choosing the virtual timer over the physical timer when they are both present and virtualization is not active, which is always the case in a virtual machine with no nested virtualization support.
- The Armv8.0 architecture provides a mechanism to emulate the physical timer by trapping reads and writes; there is no such mechanism for the virtual timer.

B. Virtual Timer Virtualization

Timer interrupts are extremely time sensitive. Timer interrupts come at regular intervals (the FreeBSD kernel configures the timer to fire once every 1 millisecond) and because they are so frequent it is extremely undesirable to spend too much time servicing the interrupt. That time can be used instead to execute other tasks. The same is true for the virtualized timer: the less time the hypervisor spends emulating a timer, the more CPU time a virtual machine has at its disposal before the next interrupt.

To achieve minimal overhead for injecting timer interrupts, bhyvearm64 assigns the virtual timer component of the Generic Timer directly to the virtual machine. The guest operating system is free to configure the timer as it sees fit, without any intervention from the hypervisor. However, virtual timer interrupts still need to be managed by the hypervisor. This is necessary because according to Popek and Goldberg's control property [8], the host must always be in control of the hardware, and this also means controlling the delivery of interrupts. There is no hardware mechanism for selecting which interrupts get redirected to the virtual machine. When a guest is running, all interrupts are routed to the host, which will choose which of them will be presented to the virtual machine.

By their nature, interrupts are asynchronous; they can come at any point in time regardless of the program that the processor is executing. This also applies to the virtual timer: a virtual timer interrupt can fire when another host program is running on the CPU instead of the virtual machine that programmed the timer. The virtual timer requires a mechanism for identifying the virtual machine that programmed it before it fired. To achieve this, we have modified the machine dependent part of `struct pcpu` to save a pointer to the last virtual CPU that ran on the core, as shown in Listing 1. The virtual CPU is changed each time a different virtual processor is run by

the machine-dependant part of bhyvearm64 and set to `NULL` when that machine is destroyed.

The correct usage of the virtual timer also requires considering the case when two distinct virtual machines are sharing the same physical core and thus using the same virtual timer. It is important to note that in this context, "sharing" means the CPU execution is alternating between the two virtual machine. The virtual machines most likely started at different times, and their virtual timer offsets will reflect that; most importantly each will set the timer to fire at different moments in the future. To account for this scenario, when switching virtual machines, it is necessary to save the virtual machine timer state and restore the state of the virtual machine that is replacing it.

Listing 1. struct pcpu

```
#define PCPU_MD_FIELDS                  \
    u_int   pc_acpi_id;                 \
    u_int   pc_midr;                    \
    uint64_t pc_clock;                  \
    void    *pc_vcpu;                   \
    pcpu_bp_harden pc_bp_harden;        \
char __pad[225]
```

There is one other important aspect of timer virtualization that needs to be addressed: what happens when the virtual machine is running behind timer interrupts? We have experienced this situation when running bhyvearm64 on the Foundation Platform simulator [23] with multiple virtual machines on the same (simulated) Armv8.0 CPU. For bhyvearm64 we have chosen a conservative approach in order to prevent the guest kernel from spending too much of its CPU time handling timer interrupts. When a virtual timer interrupt is asserted, we don't inject the interrupt in the guest unconditionally, but instead we check if another timer interrupt is active. This can happen in the interrupt handler, after the guest enables the timer and before it signals the end of interrupt, events that are shown in Fig. 3. In this case, we save the new interrupt in the interrupt buffer and we inject it next time we perform a world switch. Because world switches occur at least once every host tick, the guest will have lost at most one full host tick.

C. Physical Timer Emulation

We have discovered that FreeBSD and Linux prefer using the virtual timer when it is available. However, there is nothing stopping an operating system from choosing the physical timer over the virtual timer. Because bhyvearm64 lets the host have control over the physical timer, for physical timer virtualization we have chosen a trap-and-emulate approach. This is achieved by setting the CNTHCTL_EL2.EL1PCEN and CNTHCTL_EL2.EL1PCTEN bits, which cause all accesses to the physical timer to be trapped to the hypervisor.

Fig. 3 shows the steps the FreeBSD kernel executes when handling a timer interrupt. To get the interrupt number, the Interrupt Acknowledge Register (IAR) is read. The interrupt number is the number programmed in the List Register. This changes the interrupt state from pending to active. The kernel disables the timer by writing to the CNTP_CTL_EL0

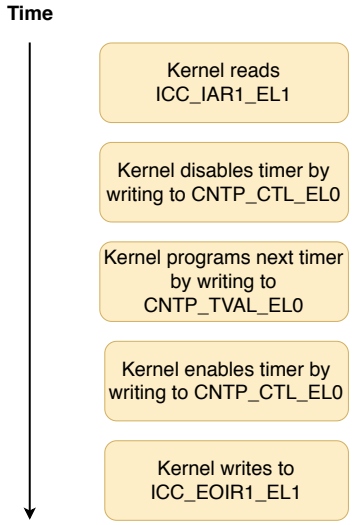


Fig. 3. Timer Usage

register, which causes a trap to the hypervisor where the hypervisor does in-kernel emulation. As a result of the write, the hypervisor disables all pending timer alarms for the guest. The guest programs the timer for the next alarm, and we save this value. We don't program any alarms to inject an interrupt because the timer is still disabled.

Only after the guest enables the timer with another write to CNTP_CTL_EL0 we trap to the hypervisor and program an alarm at the time specified by the guest by using the FreeBSD's callout API. To end the handling of this interrupt, the kernel writes to the End Of Interrupt Register (EOIR), which marks the interrupt as inactive in the List Register. The List Register that held the interrupt is now available to be used for injecting another interrupt.

VII. Future Work

bhyvearm64 is in the early stages and our main goal moving forward is to integrate it with the FreeBSD operating system. To this end, we are pursuing three different approaches: splitting the existing bhyve implementation into machine independent (MI) and machine dependent (MD) code, improving user space device emulation support and improving the hypervisor.

The arm64 vmm module duplicates code from the x86 bhyve implementation. It is obvious that, at the very least, the vmm device code should be very similar between the two architectures. This also applies to the user space components of bhyvearm64, because much of the libvmmapi ioctl wrappers and device emulation code should be shared. This issue was also raised during the review process for the Armv7 version of bhyve [24].

We are currently working on separating the machine independent from machine independent code and we have started with libvmmapi [25]. We will continue with the rest of the

user space utilities, before turning our attention to the kernel module.

At the moment, bhyve for arm64 has support for virtio devices and bvmconsole, which is a development console. This is inadequate for proper virtual machine management. We plan to emulate the Intel 16650 UART and a CD-ROM device. The UART will make interacting with the virtual machine faster, and the emulated CD-ROM will make it possible for the user to install a FreeBSD operating system inside the virtual machine.

The Armv8.0 virtualization model was intended for type 1 hypervisors, and this has the unfortunate effect of making type 2 hypervisors not only more complicated from a software perspective, but also slower. Better support for type 2 hypervisors was added in Armv8.1 under the name of Virtual Host Extensions (VHE) [12]. KVM on Linux implements VHE and this approach has led to better performance compared to Armv8.0 virtualization in all scenarios [16]. Implementing VHE brings the added advantage of noticeably reduced software complexity for the hypervisor. This reduced complexity will make our next goal easier to achieve: adding Symmetric Multiprocessor Support (SMP) support to the virtual machine. Machines with one CPU are rarely seen today, and we plan to make it possible for a virtual machine to use multiple virtual processors.

VIII. Conclusions

Operating systems rely on interrupts to communicate with input/output devices. It is therefore necessary for modern hypervisors to implement interrupt virtualization. bhyvearm64 abstracts Arm's Generic Interrupt Controller version 3 into a virtual interrupt controller by using a multifaceted approach to virtualization. bhyvearm64 takes advantage of hardware features to achieve minimum overhead by enabling the virtual CPU interface. For the seldom accessed, memory-mapped components of the GIC we make use of translation faults to emulate the corresponding reads and writes.

Timers are essential for modern computers because, among other things, they make multiprogramming possible. The first device that uses the virtual interrupt controller is the virtualized Generic Timer, which provides the virtual machine with timer interrupts. bhyvearm64 virtualizes both hardware timers that are part of the Generic Timer. A guest running in a virtual environment is allowed to take full control over the virtual timer. Physical timer accesses are emulated using a trap-and-emulate approach, where the timer state is a software construct part of the in-memory virtual machine state.

Acknowledgment

The current version of bhyvearm64 was inspired by bhyve for Armv7 by Mihai Carabas. bhyvearm64 uses a virtio MMIO implementation by Mihai Darius.

References

[1] Qualcomm Incorporated, "Always On Always Connected PCs are here." https://www.qualcomm.com/snapdragon/always-connected-pc. Last accessed: 21 January 2019.

[2] AnandTech, "Arm's Cortex-A76 CPU Unveiled: Taking Aim at the Top for 7nm." https://www.anandtech.com/show/12785/arm-cortex-a76-cpu-unveiled-7nm-powerhouse. Last accessed: 21 January 2019.

[3] Arm Ltd, "Accelerating mobile and laptop performance: Arm announces Client CPU roadmap." https://www.arm.com/company/news/2018/08/accelerating-mobile-and-laptop-performance. Last accessed: 21 January 2019.

[4] S. McIntosh-Smith, J. Price, T. Deakin, and A. Poenaru, "Comparative Benchmarking of the First Generation of HPC-optimised ARM Processors on Isambard." https://uob-hpc.github.io/assets/cug-2018.pdf. Last accessed: 28 January 2019.

[5] Amazon.com, Inc, "Introducing Amazon EC2 A1 Instances Powered By New Arm-based AWS Graviton Processors." https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-ec2-a1-instances/. Last accessed: 21 January 2019.

[6] Gartner, Inc, "Gartner says worldwide server virtualization market is reaching its peak." https://www.gartner.com/en/newsroom/press-releases/2016-05-12-gartner-says-worldwide-server-virtualization-market-is-reaching-its-peak. Last accessed: 21 January 2019.

[7] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2013.

[8] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Commun. ACM*, vol. 17, pp. 412–421, July 1974.

[9] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*. Upper Saddle River, New Jersey: Pearson Education, Inc., 2015.

[10] A. Elisei, "bhyvearm64." https://github.com/FreeBSD-UPB/freebsd/tree/projects/bhyvearm64. Last accessed: 27 January 2019.

[11] ARM Holdings, "ARM Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and version 4.0." https://static.docs.arm.com/ihi0069/c/IHI0069C_gic_architecture_specification.pdf. Last accessed: 28 January 2019.

[12] ARM Holdings, "ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile." https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf. Last accessed: 23 January 2019.

[13] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang, "Bringing virtualization to the x86 architecture with the original vmware workstation," *ACM Trans. Comput. Syst.*, vol. 30, pp. 12:1–12:51, Nov. 2012.

[14] Intel Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual." https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf. Last accessed: 20 February 2019.

[15] C. Dall and J. Nieh, "KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor." http://www.cs.columbia.edu/~cdall//pubs/asplos019-dall.pdf. Last accessed: 28 January 2019.

[16] C. Dall, S.-W. Li, and J. Nieh, "Optimizing the Design and Implementation of the Linux ARM Hypervisor," in *2017 USENIX Annual Technical Conference (USENIX ATC '17)*, 2017.

[17] M. Zyngier and C. Dall, "KVM: arm/arm64: arch_timer: Assign the phys timer on VHE systems." https://www.spinics.net/lists/arm-kernel/msg702086.html. Last accessed: 21 February 2019.

[18] PCI-SIG, "Specification." http://pcisig.com/specifications (login required). Last accessed: 26 Junuary 2019.

[19] ARM Holdings, "GICv3 and GICv4 Software Overview." http://infocenter.arm.com/help/topic/com.arm.doc.dai0492b/GICv3_Software_Overview_Official_Release_B.pdf. Last accessed: 26 January 2018.

[20] J. Thierry, "arm64: provide pseudo NMI with GICv3." https://lkml.org/lkml/2019/1/21/1060. Last accessed: 28 January 2019.

[21] The FreeBSD Project, "Generic Timer driver." https://github.com/freebsd/freebsd/blob/master/sys/arm/arm/generic_timer.c. Last accessed: 26 January 2019.

[22] L. Torvalds, "The Linux Kernel." https://github.com/torvalds/linux/blob/master/drivers/clocksource/arm_arch_timer.c. Last accessed: 27 January 2019.

[23] ARM Holdings, "Fixed Virtual Platforms." https://developer.arm.com/products/system-design/fixed-virtual-platforms. Last accessed: 27 January 2019.

[24] M. Carabas, "Adding virtualization support for ARMv7 platforms." https://reviews.freebsd.org/D10213. Last accessed: 28 January 2018.

[25] A. Elisei, "libvmmapi: Separate MI from MD code." https://reviews.freebsd.org/D17874. Last accessed: 28 January 2018.

BSD Unix Solutions in the Australian NFP/NGO Health Sector

Jason Tubnor – ICT Senior Security Lead

Latrobe Community Health Service, Victoria, Australia
jason.tubnor@lchs.com.au, jason@tubnor.net

Abstract

Latrobe Community Health Service (LCHS) is a Not for Profit (NFP)/Non-Government Organisation (NGO) in Victoria, Australia. By 2018, the organisation had grown to 51 offices across the State of Victoria with over 1,000 employees. All LCHS infrastructure is designed and managed in-house without the use of large-scale cloud infrastructure.

Since 2015, BSD Unix has been used for various workloads within the organisation, with application instances interchanging between BSD distributions where it was deemed that one type was stronger than another at specific roles in the LCHS environment. LCHS is operating system distribution and technology-agnostic and prefer both FreeBSD and OpenBSD where either one and/or its associated base tools are most suitable.

This paper outlines the various design considerations and configurations of OpenBSD and FreeBSD in multiple roles in our organisation.

1 Introduction

In 2015, Latrobe Community Health Service (LCHS) started to look towards open source to solve problems that couldn't be easily solved with available Microsoft technology or at a reasonable price point. The Information, Communication Technology (ICT) team were tasked to think outside the box and come up with solutions to these problems.

This led to some technology decisions that when reflected upon, were good for the company's long-term goals.

While there are more features of the different BSDs in use at LCHS, this paper will focus on the high level of the following technology implementation:

- OpenBSD PF(4) and traffic queueing.
- OpenBSD ripd(4) dynamic routing, network design considerations and scaling it across sites with multi-vendor implementations of RIPv2.
- Design and deployment of a commodity, 1U appliance using FreeBSD, OpenZFS and bhyve(8) as a host.
- A virtualized OpenBSD guest to provide branch offices network infrastructure and access to head office corporate services, using simple and cheaply available layer 3 Internet services.
- Migrating the marketing USB archive drive to a 36TB raw storage FreeBSD OpenZFS server, replicating data sets across 3 sites.
- Using tools such as setfib(1) in FreeBSD or rdomain/rtable(4) in OpenBSD to connect a host to a separate storage network.

2 OpenBSD ripd(8) and pf(4)

A strategic change of direction for the organisation occurred in 2015 and from an ICT perspective, it meant a move away from being solely a Windows shop. Network management was being brought back in-house from an outsourcing provider and a local Internet Service Provider was engaged to provide the organisation a private, layer 3network between sites using whatever technology was available to provide the bandwidth required. Initially this included eight sites that needed to be interconnected with around 10/10Mb symmetrical connections in a hub and spoke configuration. Due to the size of the network and the unknown state of the topology because of outsourced management, original agreements stated that routing would be managed by the ISP using static routing that was then distributed in their core network via BGP.

Once it became apparent that this was not going to be scalable if the organisation ever grew, a plan was required to allow for management and distribution of routes across the entire network. Due to the configuration of our ISPs network, BGP was ruled out. The organisation was also swapping out aged Cisco Catalyst switches for Brocade ICX switches, limiting the use of other protocols that could be used for route distribution.

The network design contained an OpenBSD firewall for the Internet gateway. Using the excellent documentation provided by the project located in the base install, assisted in making the most appropriate decision. As most are aware, OpenBGPD is a major sub-project within the OpenBSD base system but the BGP it offered was not useable due to what was mentioned above. However, another routing protocol that was part of the project at the time and still being maintained is ripd(8). While this protocol is old and there are many more modern and preferable routing protocols, this would be enough for the organisation based on the new network topology and being interoperable between all software and hardware that will make up the network refresh.

As the connection and configuration of the network took place, work went into the design and build of the OpenBSD Internet gateway. Key components of the gateway simply consisted of network components that were found in the base operating system, such as ripd(8) using version 2 only of the protocol and pf(4). Initially, the organisation size only dictated the configuration of the pf(4) rules to be block all/explicitly permit and keep the traffic as first in, first out. As the organisations requirement for Internet connectivity grew and a limited budget prevented an increase in bandwidth, creativity was required to design and build queues to manage end users' consumption, ensuring all users had a good experience.

2.1 ripd(8)

ripd(8) configuration in OpenBSD is very simple and only requires the daemon to be enabled (`rcctl enable ripd`) after the modification of the excellent example ripd.conf(5) that is located in `/etc/examples/` and copied to `/etc`.

A simple example of ripd.conf(5) when running on a firewall/route of last resort without any authentication is (bge1 being the external interface):

```
fib-update yes
redistribute default
split-horizon poisoned
triggered-updates yes

interface bge0 {
}

interface bge1 {
        passive
}
```

The ripd(8) routing protocol implementation was successful, allowing the firewall to participate in announcements and broadcasts. During implementation, use of the ripctl(8) tool allowed for debugging and rectification of the ripd(8) configuration. It provided the ability to determine if neighbours are connected and active as well as what routes the daemon was receiving.

Stability of ripd(8) was an issue in OpenBSD 5.8 with the ripd(8) daemon consuming RAM over time and not releasing it even though the network wasn't growing in size. This would cause the host to remain up but the RIPv2 routing daemon would crash, losing routes to the rest of the network. A restart of the ripd(8) daemon would be required to have the host operational again and it was hard to pinpoint what was the problem initially. More recent versions of OpenBSD appear to have corrected these issues.

One of the newer features that was tested but not yet used in production is the ability to launch multiple ripd(8) instances to run in other rdomain(4)s without one knowing about the other. This is good for where an OpenBSD router is multi-tenanted with isolated network traffic.

One feature request for ripd(8) would be to have the ability to reload the configuration file or perform other operations without having to restart the daemon (similar to iked(8) using ikectl(8)) to avoid causing an interruption to the routed traffic.

2.2 pf(4) and traffic queueing

Out of the box, pf(4) on OpenBSD provides an excellent firewall solution for the front of a private network. A clean room approach was taken in the design of the rules that were required for internal applications to work.

By using a default block all approach in the rule set, a couple of applications that were affected by missing rules failed, however, these were fixed, and appropriate documentation was created for these applications as well as their networking requirements.

Over the next 6 months, network issues started to appear with DNS timeouts and other applications failing that were latency sensitive. An investigation uncovered that the internal userbase were consuming all the inbound bandwidth of the public Internet connection. Bandwidth is expensive in Australia and it is not as simple as turning a knob to increase bandwidth.

The pf.conf(5) rule set was reviewed to break down the different protocols and sort into time sensitive, low bandwidth to low priority, high bandwidth.

What is misunderstood is that you can control inbound traffic coming into your network, even if access to the upstream device is unavailable – typically only outbound traffic can be controlled. By setting outbound queues on the internal interface, it effectively rate limits the traffic heading to end-users. In turn, this holds back the ACK for TCP traffic or the data stream for UDP.

Both pf(4) and its queueing algorithm work exceptionally well, with only minor rule changes when a new application comes along or in the case of queueing, when bandwidth is eventually increased.

Since 2018, the organisation increased the office count that used this firewall from 8 to 24 and bandwidth to 100/100Mb symmetrical with a significant increase of devices utilising this firewall as their primary Internet gateway. While different protocols NAT to individual IP addresses, the default state limits in pf(4) became an issue. The parameter *set limit states* was raised from the default 10,000 to 30,000 without any notable impact to RAM. However, since OpenBSD 6.4, the default state limit has been increased to 100,000 states[1].

On average, an observed 25MB of RAM is consumed with 12% of CPU utilisation on a single core for pf(4) under peak user load.

[1] henning@ increased the default state table size from 10,000 to 100,000
https://cvsweb.openbsd.org/src/sys/net/pfvar.h?rev=1.480&content-type=text/x-cvsweb-markup

3 FreeBSD bhyve(8) appliance

In 2017 the organisation was awarded a second round of contracts to provide services for the National Disability Insurance Scheme (NDIS), an Australian Government initiative.

The ICT team were tasked to come up with a solution that would provide network and site server facilities to satellite offices at a cost-effective rate. After a significant amount of research to come up with suitable hardware that could withstand hostile environments (high temperatures/dust), the decision was made to use FreeBSD 11.0 with bhyve(8) and OpenZFS to build a custom hypervisor instead of using VMWare's ESXi server, which was also trialled during the proof of concept stage (PoC).

In the first design of these hosts, a reliable management tool that was easily used by team members still didn't exist and bhyve/UEFI was still relatively new, making the project look rather un-realistic from the start.

The *chyves* bhyve management project was evaluated and achieved 90% of requirements and that was enough for the V1.0 of the device, further iterations of the device could occur if better tools came along.

Installing *chyves* and modifying the management shell scripts to work for more modern version of OpenBSD was successful and a solid and reliable platform was ready for use.

In V2.0, the appliance moved to FreeBSD 11.1, *vm-bhyve* replaced *chyves* and an all UEFI guest configuration. This approach simplified the number of datasets and the knowledge required to manage guests at the console level.

The V2.0 appliance then had acceptance of the team, some team members were able to perform zvol expansion for guests where required and others could manage guest operating systems with ease via a VNC console.

3.1 Storage and replication

OpenZFS allows for easy upgrades of guest operating environments by providing simple snapshots. The Copy on Write (COW) file system provided by OpenZFS is superb in this scenario, if guests don't get shut down prior to snapshots, rollbacks can occur to a previous running state without an issue.

Additional tools that were bundled into the build to automate backups, snapshots and transfers were *zfsnap2* and *zxfer*. While simple in their nature, they work well to ensure there is a consistent copy of each guest that can be rolled out to new devices in the event of a hardware failure of the production device. These backups are replicated to build servers around the state with hot spares of the appliance ready to go so they can be remotely re-imaged by the administration team and deployed by onsite ICT technicians.

Having guests imaged in the way mentioned above allows deployment of new or replacement devices within 15 minutes after power-on. Most of the delay is due to network speed for replication.

3.2 Guest network security

To mitigate the risk of guests seeing data that they are not entitled to on the wire, all VLANs were configured at the host level to the physical interface that either plugged into the switch or Internet device. Each guest then had the required interfaces presented to it and bridged to the applicable VLAN at the point of presentation. This avoided the need for configuration of VLANs within the guest and reduced the complexity for support staff.

4 OpenBSD VPN gateway

As part of the hypervisor appliance mentioned in section 3 of this document, a key feature needed was a feature rich router and firewall. OpenBSD was already providing a solid foundation to work off and was chosen for the task.

In addition to the firewall and routing capabilities of OpenBSD, the operating system also provided OpenIKED (IKEv2) in the base system, providing a simplistic and secure VPN tool to connect the remote offices up with the rest of the network, using a cheap and cost-effective public Internet link (figure 1).

Over the previous 18 months, different design concepts and configurations of OpenIKED were worked on to connect two medical centres to the core network, this helped iron out the final design for this virtual router.

Firstly, an OpenBSD VPN terminator was commissioned in the DMZ that end sites could connect to, forming part of the core network infrastructure. Setting this into a passive state and using key based authentication with VPN end points kept the configuration simple. Remote VPN hosts would be active clients and perform the connection to the main termination device.

The rest of the configuration was the same for both the main terminator and the clients. Each site was designed to have at least 5 VLANs for various tasks. The traffic then needed to be encapsulated, encrypted and signed for transport. Once end-to-end encryption was set up, a unicast virtual network using two virtual adaptors across the encapsulated link (using vxlan(4)) could be created that then could then have ripd(8) overlayed, managing the routes seen by either end.

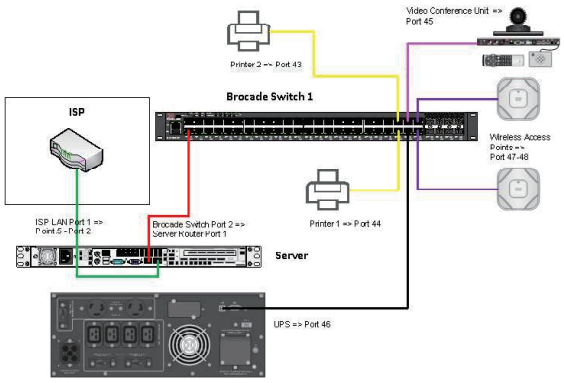


Figure 1

4.1 ipcomp(4)

The initial designed used the IP Payload Compression Protocol (ipcomp(4)) for all links to keep consistency as some remote sites have poorly performing ADSL connections providing as little as 6,000/768Kbs asymmetrical throughput. While this provided marginal improvements for these ADSL sites, more and more payload traffic are using encryption and/or transparent compression techniques and has made this no longer a benefit.

Before unwinding the use of ipcomp(4), kernel crashes were being observed in the main terminator that runs in the VMWare ESXi cluster when a significant and consistent traffic load had been passing through it. Further investigations found that this was an ESXi issue as it could not be replicated with bare metal hardware.

Another regression was also discovered between OpenBSD 6.3 and 6.4 that inadvertently forced the removal of ipcomp(4) from all links so devices could be upgraded to 6.4.

4.2 vxlan(4)

Configuring iked(8) to understand and handle traffic for a significant number of VLANs across all networks was not an option. A solution was required that would allow for the use of ripd(8) across the encrypted encapsulated link. OpenBSD comes with various encapsulation or tunnelling protocols such as gre(4), gif(4), etherip(4), however, vxlan(4) was chosen as it was a modern option and had scope for future expansion. As all traffic between the iked(8) end-points was encrypted, the vxlan(4) interface could be configured as a /30 subnet at either end of the tunnel to form a unicast network suitable for ripd(8).

By configuring ripd(8) on the vxlan(8) interface, both neighbours could see each other and pass their applicable routing tables.

4.3 pf(4) queues

While ipcomp(4) was deprecated, there was still a need to manage traffic moving between the remote and main networks.

All traffic that passes over an iked(8) connection moves through the enc(4) interface at either end. This interface does not support queue-based bandwidth control so traffic shaping cannot take place. Using pf(4) tagging allows the control of traffic flow between both end points but not the various traffic that uses this 'bridge'.

Since vxlan(4) was being used for ripd(8) and to route traffic between networks, pf(4) queuing was enabled on this interface at either end to ensure quality of service (QoS) was maintained for time sensitive, low latency packets. During the development and testing phase, it was found that no distortion occurred to Cisco Call Manager audio packets while the data links were under heavy congestion.

Two years on, queue and rule configurations produced in the initial rollout are still providing end-users with an excellent audio or video experience when they are either using handsets for regular voice calls or video conference units.

5 FreeBSD OpenZFS Storage Cluster

With the organisation growing rapidly over the last three years, ICT have had to constrain the type of data that utilised space on the flash storage array. Expanding it in the short term was not a viable option due to cost.

This saw some users within the organisation looking for options to store, low changing, static media that was significant in size. Marketing fell into this category due to the type of assets they worked on but there just wasn't the space for this type of media. When it was discovered that they were simply using a portable hard drive to store these assets on, it forced action to provide a facility for storage as well as the appropriate protection for this sort of data.

After discussions with vendors regarding the requirements and financial constraints, a hardware build was designed that was fully supported by FreeBSD 11.1 and its implementation of OpenZFS.

The design was simple, and it didn't need to be included it into the backup cluster if there was enough redundancy and protection built into the solution. By utilising the DR storage WAN links and other network links, a 3-node cluster was designed and built across three sites (figure 2).

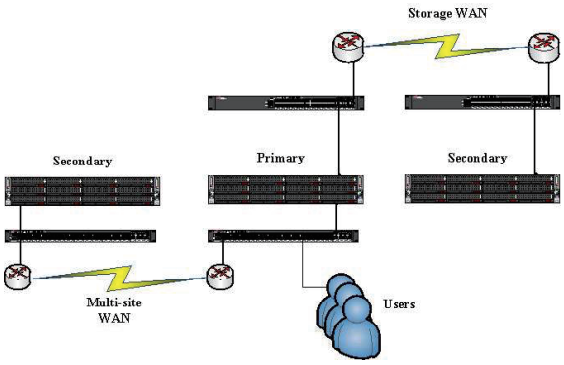


Figure 2

5.1 Storage

Deliberation occurred to determine if FreeNAS or vanilla FreeBSD 11.1 should be used for the project. As it was unknown how this storage would be utilised outside of the immediate scope, it was decided that FreeBSD would give greater flexibility moving forward as well as being able to customise it to meet the network and replication requirements.

The build consisted of Lenovo 2RU storage 3.5-inch storage chassis, 64GB RAM, a single Xeon CPU, 2 x 480GB high endurance SSD, 6 x 6TB HDD and a 2x10Gb Intel SFP+ network card.

The main storage was configured in a RAIDZ2 pool with 32GB of mirrored SLOG space dedicated from the 480GB SSDs. Only 200GB from the 480GB SSDs was given to the hosts operating system to allow for wear caused by the SLOG throughout the hosts life.

The ZFS ARC was limited to 48GB to allow for a minimal toolset to work correctly on the host and not put pressure on the available memory or swap if RAM was suddenly required.

The pool was carved up into many datasets depending on the work load and replication requirements. Initially this was minimal in the beginning, but it is now becoming more complex though is still quite easy to manage.

5.2 Replication

As the datasets are needed in the 3 locations without any special operations, *zxfer* was sufficient in managing the transfer of datasets and their associated zfs(8) properties. Data transfer was performed over *OpenSSH-portable* from the ports tree without any specific configuration except for key based authentication. While there are options and patches for tuning *OpenSSH* for large workloads, there were no perceived benefits in doing so for this use case.

At times, users can dump in excess of 100GB of files in a session. This does cause issues for us on the slower MPLS link (Multi-site WAN in figure 2) and cause a backlog of expired datasets that require removing. While not the most optimal solution, sneaker net is currently the best way to get the large dataset that is causing issues to the third site. Mounting a zpool(8) that is located on a portable hard disk, the dataset that needs to be replicated is copied to it using *zfs send/recv* and taken to the remote server by an ICT technician. The dataset in question is then copied to the third server, allowing for standard replication to resume as well as dataset clean up.

5.3 Services

Initially, only *smb* connectivity was provided for end users and *smb* used authentication from our main Active Directory (AD) infrastructure.

As time progressed, the internal mirror of application files was growing rapidly so this saw a jail serving out a mirror dataset via *nginx* be implemented. This worked so well that since it also contained the ISO files that were used for guest builds on our VMWare ESXi infrastructure, a read-only NFS share of the same dataset was created and presented as a data store to ESXi.

Eventually further storage was added to the main production environment. Scratch space was needed to move data around and to re-structure it. These storage servers were able to provide temporary iSCSI targets with zvols for storage presentation to the ESXi hosts making it easy to give redundant storage for *vmdk* moves.

6 Multiple routing tables

Without multiple routing tables, the replication discussed in section 5.2 could not occur over the DR storage WAN connection as the two hosts did not share the same layer 2 (L2) network segment. This is where the features of OpenBSD rdomain/rtable(4) and FreeBSD setfib(1) become essential in managing two isolated networks on the same host.

More than one routing table allows a system to host multiple applications on the same host that the attached networks or the applications are not aware of the other. For example, a host could run two vastly different web server instances for two different groups on the same host and neither group would know about the other instance except for the system administrator. Both groups would come in via two different network paths and their traffic would never cross.

This is what was used to connect hosts to the IP storage network that needed to see storage across multiple subnets (figure 2).

Another aspect is that it allows for more than one default router or route of last resort on a host so depending on the use case, an application can use the most optimal path.

Using this type of feature in either operating system is quite simplistic. In FreeBSD, invoking setfib <*table number*> <*command*> will run a command, launch a daemon or even adjust the applicable routing on the table defined in <table number>. OpenBSD has the feature built into route but the network stack can be modified independently through features in pf(4) and ifconfig(8). OpenBSD ping(8) and ps(1) are also both rdomain aware programs.

Out of the box, no additional configuration is required in OpenBSD, however FreeBSD requires the */boot/loader.conf* to contain *net.fibs="N"* with *N* being the number of tables your host needs.

7 Conclusion

A mix of BSD Unix variations have proven a valuable and successful asset to the LCHS organisation. The tools and technologies that are developed and maintained by FreeBSD and OpenBSD are on par with, or better than their commercial counterparts. Overall, adopting BSD has ensured that the technology managed within the organisation assists in providing better delivery of services for our clients.

bhyve - Improvements to Virtual Machine State Save and Restore

Darius Mihai
University POLITEHNICA of Bucharest
Splaiul Independenței 313, Bucharest, Romania, 060042
Email: dariusmihaim@gmail.com

Mihai Carabaș
University POLITEHNICA of Bucharest
Splaiul Independenței 313, Bucharest, Romania, 060042
Email: mihai.carabas@cs.pub.ro

***Abstract*—As more complex tasks are delegated to distributed servers, virtual machine hypervisors need to adapt and provide features that allow redundancy and load balancing. One such mechanism is the virtual machine save and restore through system snapshots. A snapshot should allow the complete restoration of the state that the virtual machine was in when the snapshot was created. Since the snapshot should encapsulate the entire state of the virtualized system, the guest system should not be able to differentiate between the moment a snapshot was created and the moment when the system was restored, regardless of how much real time has passed between the two events. This paper will present how the time management and block devices are saved and restored for *bhyve*, FreeBSD's virtual machine hypervisor.**

I. Introduction

Virtual machines can be used by extremely powerful systems (e.g., server farms) to more efficiently split resources between users, or run a variety of compatible operating systems without specifically installing one directly on the hardware system. These mechanisms are employed to reduce administrative complexity and better automate processes. Since a virtualized operating system is expected to still run as any other, tasks that depend on timer functionality and clock measurements are still expected to run with enough precision so results are not skewed over time.

This article refers to any system that allows running virtual machines as **host**, and the operating system running on virtualized hardware as **guest**. Since the host is responsible with managing the system resources, and therefore must not have its functionality hijacked by a badly behaving guest, a hypervisor (also known as virtual machine manager) is required to allow the guest systems access to hardware resources without impacting the host.

In some circumstances (e.g., the host will have to be stopped or if the host becomes over encumbered), users may want to stop the virtual machine, and potentially even move it to another system to continue work. This is achieved by creating a snapshot of the virtual machine and then restoring the virtual machine from the checkpoint.

A. Timers and Clocks

In order to perform periodic tasks, an operating system has to measure time in some manner, and, if the hardware permits it, request that an interrupt is sent when an interval has passed. Regardless of their exact function, a periodic task is a routine that will have to be called at (or as close as possible) a set interval.

For example, the basic Unix `sleep` command can be used to perform an operation every *N* seconds if `sleep $N` is called in a script loop. Since it is safe to assume that all modern processors have hardware timekeeping components implemented, sleep will request from the operating system that a software timer (i.e., one that is implemented by the operating system as an abstraction [1]) to be set for `N` seconds in the future and will yield the processor, without being rescheduled, until the system "wakes" it. The process "waking" is automatically done when the time has elapsed, and the process will be rescheduled to run by the operating system since it no longer waits for external events (in the case of sleep, it will usually simply end when the timer is done).

Considering that the virtual machine must not lose any functionality, time sensitive tasks should not be impacted either. More specifically, this implies that if a task ran in the virtual machine should end `N` seconds after it was created, it will expire `N-X` seconds after the virtual machine was restored from a checkpoint, if that checkpoint was created after `X` seconds.

Note that some tasks may be affected even when the virtual machine behaves as expected after a restore. For example, network communications may be timed out by the other end, even if from the guest's point of view the packets are expected to arrive well within the allotted time frame.

B. Block devices

A block device [2] is a permanent storage device that imposes access to data using a fixed width. A commonly used size is 512 bytes, equivalent to the length of a sector of a hard-disk drive (*HDD*). Reading or writing data to such devices requires sending requests of size that is a multiple of that size.

As with hardware systems, a virtual machine requires a virtualized block device as permanent storage to install the operating system, as well as other software. Similar to other virtual machine managers, *bhyve* [3] uses special files stored on a physical drive that act as storage medium for the virtual machine. The files are kept open by a part of bhyve and read/writes are performed in these files in stead of directly

on the block device (i.e., the file will ultimately be stored on a hardware device, using the abstraction layers of the FreeBSD kernel).

At the time this paper is written, bhyve only supports raw disks (i.e., data is stored without being compressed, and no extra features, such as copy-on-write, are available).

As with timers and clocks, the state of block devices needs to be correctly saved to avoid the corruption of the file systems used by the guest. Failing to do so, either by losing some requests received from the guest, or failing to send notifications when the operations have finished may result in the guest having an inconsistent view of the data on the disk, or corrupted data. Both scenarios can lead to damage to data and systems with bad behavior.

The following sections are split as follows:

- **section II** talks about how the timers and time measuring circuits work, how they are virtualized, and the previous state of the save and restore mechanism.
- **section III** presents how AHCI compatible block devices are virtualized
- **section Iv** shows the state of the overall save and restore feature of bhyve
- **section V** focuses on the improvements to the snapshot mechanism our work has achieved.
- **section VI** shows the results obtained after successfully restoring time components and virtualized AHCI compatible block devices on systems with Intel processors.
- **section VII** draws some conclusions and talks about future work on the save and restore feature for bhyve [3].

II. Time Management Virtualization

Timers are hardware resources used by the operating system for synchronization. Usually, the timers have internal counters that are incremented or decremented to measure time passage. Depending on how often the counter value is changed timers have a measurable *resolution*. The resolution is a measure of how good a timer is, and more precise timers are commonly more used when available, and unless tasks only need a rough estimation of time.

Clocks are time measuring circuits that can be used by an operating system to precisely measure wall clock time (i.e., real time). A clock uses an internal clock counter that is incremented by a monotonic clock signal. Using this value, and knowing how often it is incremented (i.e., the clock frequency), software can measure how much time has passed since the clock was reset.

In bhyve [3], the following timers and clocks are of particular interest:

- Local Advanced Programmable Interface Controller [4] (known as **Local APIC**, or **LAPIC**).
- High Precision Event Timer [5] (known as **HPET**).
- Time Stamp Counter [4] (known as **TSC**).

A. Timer: LAPIC

The Local APIC described in the *Intel 64 and IA-32 Architectures Software Developer's Manual Vol. 3A* [4], Chapter 10, is a per-CPU programmable interrupt controller (i.e., aggregates multiple sources of interrupt and delivers them to a specific CPU core) that is equipped with an internal timer.

The LAPIC's timer is a software programmable 32-bit timer that can be used to time events and system operations. The timer settings are split among four registers:

- Divide Configuration Register.
- LVT Timer Register.
- Initial Count Register.
- Current Count Register.

The Divide Configuration Register is used to configure the rate at which the current counter value will be decremented. Possible divider values range between 1 and 128 (inclusively), in powers of two. The set value is used by a circuit to reduce the processor's bus clock by the set factor (e.g., the timer's frequency will be half that of the bus clock, when the divider is set to use a factor of 2).

The LVT Timer Register can be used to set the timer in either periodic, or one-shot mode. Periodic mode means that the timer will rearm itself after expiring. In one-shot mode the timer will remain stopped after firing once. The register also determines which interrupt will be delivered to the processor.

Current Count Register and Initial Count Register are used together. When the Initial Count Register is set, its value will be copied to the Current Count Register. At every clock cycle of the timer, the value in the Current Count Register is decremented. When the value reaches zero, an interrupt is sent to the processor. If the timer is set to run in periodic mode, the value of the Initial Count Register is copied again and the cycle restarts. Setting the Initial Count Register to 0 will stop the timer. Figure 1 shows the workflow of the LAPIC timer.

To virtualize the Local APIC, bhyve traps access to the memory mapping of the device and updates the internal device state if required. Whenever the timer Initial Count Register is programmed, or a periodic timer expires, a `callout` [6] is set by the virtual machine manager, based on its internal state (i.e., the values of the four aforementioned registers) to program a timer on the host. When the timer expires, an interrupt is sent to the guest to signal that the time is up. In the case of periodic timers, another timer is set, and the cycle repeats.

Timer deadline for `callout` is computed as a function of system uptime, as required by the interface. More specifically, the timer frequency is computed as the fraction between the predefined value of frequency and the value of the divider. Since the timer is programmed using relative time (i.e., how much time should pass since it is programmed until it expires), when the Initial Count Register is set, simply adding the current system uptime with the product of timer frequency and the set counter will give the time when the timer should expire.

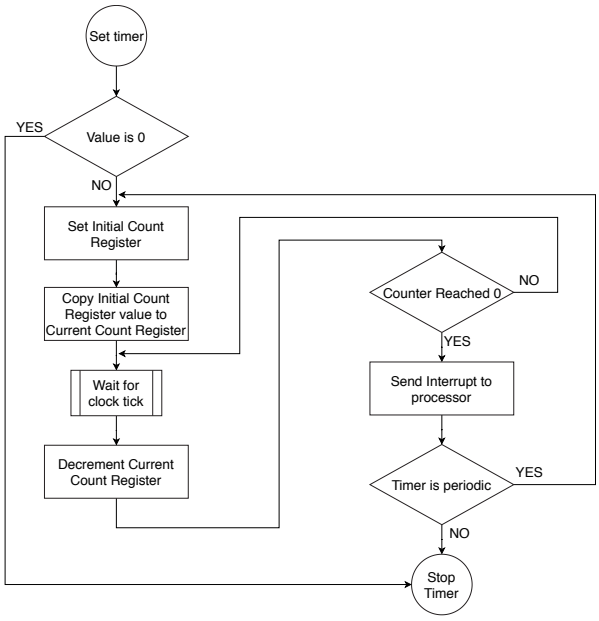


Fig. 1. Local APIC timer workflow.

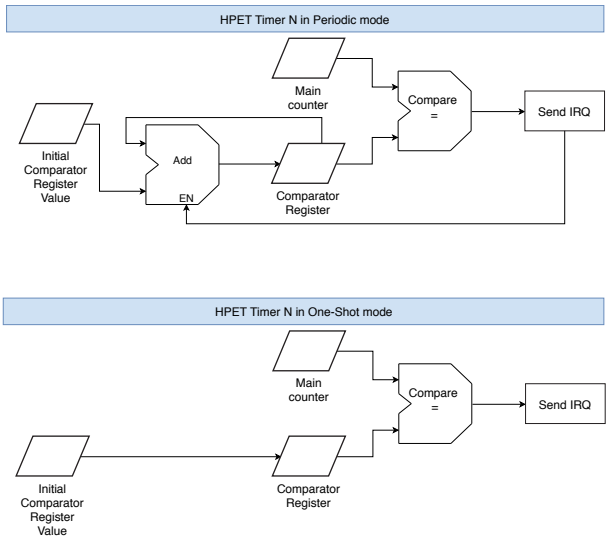


Fig. 2. HPET timer logic.

B. Timer: HPET

The High Precision Event Timers [5] are hardware timers developed by Intel for their processors. The timers use a 64-bit main counter that is incremented and an implementation-defined number of timers, with a minimum of 3.

Functionally, the timers use comparators to see when the main counter has reached a certain value. The value used for comparison is stored in a "match" register that can be either 32 or 64-bit wide. The value of the main counter is compared with the reference value using N-bit (with N being either 32 or 64, depending on the implementation) comparators and whenever the value compared matches the value of the value of the counter, an interrupt is generated.

The timers can function in both one-shot and periodic modes. In periodic mode, the comparator value is set to `value(base_counter) + value(comparator_register)`, so a new interrupt is sent every `value(comparator_register)` ticks.

The frequency of the HPET timer can be much lower than that of the LAPIC (i.e., the specification document [5] imposes that it should be at least 10MHz, while the LAPIC runs at the same frequency as the processor, unless divided), so it is less precise.

Since the HPET has the main counter, a monotonically incremented counter value, it can also be used as a rougher granularity clock source.

Figure 2 is a representation of the logic used to implement one of the timers in HPET.

HPET virtualization relies on device memory mapping to identify register access and update internal device state. HPET timers are emulated using `callout` [6] structures, one for each timer. The guest can set and get values for the main counter and timer counters. For virtualization purposes, the value of the main counter is set once, and reading it will simply add the value set by the guest and add it to the time elapsed since it was set divided by the frequency. Using this mechanism, the value of the counter can be precisely computed without using periodic timers to increment it.

Each timer is virtualized by programming a callback structure to the time when it should expire. However, since the timers programming relies on absolute values, rather than relative, the "current" value of the main counter will have to be determined and subtracted from the value of the comparator to obtain a relative time in the future when the timer must expire. Similar to the way LAPIC timers are set, the moment in the future the timer will expire is computed as the sum between the current time and the number of relative "ticks" of the HPET timer multiplied by the its frequency.

For periodic timers, after the timer interrupt is sent to the guest, another `callout` will be set to expire after the another time period has passed.

C. Clock: TSC

The Time Stamp Counter, as described in Section 17.15 of the *Intel 64 and IA-32 Architectures Software Developer's Manual Vol. 3A* [4] is a per-CPU internal counter that is incremented at the same rate, regardless of CPU frequency changes. The constant rate means that TSC can be used as a wall clock timer (i.e., measures real time, as opposed to how much a process has been running on the CPU). It is possible to adjust the value of the Time Stamp Counter using the `wrmsr` special instruction with appropriate offset.

The value of TSC can be read using the `rdtsc` instruction without requiring special privileges in the operating system (e.g., for Unix systems the command will not run as `root`).

Using this mechanism, software can determine how much time has passed from a reference point by computing the absolute difference of the two values.

An extension to TSC, called Invariant TSC, will guarantee that the value of the TSC counter will continue to increment while the system moves to power saving states. However, this behavior is not supported on older CPUs, so TSC may not be as stable as HPET on all systems.

TSC virtualization relies on the hardware extensions provided by modern Intel and AMD processors. This article refers to Intel specific extensions, but AMD offers very similar functionality.

Since the value of TSC is directly provided by the processor, its value is shared between host and guest. Because of this, the guest cannot be allowed to directly change its value; in stead, the virtualization extensions provide two special registers: TSC offset and TSC multiplier, as described in Section 25.3 of the *Intel 64 and IA-32 Architectures Software Developer's Manual Vol. 3A* [4].

Considering that bhyve does not use the TSC multiplier, the TSC offset is used when the guest attempts to access the TSC register as follows:

- **write** - the TSC offset is set to the difference between the value desired by the guest and the current host TSC value.
- **read** - the sum between the value of the TSC offset register and the current host TSC is returned.

III. Block Devices Virtualization

The Advanced Host Control Interface (AHCI) [7] is a PCI class device used to transfer data between system memory and SATA devices. Using the AHCI device, the system can enqueue multiple requests to a single device, with the possibility to aggregate requests to reduce disk wear and improve performance.

AHCI can connect multiple hardware devices to the system CPU, offloading the CPU workload through DMA transfers. By offloading the AHCI device, the system can asynchronously send transfer commands to I/O devices. At most 32 ports (i.e., physical connections with other devices; more than one device can be connected to a single port using port multipliers). Each port must function independently from one another.

By intercepting accesses to AHCI through memory mapping, bhyve [3] is able to identify when the guest is attempting to access the virtual disk. The virtual machine monitor can intercept guest memory accesses to detect when the guest attempts to program the AHCI controller for data transfers. Since requests are in a standard format the host can then decode the requests sent by the guests, by interpreting the commands sent.

Device virtualization implements asynchronous operation by offloading requests to additional worker threads with common work queues. Currently, bhyve uses eight workers in a generic block interface shared with the VirtIO [8] block device.

The device is emulated by translating the interpreted commands into I/O requests that can be executed on the host side. To emulate the AHCI's multiple command capabilities, reads and writes can be combined using FreeBSD's `readv` [9] and `writev` [10] system calls. Through the use of I/O vectors (as described by the manual pages of both `readv` and `writev`), numerous data transfers between the disk and guest memory can be executed using single system call, reducing device emulation complexity.

It is worth mentioning that using I/O vectors is only possible because the guest's memory is mapped directly into host memory space, such that data access to a specific memory address will be seen by the guest as if the device has completed the requests.

IV. Related Work

Bhyve [3] already featured a partially functional save and restore feature capable of resuming a guest running FreeBSD from a snapshot. The guest ran using VirtIO [8] network and block devices, and tests showed that it was able to connect to the internet, continue running timers, and read data on the disk after being restored. Obviously, testing advanced features like timers, network, and disks, would not have been possible if the guest CPU and memory were not properly restored before.

A regular test script to check if the internet connection and timers work correctly is shown in listing 1. If the script continued running properly after restoring the virtual machine, both the network connection and timers would have to work properly. Since `sleep` relies on a timer to finish its work (and so the contents of the `while` loop would execute), general timer misbehaviour would easily be spotted, since the command would not finish in time.

```
while sleep 1; do
  ping -c 1 8.8.8.8
done
```

Listing 1. Network and Timers testing.

To test disk functionality, simply being able to read a file from the virtual disk without any errors, and without visible data corruption was considered enough.

Since the tests performed on a FreeBSD guest were usually successful, the devices were considered functional. However, after changing test parameters, such as the guest operating system and attempting to restore after the host was restarted, a number of issues started to become apparent.

Linux and Windows, as opposed to FreeBSD do not communicate with the host directly through a serial console, but in stead use a frame buffer where they output a graphical interface, seen as either a classical CLI or GUI, and receive input from emulated xhci (i.e., USB) mouse and PS/2 keyboard. To properly communicate with the guest, these interfaces (frame buffer, xhci and PS/2) also had to be saved and restored, but their implementation is outside the scope of this paper and will be considered as de-facto functional.

The Linux guest had mostly the same functionality as the FreeBSD guest, but when running `dmesg` after a VM restore,

multiple filesystem operation errors on log files were displayed. This showed that despite being seemingly functional for files that were unchanged near the time a checkpoint was created, frequent background file I/O lead to data corruption. By using the same disk image multiple times to suspend and restore the virtual machine (as a special case of snapshot, that stops the guest after the snapshot is taken), after a few (around 4-5) iterations the disk usually became corrupted enough to render the kernel or core utils binaries unusable.

Improper functionality of the block devices was caused by how handling of incomplete requests was done. Before the snapshot of the virtual machine is created, the guest CPUs are frozen, and thus any interaction such as notifications that disk operations have finished were lost.

Windows guests running Windows Server 2016 (both with, or without, the full GUI) were completely frozen after restore. The issue with Windows, as we have assessed it, is that the user interface is directly linked to the system functionality, so if the interface is unable to properly update, applications with any form of interface would also stop working.

Moreover, a seemingly inconsequential difference, rebooting the host, would cause a kernel assertion to fail, and so the host system would crash whenever trying to restore a virtual machine. This was caused by the fact that the virtualization of some devices (e.g., HPET) uses the `callout` interface which relies on system uptime. Attempting to restore the value of the device state variables that kept track of system uptime usually meant that they would usually be reset to an "earlier" time (i.e., the host system at restore time had less uptime than it had when the snapshot was created), and an assertion meant to make sure the timer expiry date would not go backwards would fail, crashing the host.

The referenced behavior can be seen in listing 2, where the `KASSERT` instruction fails. `vhpet->countbase_sbt` is a variable set when the device emulation starts to the (then) current system uptime. Simply restoring it to its previous value can mean that its value is higher than the value of `now` (actual system uptime), thus resulting in a negative value of `delta`.

```
val = vhpet->countbase;
if (vhpet_counter_enabled(vhpet)) {
  now = sbinuptime();
  delta = now - vhpet->countbase_sbt;

  KASSERT(delta >= 0, ("vhpet_counter:"
    " uptime went backwards: "
    "%#lx to %#lx",
    vhpet->countbase_sbt, now));
  val += delta / vhpet->freq_sbt;
  if (nowptr != NULL)
      *nowptr = now;
}
```

Listing 2. HPET Virtualization Assert.

V. Save and Restore Debugging & Improvements

A. Time Management

Before being able to debug the more complicated issues with bhyve, the issue with the HPET virtualization that was shown in section IV had to be addressed.

Please note that issues described in this section refer to the manner a virtual machine behaved when restoring its state shortly after a host system reboot, when the system uptime is small. The virtual machine in all test cases is suspended when the snapshot is created, instead of having it continue running, since the disk currently does not support any copy-on-write mechanisms (e.g., qcow2, or similar).

Since the reference value used by HPET could not be restored as-is, if the timer was enabled prior to the virtual machine snapshot save, it would be reset using the current system uptime. While this did not solve any issues related to the guest's stability, it did stop the virtual machine from crashing the host, and allowed us to further guest behavior inspection.

While the host would no longer crash, the guest operating system did not work as expected: the `sleep 1` command did not finish after one second as expected, but rather after a seemingly random interval (the exact amount of time was not always the same). Moreover, trying to read the system time using `date` in this interval always returned the same value regardless of how much time had passed since the guest was restored. The reason for the inconsistent intervals after which the guest would resume functioning was caused by the fact that its timers were not restored.

As described in section II, the Local APIC virtualization relies on the `callout` [6] system to set a timer to expire Initial Count Register units (multiple of a base clock) into the future. At any point, a non-virtualized LAPIC keeps track of the amount of time until the timer must expire using the Current Count Register. Consequently, the snapshot mechanism saves the value computed for the CCR when virtual machine state is committed to disk (i.e., to not keep a separate counter that is decremented at a set frequency, the value is computed as a function of the value of the ICR and how much time has passed since the timer was set). When restoring the virtual machine, the timers that were set before the snapshot was created are reprogrammed using the value of the previous CCR.

Correctly saving and restoring the LAPIC timers resulted in a more stable, albeit incorrect, guest functionality. To be more precise, the system uptime still refused to update, but the intervals became more stable. Since the length of such intervals seemed close to the amount of time the host ran before the snapshot was created, the Time Stamp Counter and HPET were approached.

For the virtual machine, unless explicitly offset, the perceived value of the counter is the same as the one on the physical host. This means that if the virtual machine is restored after a system reboot, the value of the counter it reads may be smaller compared to a value it read before the snapshot

was created, because the reset value for TSC is 0. In turn, this implies that trying to determine how much time has passed by subtraction, using a value read before the snapshot, and one read after the restore would result in integer underflow (i.e., the value of TSC is a 64-bit unsigned integer).

TSC virtualization is done entirely by the virtualization extensions provided by the CPU manufacturer, and relies on their specific implementation. As such, the save and restore feature was added as a CPU-specific functionality, currently implemented for Intel CPUs only.

To keep track of both the offset set by the guest and the one required to compensate for system uptime differences between save and restore, two offset variables have been added for each virtual CPU: `guest_off` and `restore_off`.

The `guest_off` variable, which is used to keep track of the offset imposed by the guest is set when the `wrmsr` instruction with the proper offset is intercepted.

For the second offset variable, `restore_off`, the value of the counter is saved when the snapshot is created, and the offset is computed by subtracting the value of the counter at restore time from the value that was previously saved. If a new snapshot is created from a restored virtual machine (i.e., it runs as a result of the restore operation), the values of the restore offsets are added together.

By adding the values of `guest_off` and `restore_off`, and setting the result as offset for the Time Stamp Counter, the guest is not affected by the snapshot operation.

Despite no longer being stuck reading the same value for the time / date, the guest did not run as expected. When the guest finished restoring, a message could be seen in `dmesg` saying that TSC was an deemed as unstable and replaced with HPET.

Linux uses different counters as clocksources, including the HPET main counter and TSC. Since TSC is considered a precise clocksource, it is usually preferred over the more coarse alternatives (e.g., HPET). However, since the behavior of TSC is not guaranteed to be the same on all systems, less precise, but stable clocks are used to check if TSC has deviated, or not. If intervals measured using the two clocksources differ by an amount greater than a threshold, the next best source is selected to replace TSC.

In this case, however, TSC was the clock running properly, while HPET was incorrectly restored. To snapshot HPET, the value of the main counter is saved and used as offset after the restore, and added to the value of the current counter.

B. Block Devices

To solve the issues with losing notifications by sending them to a frozen host, when a snapshot request is processed, the emulated devices are paused before freezing the guest CPUs. Pausing is handled by the same thread that handles the snapshot, so synchronization with the worker threads is done to ensure that the worker threads will not undertake any more work while the device is paused, and the pause functionality will not end until all workers have been paused.

By pausing the devices while the virtual machine's CPUs are unfrozen means that the guest will not miss notifications for completed requests.

Furthermore, because the workers are unable to complete any tasks while the device is paused, new requests sent by the guest are also saved and restored as part of the snapshot process.

VI. Results

The stability of guests has been greatly improved as a result of fixing the issues of time management and disk virtualization.

Since the virtual disk is only implemented to use raw disks, and no mechanism to take disk snapshots is in place, the virtual machine tests are only performed on suspend and restore scenarios. If the guest would instead be allowed to run after the snapshot is created, the state of the virtual machine disk would not be the same as when the snapshot was created.

A. Linux Guest

The Linux guest performs properly after a restore in the following scenarios, where the virtual machine was suspended and restored while the respective operation is underway:

- sleep for a predefined interval.
- copy large files in the guest.
- copy files from a guest NFS shared directory to the host.

The sleep test scenario is meant to prove that the errors with timekeeping have been solved. This is the simplest test, since it does not rely on any other device functionality to be done - the binaries do not need to be read from disk after the restored since (if the virtual machine has enough memory) they are mapped into memory. Also, other devices like network are also not required to work since all operations are local.

Creating a copy of a larger file in the guest tests the virtual disk save and restore. As an example, a 4GB file with random data is created using the command in listing 3.

```
dd if=/dev/urandom of=test.ref bs=1G \
   count=4
```

Listing 3. Creating a file with random data

The testing of the disk can only be currently done using AHCI virtualization, since the VirtIO block device does not have the pause and resume functionality implemented. If the virtual machine is suspended while the copy operation is underway and the disk loses any requests, the copy of the file differs from the original. An observed result is that if the requests are not restored, some chunks of the file will only contain bytes of value 0.

The last experiment tests mostly timers and the network. Since the NFS protocol accepts a large enough system down-time from any of the ends, the guest can be suspended while a file is copied from the guest to the host, and the operation must resume when the guest is restored. As is the case with the other file test, large files with random contents are used to assess performance because any inconsistencies are easily spotted.

As a side note, the log file errors seen in `dmesg` described in section IV no longer showed after safely restoring the disk.

B. Windows Guest

For the Windows guest the following test scenarios have been considered:

- interaction with the guest is possible after restore.
- sleep for a predefined interval.
- copy files from a guest samba (SMB) shared directory to the host.

The first point should, obviously, be a prerequisite for testing any other advanced functionality, but the graphical user interface was completely frozen after a restore, due to errors in the way time management was saved and restored. Since Windows did not provide much information about the reason it didn't work, all debugging was done on Linux, and thus, Windows was simply a "bonus" that confirmed that timers and clocks are working.

For the second and third scenarios, the implications and testing methodologies are similar to those presented for Linux, with the exception that SMB has a smaller window for system downtime. As a result, `rsync` may show an error when the transfer reaches 100%, but the copied file proves to be identical to the source.

VII. Conclusions and Future Work

The virtual machines are more stable, and all three operating systems of interest - FreeBSD, Linux and Windows are able to run. However, a number of issues will be addressed:

- separate snapshot and migration code - the current implementation has mixed code for virtual machine snapshots and warm migration (i.e., moving the saved data from one host to another through a socket).
- the offset for TSC is unnecessarily split between two variables - in case the guest sets the offset explicitly, the offset should be computed relative to the current value of TSC on the host, and discard the restore offset entirely.
- the offset used by HPET should be set to 0 when the guest explicitly sets the value of the counter - the set value should be used as the counter value, regardless of whether the guest runs as a result of a restore or not.
- refactor the snapshot code - the generic code currently iterates through all devices to check if it is used, so it will be snapshot; additionally, most of the device-specific code for save and restore is the same (i.e., fields are saved one-by-one and in the same order, so the difference lies in whether the snapshot buffer is written or read from).
- extend the block device pause and resume functionality to the VirtIO block device
- add suport for TSC on AMD CPUs

Acknowledgments

A special "thank you" to Sergiu Weisz and Elena Mihailescu for their continued support in implementing the code and debugging all issues that we have encountered.

We would like to thank Marcelo Araujo and Matthew Grooms for their insight while debugging and structuring the code.

We would also like to thank iXsystems and Matthew Grooms for their financial support.

References

[1] Daniel P. Bovet and Marco Cesati. "Timing Measurements," in *Understanding the Linux Kernel*, 3rd ed., O'reilly & Associates Inc., 2009

[2] Jonathan Corbet, Alessasndro Rubini and Greg Kroah-Hartman. "Block Drivers," in *Linux Device Drivers*, 3rd ed., O'reilly & Associates Inc., 2005

[3] FreeBSD Project. *Freebsd as a Host with bhyve*. [Online]. Available: https://www.freebsd.org/doc/handbook/virtualization-host-bhyve.html [Last accessed December 18th, 2018].

[4] Intel Corporation. *Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3A*. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html [Last accessed November 25th, 2018].

[5] Intel Corporation. *Intel IA-PC HPET (High Precision Event Timers) Specification*. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf [Last accessed December 5th, 2018].

[6] FreeBSD Project. *FreeBSD Kernel Developer's Manual - callout*. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=callout&manpath=FreeBSD-12.0-RELEASE&arch=default&format=html [Last Accessed January 1st, 2019].

[7] Intel Corporation. *Serial ATA Advanced Host Controller Interface (AHCI) Rev. 1.3.1*. [Online]. Available: https://www.intel.com/content/www/us/en/io/serial-ata/serial-ata-ahci-spec-rev1-3-1.html [Last Accessed January 1st, 2019].

[8] OASIS Committee Specification 04. *Virtual I/O Device (VIRTIO) Version 1.0*. Edited by Rusty Russel, Michael S. Tsirkin, Cornelia Huck, and Pawel Moll. 03 March 2016. [Online]. Available: http://docs.oasis-open.org/virtio/virtio/v1.0/csprd01/virtio-v1.0-csprd01.html [Last accessed January 2nd, 2019].

[9] FreeBSD Project. *FreeBSD System Calls Manual - readv*. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=readv&manpath=FreeBSD-12.0-RELEASE&arch=amd64&format=html [Last accessed January 10th, 2019].

[10] FreeBSD Project. *FreeBSD System Calls Manual - writev*. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=writev&manpath=FreeBSD-12.0-RELEASE&arch=amd64&format=html [Last accessed January 10th, 2019].

FreeBSD - Live Migration feature for bhyve

Maria-Elena Mihăilescu
University POLITEHNICA of Bucharest
Splaiul Independenței 313, Bucharest, Romania, 060042
Email: elenamihailescu22@gmail.com

Mihai Carabaș
University POLITEHNICA of Bucharest
Splaiul Independenței 313, Bucharest, Romania, 060042
Email: mihai.carabas@cs.pub.ro

***Abstract*—When talking about servers and clouds, live migration is one of the most powerful tools that can be used to manage resources that are abstracted by virtual machines due to its small downtime. bhyve, FreeBSD's hypervisor, does not have a live migration feature implemented yet, even though it is a very useful feature for a hypervisor.**

This paper presents two approaches for implementing a live migration feature for bhyve that use the FreeBSD's virtual memory subsystem. The first one uses a Copy-on-Write mechanism that cannot be implemented due to bhyve memory layout, and the second one uses a dirty page detection mechanism.

I. Introduction

Cluster and grid solutions have become more important each day, whether we talk about web servers or data centers. The cluster and grid framework usually offers resources for the clients by providing them access to certain virtual machines that abstract the hardware resources.

The virtual machine migration is a powerful tool that is used for load balancing or as a method to avoid data loss when one of the cluster's systems may become inaccessible in the near future (e.g., partial hardware failure, the need to upgrade the infrastructure). The migration process may be automated or may be done manually by the system administrator.

One of the migration's challenges is related to the guest's downtime: the more memory a guest has assigned, the more it may take for the migration process to finish. One of the fastest ways of migrating a virtual machine is by using the live migration procedure and migrate a guest from one host to another while it is still running.

bhyve [1] is a type 2 hypervisor implemented in the FreeBSD operating system and can be used on Intel and AMD CPUs systems that have support for virtualization. Linux, FreeBSD and Windows are some of the guest operating systems that can run in a virtual machine created with bhyve.

Unlike hypervisors such as VirtualBox, Xen, Hyper-V, VMWare ESX, and KVM that have a live migration feature, bhyve does not have one, even if it is necessary. In this paper, we present a Copy-on-Write mechanism that can be used to detect memory changes between live migration rounds, but that cannot be used by bhyve due to a dual memory layout implementation. Also, we propose a live migration feature for bhyve that uses a dirty-bit mechanism to detect the memory changes, and a save and restore mechanism feature [2] developed for bhyve to migrate the guest CPU and devices state.

This paper is split in eight sections. In Section II, we will present some of the main concepts that are used to develop the live migration feature for bhyve and for finding memory differences between migration rounds. In Section III, we will present a guest state save and restore mechanism and a cold and a warm migration feature for bhyve that is based on the save-restore procedure and that will be used in the live migration development process. In Section IV, we will show a Copy-on-Write approach for live migrating a guest memory that led to the current implementation and the reason it cannot be used in bhyve. In Section VI, we will suggest an algorithm that is based on the dirty-bit mechanism to detect memory changes. In the fifth section, the current status of the project along with its results is presented. In Section VII, we will present the future work that should be done to allow this project to evolve. In the last section, we will draw some conclusions for this paper.

II. State of the Art

A. FreeBSD's Virtual Memory Subsystem

The Virtual Memory Subsystem is one of the most important parts of an operating system since it manages the relationship between the physical memory and processes. This subsystem creates an abstraction layer between the software and the hardware so that a process can see a contiguous memory allocation space. Moreover, it ensures a level of security since a process cannot access another process's memory if the access was not granted using special mechanisms such as shared memory.

In the FreeBSD operating system, the virtual memory subsystem is object oriented and has four main components that are used to abstract the physical memory:

- `struct vm_page` – is the smallest virtual memory representation entity and represents a virtual page. It is mapped one-to-one with a physical memory page.
- `struct vm_object` – is a collection of `struct vm_page` entities that have the same characteristics. A `struct vm_object` entity represents an allocated area of contiguous memory.
- `struct vm_map_entry` – is an entry into an address map (represented by a `struct vm_map` object) that place an `struct vm_object` entity or another address map between a start address and an end address into a process's address space.

- `struct vmspace` – is an entity that represents the process virtual address space and points to a `struct vm_map` that contains a list of `struct vm_map_entry` entities. Moreover, it contains a link to the physical page table (`struct pmap`) for the represented process.

A `struct vmspace` entity is associated with each new process that is created on the system. This entity contains both a virtual memory mapping of the virtual pages and a reference to the physical page table. For each of the process's contiguous memory regions with the same characteristics (i.e., same permissions and flags) a `struct vm_map_entry` entity and a `struct vm_object` entity are created.

The Copy-on-Write (CoW) mechanism is used to optimize the system's memory usage and to rapidly create a new process when the `fork()` function is called. When `fork()` is called, the parent process' memory is marked as copy-on-write which means that the parent process and the child process share the same memory pages until one of them tries to modify one of the pages. Then, a page fault is triggered and the page is duplicated such that each of the two processes have an individual copy.

In FreeBSD, the Copy-on-Write mechanism is implemented using shadow objects [3]. A shadow object is a `struct vm_object` entity that is backed by another `struct vm_object` entity. It may happen that the backing objects can be another shadow object creating a list of shadow objects.

When a page from the shadow object is accessed, the page is first searched in the shadow object and if it is not found there, the page is searched in the backing object list. If the page resides in a backing object and is accessed for a write operation, then a copy of that page is added to the shadow object. If the page is accessed for a read operation, the object's layout is not modified. In the case of a `fork` call, an object is shadowed by two new objects: one for the parent and another one for the child.

B. Guest Memory Management

In a bhyve, the guest address space is split into three main components [4]:

- lowmem segment – this is the memory segment that is mapped between 0GB and lowmem limit size which is set at 3GB [5] at the time this paper was written. If the guest assigned memory size is smaller than the lowmem limit value, then the segment size is equal to the guest memory size. Otherwise, it is equal to the lowmem limit value.
- PCI hole – this is a non-mapped memory region between lowmem limit and 4GB (currently between 3GB and 4GB) that is used to access the devices through Memory-Mapped I/O (MMIO).
- highmem segment – this is the memory segment that is mapped starting from 4GB. This segment is equal in size to the difference between the guest assigned memory size and lowmem limit value, and it may not exist if the guest memory size is smaller than the lowmem limit value.

In bhyve, the guest memory is allocated during initial setup [5], where the user-space utility, `bhyve`, maps a contiguous area that is then divided between the lowmem segment and the highmem segment. Each of the two segments will have a new object assigned in the bhyve user-space process address space (a new `struct vm_map_entry` entry in the process's `struct vmspace` that will indicate the user-land bhyve process address range in which the segment was mapped). Then, in the kernel-space, the bhyve hypervisor, using the architecture dependent implementation for the `vmspace_alloc` function from the `struct vmm_ops` entity (a wrapper of architecture dependent functions), creates a new `struct vmspace` entity that will eventually point to the same memory objects that are allocated for the lowmem and highmem segments. However, the two `struct vmspace` entities (the one for the bhyve user-land utility, and the one that is allocated in the kernel) have different virtual and physical mappings, the latter corresponding to the guest address space layout that was previously discussed.

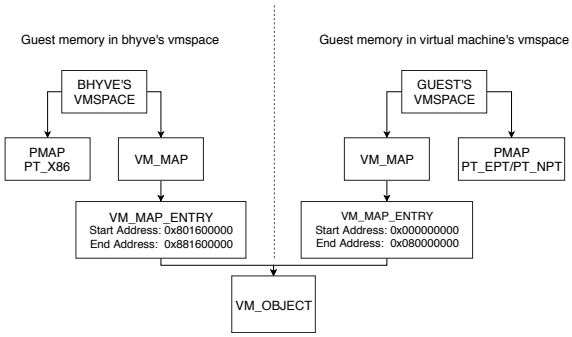


Fig. 1. Dual Guest Memory View - lowmem segment

In Fig. 1 the dual guest memory view previously described for a bhyve virtual machine that has assigned 2GB of memory is represented. The left side represents the guest memory as it is mapped by the bhyve user-space tool and the right side is the guest memory representation as it is seen by the virtual machine itself. The same object that contains the guest memory pages is referred by two `struct vm_map_entry` entities. Each of the two `struct vmspace` entities have a link to a `struct pmap`. Since the nesting paging feature was introduced in bhyve [4], each physical mapping for the amd64 architecture has a mapping of type `PT_x86` for normal mapping, or `PT_EPT` (for Intel Extended Page Table feature), and `PT_NPT` (for AMD Nested Page Table feature) for guest memory mappings.

The dual guest memory view represents a communication mechanism between the host and a virtual machine. User-space emulated devices (e.g., virtio, ahci, e1000) are running in different threads and receive and fulfill requests from a guest (e.g., reading or writing data on disk, receiving and sending network packages).

The left side view from Fig. 1 displays what happens when a request to read from disk is received by the host from the

guest: the thread that emulates the disk interface from host user-space (e.g, virtio-block, ahci) is reading the data from disk and updates the guest memory by writing directly to it. The right side view from Fig. 1 shows how the enities are used when the guest accesses its memory and the information is directly read from the guest's `struct vmspace` entity.

C. Virtual Machine Migration

The virtual machine migration mechanism allows a user to move a guest from one host to another. From literature [6] [7] [8] [9] [10], the migration techniques can be divided into two main categories:

- Non-live Migration - the guest is powered off or suspended at migration time.
- Live Migration - the guest is running during the migration process.

The **Non-Live Migration** technique is divided in two main categories as well: cold migration and warm migration [7] [6]. The cold migration implies that the guest is powered off and all its data (disk and auxiliary files) are moved to another system. The warm migration procedure implies that a guest is suspended, its state copied onto the destination, and then the guest is resumed from the saved state.

Whereas in the cold migration case there are no restrictions regarding the guest's disk (because it is copied from one system to another), for the warm migration procedure, the disk image must be shared between the source and destination hosts. In terms of performance, the warm migration technique is faster than cold migration because of the fact that the disk is shared among systems.

The **Live Migration** technique has the best results in terms of migrated guest's downtime because the virtual machine is migrated while the guest is still running. The live migration procedure [8] [9] [10] has two phases: a phase in which the guest memory is migrated in rounds while the virtual machine is still running, and a phase in which the guest is stopped and the CPU's and devices' state are migrated to the destination.

Based on the method that is used in order to live migrate the guest's memory, there are two types of live migration [8] [9]:

- Pre-Copy Live Migration [8] – The memory migration is done in rounds. In the first round, all the guest pages are copied to the destination. For each of the following rounds, only the pages that were written between two rounds are copied to the destination. After a number of memory migration rounds or when a threshold number of dirty pages is reached, the virtual machine is stopped and the remaining dirty pages, together with the CPU's and devices' state is transferred to the destination host and the guest is started.
- Post-Copy Live Migration [9] – The memory is migrated using a page-fault approach. In the first phase, the source guest is stopped, the CPU's and devices' state is migrated to the destination and the guest is started on the destination host. When a memory access occurs, a page fault is generated on the destination, and then, the destination requires the page that caused the page fault from the source. To optimize the process, other pages will be delivered with the required page as well.

While the Post-Copy Live Migration has the advantage that the memory is transmitted a single time through the network, a fall-back mechanism is hard to be implemented, as opposed to the Pre-Copy Live Migration where if the migration process fails, the guest will continue running on the source host.

III. Related Work

A. Suspend and Resume a bhyve guest's state

As presented in Section II-C, during the migration process, a state save and restore procedure is needed: the guest's state is saved on the source host and restored on the destination host.

A project for bhyve state save and restore is also developed at University POLITEHNICA of Bucharest [2]. The project [2] introduces a suspend/resume feature for bhyve. When the suspend command is received, the bhyve process stops the virtual machine, saves the guest's state and its memory to disk files and destroys the guest. When resuming a virtual machine, the bhyve process restores the guest state based on the saved information.

```
# Suspend bhyve guest
root@host# bhyvectl --suspend=file.ckp \
                    --vm=vmname

# Restore a guest from checkpoint
root@host# bhyve <bhyve_options> \
                 -r file.ckp vmname
```

Listing 1. Suspend and Resume a bhyve guest

The Suspend request is sent to a bhyve guest by using the `bhyvectl` tool with the `--suspend` option and a file name for saving the data, as seen in Listing 1. Considering the code snippet in Listing 1, during the suspend process, three new files are created:

- **filename.ckp** - contains guest's memory.
- **filename.ckp.kern** - contains guest's devices and CPU state.
- **filename.ckp.meta** - contains metadata related to the saved devices and their offset in the **filename.ckp.kern** file.

Aside from the guest memory, there are other three main components whose state is saved during the suspend process:

- CPU state and related structures,
- Kernel devices such as VHPET (Virtual High Precision Timer), VRTC (Virtual Real Time Clock), VLAPIC (Virtual Local APIC), TSC (Time Stamp Counter).
- Userspace emulated devices (e.g., virtio-net, virtio-block, uart, ahci, lpc, frame buffer, xhci).

The Resume request, as seen in Listing 1, uses the `bhyve` tool with the `-r` parameter followed by the file name used when suspending the guest state. The restore process creates

a fresh virtual machine based on the given disk image and updates its state and memory before the virtual CPUs are started.

B. Warm Migration in bhyve

Based on the save and restore feature for bhyve presented in Section III-A, a warm migration feature was added to bhyve [12]. Using the same API to retrieve a guest's state and memory as the save and restore project, the migration feature opens a connection between the source host and the destination host and sends the guest's state and memory through a socket.

As presented in Section III-A, the suspend/resume feature does not provide a disk checkpoint mechanism, and therefore, in order to warm migrate a bhyve guest, the same disk image must be shared between the two hosts using a storage sharing mechanism such as NFS (Network File System).

```
# Start source guest
root@src# bhyve <bhyve_options> vmsrc

# Start destination guest
# and wait for migration
root@dst# bhyve <bhyve_options> \
                -R src_ip,port vmdst

# Migrate guest
root@src# bhyvectl --migrate=dst_ip,port \
                   --vm=vmsrc
```

Listing 2. Warm migrating a bhyve guest

In Listing 2 is presented an example of warm migration usage. In order to warm migrate a virtual machine, a fresh guest is started on the destination host using the `bhyve` tool with the `-R` parameter followed by the host's IP and the listening port. The destination host waits to receive source guest's state. To send the guest's state, on the source host, the `bhvyectl` tool is used with the `--migrate` parameter followed by the destination host and a port. After the communication between the two hosts is established, the source host is stopped, its state and memory is sent to the destination host and if the migration is successfully completed, the source guest is destroyed (otherwise, if an error occurs, the guest will continue running on the source host). The destination host receives the guest's state and memory and based on the resume state API [2], restores the guest and starts the virtual machine's CPUs.

IV. Live Memory Migration using a Copy-on-Write approach

The memory migration is the core of a live migration feature, and in the same time, is the most difficult part to implement. As stated in Section II-C, the memory can be migrated before [8] or after [9] starting the guest on the destination host.

Considering the pre-copy live migration feature [8], the same memory page can be migrated more than once, whereas in the post-copy live migration approach [9], each page is migrated only once. However, when the migration procedure fails (e.g., network connection become unavailable), the post-copy live migration approach needs to implement a fall-back mechanism [9]. In the pre-copy live migration, when an error occurs, the fall-back mechanism is ensured by default because the guest continues running on the source host. Considering this, we choose to implement a pre-copy live migration feature for bhyve.

As presented in Section II-C, in a pre-copy live migration approach, the memory is migrated in rounds and, each round, the algorithm has to determine the pages that were dirtied since the last round started. In other words, the procedure needs to determine the changes that occured in the same memory area between two moments in time.

The Copy-on-Write (CoW) mechanism can be used to determine the memory differences that were made in a time interval. As presented in Listing 3, in FreeBSD, to determine the memory differences, the memory object can be marked as Copy-On-Write. Then, the object and its shadow object can be compared to determine what pages were modified. As presented in Section II-A, when a write operation occurs in a page that was not been dirtied since the initial object was marked as CoW, a copy of that page is added into the shadow object. Considering this, the shadow object contains only the pages that were dirtied since the initial object was marked as CoW.

```
1 VM_OBJ = get_current_mem_obj();
2 SHADOW_OBJ = set_obj_cow(VM_OBJ);
3 wait_for_round_to_finish();
4 pg = get_pg(SHADOW_OBJ);
```

Listing 3. Determining memory differences using the Copy-on-Write

The Copy-on-Write approach presented in Listing 3 can be used to determine the memory that can be migrated in each round. The algorithm needs guest memory objects to be marked as CoW before starting a new memory migration round. In the next round, the pages that should be migrated are represented by the shadow object pages.

However, due to the dual memory view presented in Section II-B, the Copy-on-Write approach previously presented cannot be used to migrate de virtual machine's memory. In order to mark a virtual memory object (the `struct vm_object` entity) as Copy-on-Write, some flags (`MAP_ENTRY_COW` and `MAP_ENTRY_NEEDS_COPY`) have to be set in the `struct vm_map_entry` entity. However, as seen in Fig. 1, the guest and the host have different `struct vm_map_entry` entities to refer the same object.

If one of the `struct vm_map_entry` entities (from guest side or from host side) is marked as Copy-on-Write, the other `struct vm_map_entry` entity points to other `struct vm_object` so the two sides will no longer have the same view of the memory. If one of them writes in the guest's memory (such as virtio devices) the page will be copied into the shadow object and the other side will not see the

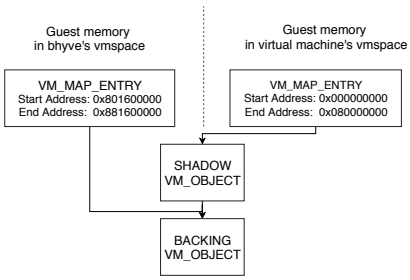


Fig. 2. Dual Guest Memory View - lowmem segment - bhyve's VM_MAP_ENTRY is marked as COW

changes. This leads to communication errors between the host and the guest that will eventually crash the virtual machine.

In Fig. 2, it is shown the virtual memory layout after the guest's side `struct vm_map_entry` that contains the memory object was marked as copy-on-write.

This approach is the one of the easiest methods of determining the memory that should be migrated each round because of the FreeBSD virtual memory subsystem implementation. Using the CoW mechanism, the only pages that should be migrated are the ones that reside in the shadow object. Even if this method cannot be used in bhyve due to its dual memory view, it gave us a clearer picture of the virtual memory subsystem functionality and we developed another memory modification detection mechanism to implement the live migration feature for bhyve.

V. Live Memory Migration using a dirty-bit approach

As stated in Section IV, the memory migration is one of the most challenging parts in a live migration feature. In Section IV we presented a memory modification detection mechanism that uses with virtual memory objects.

Even though the virtual memory objects that represent the guest memory are the highest entities in the virtual machine memory hierarchy (as presented in Fig. 1) that are common for both the guest (i.e., as seen by the virtual machine's `vmspace` entity) and the host (i.e., as seen by the bhyve user space utility tool's `vmspace` entity), there are other constraints that do not allow the use of virtual memory objects to determine memory differences. For instance, one of the constraints is the fact that a `vm_object` entity is marked as CoW by setting some flags in a memory layout enitity that is not shared between host and guest because of their separate memory views. Thus, we will present a solution that uses another entity shared between host and guest: the `struct vm_page` entities.

Each `struct vm_page` contains a dirty flag field that indicates if the page has been modified and if so, the changes should be written on the disk. This flag is updated from time to time based on the modified bit of the physical page by inspecting the A/D - access/dirty - bits. Even though this flag could be used for determining the pages that should be migrated, we cannot interfere with this flag since the virtual memory system relies on it to perform some virtual memory subsystem actions such as the memory laundering process [11], and modifying its behavior would imply some undesired operating system behavior.

Instead of using the dirty flag, the proposed approach uses a custom dirty bit used only by the bhyve hypervisor. The custom dirty bit, named **virtual-machine-dirty bit**, is set each time the dirty flag is set, but unset only by the bhyve hypervisor.

```
1 VM_OBJ = get_current_mem_obj();
2 clear_vmm_dirty_bit_for_pg_in(VM_OBJ);
3 wait_for_round_to_finish();
4 pg = get_pg_with_vmm_dirty_bit(VM_OBJ);
```

Listing 4. Finding a guest's dirty pages using dirty bit

The pseudo-code snippet in Listing 4 presents a mechanism that can be used to determine memory pages changed between two moments in time. Firstly, each page from the memory object should have the virtual-machine-dirty bit clean (to eliminate false-positive cases). The memory differences can be determined by inspecting the virtual memory object pages that have the virtual-machine-dirty bit set.

VI. Algorithm

In order to implement a live migration feature for bhyve, we use the approach presented in Section V for migrating the memory and the state save and restore mechanism implemented for bhyve [2] that was presented in the Section III-A. The connection mechanism between the source host and the destination host is the socket solution also used by the warm migration feature for bhyve [12] shown in Section III-B. The live migration algorithm is similar to the warm migration algorithm, the major differences are related only to the memory migration.

```
1 connect(src, dst);
2 check_compatibility(src, dst);
3 live_migration_send_memory_to(dst);
4 snapshot_and_send_state();
5 destroy_vm()
```

Listing 5. Live Migration Algorithm - Source host method

```
1 connect(src, dst);
2 check_compatibility(src, dst);
3 live_migration_recv_memory_from(src);
4 recv_and_resume_state();
5 spinof_vcpus();
```

Listing 6. Live Migration Algorithm - Destination host method

The pseudo-code snippets from Listing 5 and from Listing 6 present the functions that run on the source host and on the destination host in order to migrate a guest. After the connection between the source and the destination hosts is done, there is a check to determine whether the two are compatible for migration (e.g., same CPU vendor and model, same guest memory size, same virtual memory page size). After that, the memory is migrated in rounds. In the last step,

the guest's remaining dirty memory is sent to the destination, and the guest's CPU's and devices's state is being snapshot using the state save and restore feature. The virtual machine's state is restored at the destination. The guest is stopped on the source host before the last step.

```
1  live_migration_send:
2    for i=1:N
3      if i == 0
4        // First Round
5        mark_all_memory_dirty();
6      endif
7      if i == N
8        // Last Round
9        stop(vm);
10     endif
11     pages = get_dirty_pages();
12     send(pages);
13   end for
14
15 send(pages):
16   for each page : pages
17         get_from_memory(page);
18         clear_dirty_bit(page);
19         send_to_dest(page);
20   end for
```

Listing 7. Live Memory Migration Algorithm - Send Memory

```
1  live_migration_recv:
2    while recv_from_src(page)
3      update(page);
4    end while
```

Listing 8. Live Memory Migration Algorithm - Receive Memory

In Listing 7 the algorithm used for sending the guest memory in rounds to the destination is presented. In the first migration round, all guest pages should be migrated so each page is artificially set as dirty by setting the virtual-machine-dirty bit. In the next rounds, the memory differences are determined by iterating through all of the guest pages and sending them one by one to the destination. When copying a page from the guest memory, we clear the virtual-machine-dirty bit. By clearing the virtual-machine-dirty bit for each migrated page, we prepare the guest for the next round. In the last round, the virtual machine should be stopped and the remaining memory migrated. Listing 8 shows the algorithm used by the destination host. It receives pages one by one and updates the guest memory that will be started after the migration process completes.

VII. Current Status in bhyve and Future Work

The algorithm presented in Section VI is implemented in bhyve [13] and the project is still under development. To start a migration procedure, the process is similar to the warm migration algorithm. As seen in Listing 9, in terms of usage, the only difference between warm and live migration is related to the `bhyvectl` command that starts migrating the guest: instead of `migrate=dst_ip,port` the `migrate-live` option is used.

```
# Start source guest
root@src# bhyve <bhyve_options> vmsrc

# Start destination guest
# and wait for migration
root@dst# bhyve <bhyve_options> \
                -R src_ip,port vmdst

# Migrate guest
root@src# bhyvectl \
             --migrate-live=dst_ip,port \
             --vm=vmsrc
```

Listing 9. Live migrating a bhyve guest

In order to have the same connection framework for both warm and live migration, we modified the algorithm so among the initial messages related to specification checks, the type of migration (warm or live) is sent to the destination. Based the migration type, the destination determines the functions to be called for memory migration.

The framework for live migration is implemented, the live migration feature is not yet stable and there are currently some limitations that should be considered next:

- the guest memory should be wired - since we added a mechanism for retrieving pages from memory and the first migration round is supposed to migrate all the pages, all memory should be allocated, and the pages should not be swapped out. Thus, we choose to live migrate only wired guests.
- the guest memory size should be less than the lowmem segment - we implemented the framework to work for guests that have assigned a virtual memory object only for lowmem segment.
- the migrated guest crashes in some of the test scenarios. The debugging process is still ongoing and this behavior may be caused by the emulated devices that run in user-space threads, that will continue running even when the guest's virtual CPUs are locked, affecting or even corrupting the guest's state and disk.

VIII. Conclusion

In this paper, we presented two mechanism for determining the memory differences between two memory rounds.

The first approach is based on the FreeBSD Copy-on-Write mechanism. To identify the pages that should be migrated using shadow virtual memory objects. Even if the algorithm can determine the modified pages between two memory migration rounds, it cannot be implemented in bhyve due to the dual memory view of the guest memory.

The second mechanism for detecting the modified pages is based on a dirty bit approach. We use a bit, named virtual-machine-dirty bit, that is managed only by the hypervisor.

The dirty-bit approach is currently used for live migrating the guest memory. Even if the framework is implemented, the live migration feature is not yet stable and there are some challenges and improvements that should be considered in the future.

ACKNOWLEDGMENTS

The authors would like to thank to John Baldwin, Mark Johnston and Alan Cox for their help and advice in regards to bhyve and FreeBSD's virtual memory subsystem implementation and functionality. Also, the authors would like to thank to Matthew Grooms for his technical and financial support in form of scholarship for Elena Mihailescu.

REFERENCES

[1] FreeBSD Handbook, Chapter 21. Virtualization, Section 21.7 FreeBSD as Host with bhyve [Online]. Available: https://www.freebsd.org/doc/handbook/virtualization-host-bhyve.html, [Accessed Dec, 21st, 2018].

[2] University POLITEHNICA of Bucharest, Save & Restore Project for bhyve (amd64) [Online]. Available: https://github.com/FreeBSD-UPB/freebsd/tree/projects/bhyve_snapshot, [Accessed Jan, 27th, 2019].

[3] Matthew Dillon, "Design elements of the FreeBSD VM system" [Online]. Available: https://www.freebsd.org/doc/en/articles/vm-design/, [Accessed Dec, 21st, 2018].

[4] N. Natu, P. Grehan, "Nested Paging in bhyve", in *AsiaBSDCon, Tokio, Japan, March 2014*

[5] The FreeBSD Project, FreeBSD Source Code [Online]. Available: https://github.com/freebsd/freebsd, [Accessed Dec, 21st, 2018].

[6] VMware, "VMware virtual machine migration types vSphere 6.0" [Online]. Available: https://communities.vmware.com/docs/DOC-31922 [Accessed Dec, 21st, 2018].

[7] Network Startup Resource Center, "Virtual Machine Migration" in *Cloud / Virtualization workshop*, Thimphu, Bhutan, 17-21 January 2014 [Online]. Available: https://nsrc.org/workshops/2014/sanog23-virtualization/raw-attachment/wiki/Agenda/migration-storage.pdf [Accessed Dec, 21st, 2018].

[8] C. Clark, and K. Fraser and S. Hand, and J.G. Hansen, and E. Jul, and C. Limpach, and I. Pratt, and A. Warfield, "Live migration of virtual machines" in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, 2005, pp. 273-286

[9] M.R. Hines and U. Deshpande and K. Gopalan, "Post-copy live migration of virtual machines", *ACM SIGOPS operating systems review*, vol. 43, pp. 14-26, 2009.

[10] A. Kivity and Y. Kamay and D. Laor and U. Lublin and A. Liguori, "kvm: the Linux virtual machine monitor", in *Proceedings of the Linux symposium*, vol. 1, pp. 225-230, 2007

[11] The FreeBSD Documentation Project, "FreeBSD Architecture Handbook", Chapter 7. Virtual Memory System [Online]. Available: https://www.freebsd.org/doc/en/books/arch-handbook/vm.html [Accessed Jan, 27th, 2019]

[12] University POLITEHNICA of Bucharest, Warm Migration for bhyve (amd64) [Online]. Available: https://github.com/FreeBSD-UPB/freebsd/tree/projects/bhyve_warm_migration, [Accessed Jan, 27th, 2019].

[13] University POLITEHNICA of Bucharest, Live Migration for bhyve (amd64) [Online]. Available: https://github.com/FreeBSD-UPB/freebsd/tree/projects/bhyve_migration_dev, [Accessed Jan, 27th, 2019]

Yet Another Container Migration on FreeBSD

Yuhei Takagawa
Future University Hakodate

Katsuya Matsubara
Future University Hakodate

Abstract

The container-based virtualization, that multiplexes and isolates computing resource and name space which operating system (OS) provides for each process group of application, has been recently attracted. We focus on container migration among machines since it is one of the most important technology for realizing load balancing and increasing availability in cloud computing, that is a major application of the virtualization.

Although FreeBSD VPS has already implemented one kind of migratable containers in FreeBSD, it is not enough in terms of resource limitation, compared to Linux one. This paper shows a novel implementation that how resource limitation and isolation close to that of Linux can be realized for FreeBSD containers. We also explain how processes, which could have sessions of file open and network connection, running in a FreeBSD container can be checkpointed and then they can be restored in another container. This implementation bases on *runC* which is one of standard container runtime and *CRIU* which is a major process migration tool in Linux.

1 Introduction

The container-based virtualization, that multiplexes and isolates computing resource and name space which operating system (OS) provides for each process group of application, has been recently attracted, such as Docker [2], LXC [3], and FreeBSD Jail [1].

We focus on container migration since it is one of the most important technology for realizing load balancing and increasing availability in cloud computing, that is a major application of the virtualization. Especially We claim that implementing containers and its migration along the standard interfaces and protocols could promote interoperability with recent containerizing services such as Docker and Kubernetes. There have already presented some of study and work related to container on FreeBSD. The Docker-FreeBSD port [4] exists as a native port of the Docker v1 engine with Jail and zfs, however, unfortunately it may have become a fossile since it has not been updated since 2015. FreeBSD VPS(Virtual Private System) [6] realizes migratable containers isolated by Jail. Unfortunately, the containers lacks of limiting resources such as CPU and memory usage, and the system has no compatible interfaces to cooperate with the major containerizing services.

In this paper, we propose a novel implementation of containers which limit and isolate resources, close to that of Linux, and container migration functionality with the emphasis on cooperating the defact-standard containerizing services such as the latest Docker and Kubernetes. We have ported two of the standard software components for container system implementation in Linux, runC and CRIU [5,7] (Checkpointing and Restoring in User-space), to FreeBSD: The revised runC not only can isolate but also can limit resouces. The ported CRIU supports processes natively running on FreeBSD.

The following sections shows resouce limit on FreeBSD realize in runC, then explains how running processes in a container, which could have sessions of file open and network connection, can be captured with the CRIU-defined representation and then they restore on another OS with sustaining their sessions. Finally, we also show result of a preliminary evaluation of the implementation.

2 Limiting Resource Usage in a FreeBSD Container

Fortunately, an implementation of runC for FreeBSD has existed, which was developed by Hongjiang Zhang [8]. Especially it is ideal for our purpose that it has compatible with Linux one on the configuration by config.json file. However, it has been only implemented the resource isolation, not the reource limitation. So we has realized additional functionality of the resource limitation in the FreeBSD runC implemetation.

Corresponding with Linux namespace, FreeBSD jail can be used to realize to isolate resources for each container. Table 1 shows counterparts for functionalities of the resource isolation in Linux and FreeBSD. Althrough almost all resources besides PID and User can be isolated also in FreeBSD, jail does not permit to assign PID 1 to multiple processes simultaneously in a system. And it is impossible to isolate set of users and groups identification for each container in FreeBSD. Note that jail requires to execute the jail command at first, and then to spawn the target process as

Table 1: Isolated resources by Linux namespace and FreeBSD jail

Isolated resource	Linux	FreeBSD
IPC	namespace	jail
Mount		
UTS		
Network		
cgroups		
PID		jail (limited support)
User		

Table 2: Counterparts of Linux's resource controls in FreeBSD

Linux cgroups	Counterpart in FreeBSD
memory	RCTL memoryuse
cpushare	(N/A)
cpuquota	RCTL pcpu (convert)
hugepage	superpages (limited support)
devices	devfs
cpuset	cpuset
cpuperiod	RCTL cputime

a child of the jail command process because it disallows to control resources for other processes. The runC implementation in FreeBSD follows the flow.

The specification of the resource usage control in runC follows the OCI (Open Containers Initiative) standard, which has been based on Linux cgroups' functionalities. FreeBSD provides similar functionalites for the resource usage control via several frameworks such as jail, RACCT/RCTL and so on. Table.2 shows some of resources, which are especially supported in Kubernetes, and control framework for each resource in Linux and FreeBSD. We have replacing almost all implementation of the resource usage control based on Linux cgroup in runC with the FreeBSD counterparts. Unfortunately, some of limitations remain in our runC implementation; cpushare and hugepage unsupported.

By contrast, FreeBSD RCTL allows to specify an action when amount of resource usage reaches the limit although Linux cgroups always exercises 'deny'. Our FreeBSD runC aligns the behavior to Linux one.

To pass a value of cpuperiod in Linux cgroups into cputime in FreeBSD RCTL, it must require to convert the unit with Eq.(1).

$$cputime = \frac{cpuperiod}{1000000} \tag{1}$$

Also Eq.(2) shows a conversion from cpuquota of Linux cgroups to pcpu of FreeBSD RCTL for specifying rate of CPU usage.

$$pcpu = \frac{cpuquota}{cpuperiod} \times 100 \tag{2}$$

Linux cgroups uses major and minor numbers to identify devices, however, name must be used to specify devices in FreeBSD. Unfortunately, a device could be assigned with different numbers in Linux and FreeBSD althrough it has the same name. We added a conversion of identification between device numbers and names only for some of special device; null and zero devices, tty, urandom, random, console, pts, and so on.

3 Checkpoinging and Restoring a FreeBSD container in User-space

CRIU is enabled to migrate processes by getting and restoring the process state. Our target process is Linux ELF on FreeBSD running by Linux emulation (Linuxulator). The process state is cpu register and memory basically. In addition, there are opened file state, network state and etc.. In this paper, we subscribe basic process migration, opened files migration and network state.

3.1 Migrating a Process

Normally, the process migration requires register information and memory. The memory data is the same presentation if a process is running on the same architecture. Also, most recent OSes allocate memory to enhance security with address space layout randomization (ASLR). On an OS with ASLR enabled, memory is allocated to process randomly so memory layout is different always. Although FreeBSD does not yet implement ASLR, we will make it possible to change the memory layout corresponding to ASLR. We reallocate memory layout to convert memory layout.

The register information can be gotten and set by the ptrace system call. Note that the set of segment registers use as it is. Also, the memory can be gotten by the read system call from mem file of procfs, set by the write system call from memory file of procfs. Before memory data is set, memory layout is reallocated with the mmap system call and the unmmap system call.

For process state restoration, the process is set traceme option of ptrace system call, run with execvp system call. The important thing for Linuxulator is that you need to set register information and memory after the first instruction of the main function is executed.

3.2 Migrating State of Opened Files

We get and restore the opened file state. The opened file state include file path, file descriptor number, file offset, file mode, and file flags. On Linux, we can get the opened file state from fdinfo directory of procfs. However, FreeBSD doesn't have fdinfo directory. We can get only the process's file information via fd directory of devfs. Therefore, we adopt libprocstat library to get another process's opened file state by __sysctl system call.

Table 3: The difference of open system call

	Linux	FreeBSD
system call number	2	5
option O_CREAT	0x0200	0x0040

To restore the following procedure.

1. To open the file with the open system call.
2. To set flags and access mode to options of the open system call.
3. To change file descriptor number with the dup2 system call.
4. To update file offset with the lseek system call.
5. To start process with the execvp system call.

3.3 Migrating TCP Connections

We present checkpointing and restoring TCP connection state, called TCP repair. The TCP connection state is consist of send queue (*SNDQ*) data, receive queue (*RCVQ*) data and sequence number. TCP repair is sent patch Linux Kernel 3.6 by CRIU project and we implement TCP repair for FreeBSD Kernel based on Linux TCP repair. TCP repair requires following things.

- Getting *SNDQ* data and *RCVQ* data.
- Setting data to *SNDQ* and *RCVQ*.
- Getting/Setting sequence number.
- Closing socket without not sending FIN packet.
- Connecting socket without not sending SYN packet.

Figure 1 shows packet interactions of the three-way handshake initiated by the connect and the close system calls. In order not to close network connection and to redo three-way handshake, we should extend connect and close don't send SYN/FIN packet. To make correct communication after migration, sequence numbers are set to destination machine.

Figure 2 shows data on TCP connections. Meaning of number is as follows:

1. This hasn't been copied to a queue in kernel.
2. This hasn't sent to the network yet.
3. This has sent network but it hasn't been received yet.
4. This hasn't been read by a process yet.
5. This has been received by a process.

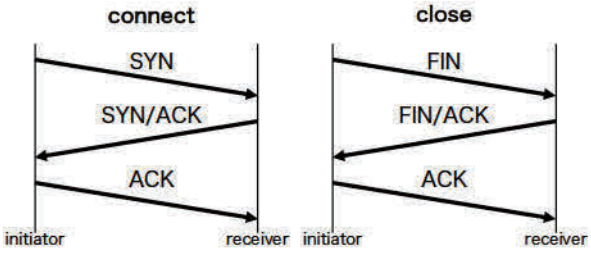


Figure 1: The three-way handshake for TCP connection

Figure 2: Data kept on TCP connection

The data of (1), (5) are gotten and restored with process's memory. The data of (3) is retransmitted by TCP if it lost in the network. Normally, the data of (2), (4) cannot be gotten and restored so we implement getting and restoring these.

In Linux, the getsockopt sytem call and the setsockopt system call are added options, TCP_REPAIR, TCP_REPAIR_QUEUE and TCP_QUEUE_SEQ option. Using setsockopt with TCP_REPAIR option, connect/close don't send SYN/FIN packet. Using setsockopt with TCP_REPAIR_QUEUE option, a target queue is chosen. Using setsockopt/getsockopt TCP_REPAIR_SEQ option, get/set sequence number. The extended the recvmsg system call get data from queue which chosen by setsockopt with TCP_REPAIR_QUEUE option. The extended the sendmsg system call set data to queue which chosen by setsockopt with TCP_REPAIR_QUEUE option. We implement TCP repair for FreeBSD Kernel based on Linux TCP repair but the original implementation of these functions differs depending on the kernel.

In FreeBSD Kernel, *SNDQ* data and *RCVQ* data consist of mbuf. When sendmsg called, a buffer is copied only the specified size to mbuf for *SNDQ* from userspace through kernel buffer. When recvmsg called, a buffer is copied only the specified size to userspace from mbuf for *RCVQ* through kernel buffer. Normally, sendmsg cannot copy buffer to mbuf for *RCVQ*, recvmsg cannot copy the buffer from mbuf for *SNDQ*. If the kernel can copy buffer to *RCVQ* with sendmsg and copy the buffer from *SNDQ* with recvmsg, we can repair TCP queue with the same interface as before.

FreeBSD Kernel holds sequence number with struct *tcpcb*. The struct *tcpcb* has member snd_nxt and rcv_nxt. The snd_nxt is number next sending data. The rcv_nxt is number next coming data. We extend setsockopt/getsockopt to switch repair mode (TCP _REPAIR), choose a queue

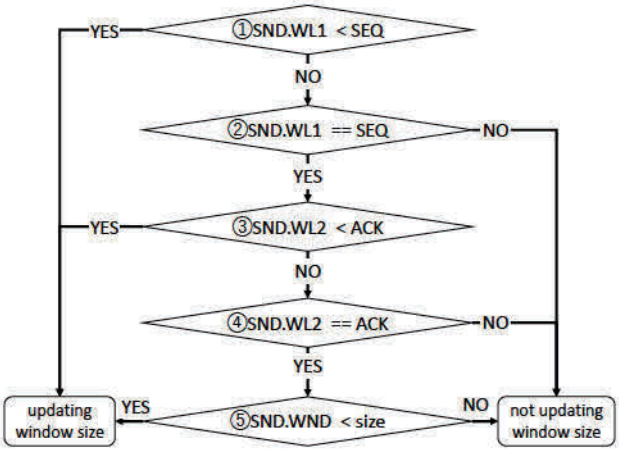


Figure 3: flow chart of updating window size

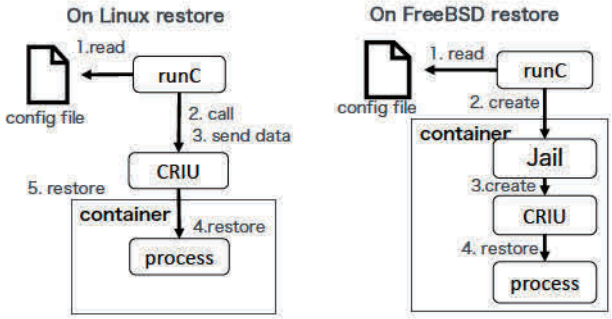


Figure 4: abstract of container migration

(TCP_REPAIR_QUEUE) and get/set sequence number referencing struct *tcpcb* (TCP_QUEUE_SEQ).

Also, FreeBSD Kernel's protocol stack is different from Linux Kernel's so getting and setting data are difference. In particular, it can cause the window size not to be updated by flow control. In FreeBSD, updating window size require SND.WL2 but Linux doesn't get it. Fig.3 shows the flow chart when update window size in FreeBSD. In evaluation formula (3) and (4), window size don't be updated because don't available SND.WL2 and there are false. Therefore, it becomes the cause not to update the case that it becomes conditional expression (2) As for the approach of this paper, we set to satisfy conditional expression (1) so that only the first evaluation after restoration is always true. In other words, restore SND.WL1 to keep it smaller than the sequence number.SND.WL1 and SND.WL2 have no problem because correct values are set immediately after evaluation. In order to solve this problem, evaluate the conditional expression for updating the first window size after restoration to be true.

3.4 Migrating a Container

In Linux, at restore container runC read required restoring data from config.json and dump files then it sends the data to CRIU by Unix Domain Socket with Google Protocol Buffer, in the last, CRIU restore the process and container. In this paper, at migrating a container, runC reuses config.json and function which create container. This method is not same runC on Linux because it seperates the role of runC and CRIU. At dump a container, runC call CRIU as the additional process with jexec command. At restore a container, runC call CRIU as the initial process with jail command. Therefore, the executable file of CRIU copy to jail root directory.

Fig.4 shows the restoring container in Linux and FreeBSD. On Linux, when a container is restored runC reads the above setting from config.json which is the configuration file, passes data to CRIU via Unix Domain Socket in the form of Google Protocol Buffer, CRIU restores the process and restores container. Container migration in this paper reuses the container creation function of the runC and the config file config.json After the runC reads the config.json, it creates a container with the Jail command and the CRIU processes in the container The CRIU operates only on the process and runC operates only on the container. In regard to the information on the container, it is not necessary to acquire the configuration file as it is, but it is not necessary to acquire it, Since the limit value changed without the intervention of run C is not reflected in config.json, it is necessary to consider the acquisition method.

Table 4: Experiment PC spec

OS	FreeBSD 11.2-RELEASE
CPU	Intel(R)Core(TM)i5-7200 3.40GHz
RAM	8GB
HDD	1TB

4 Preliminary Evaluation

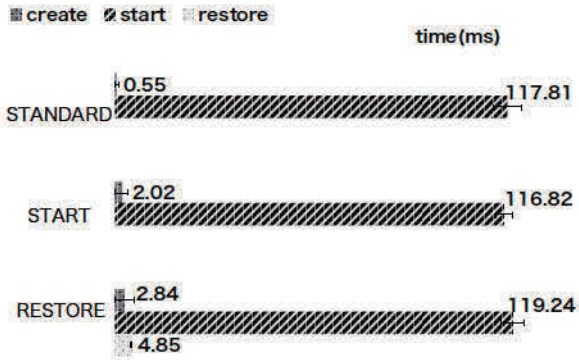


Figure 5: Time taken to start and restore container with runC

Table.4 show experiment environment. To measure the

time it takes to start and restore container with runC 50 times. The target process of this test is opened file but don't have TCP connection. The result shows in Fig.5. The horizontal line shows time taken. The time of "STANDARD" is taken start container with runC which no resource limitation function and migration function. The time of "START" and "RESTORE" is taken start and restore with runC which have resource limitation function and migration function. The "create", "start", "restore" shows took to create container, start target process, restore target process. In the case the "RESTORE", the "start" presents the time to start CRIU.

In the starting, creating container time increase 1.46 ms than the normal starting, this is 1.23% of total time of the normal starting. In the restoring, creating container time increase 2.29 ms than the normal starting, this is 1.93% of total time of the normal starting. In addition, the standard deviation of the time until the target process starts execution is 2.62 ms in the case of the resource limiting function, 3.43 ms in the case of the resource limiting function and the container restoring function, that is, The overhead is smaller than that of the shake, and it can be said that it can be extended with small overhead.

5 Conclusion

We proposed how resources consumed by processes running in a container can be limited with the runC runtime. The revised runC uses the RCTL command to set a limit on memory and cpu usage for each container. Some parameters of cpu limit specified in runC config should be converted since they have been defined among the specification of Linux cgroups.

We also showed how state of a process running on FreeBSD can be checkpointed and restored from user-space with the CRIU tool. The ptrace syscall in FreeBSD can be used enough to read and write values of cpu registers for each process. Content of process memory can be dumped and restored through the procfs mem entry. In FreeBSD, state of files opened by the target process can be captured from the devfs fd entry. Then restoring the file state can be realized by injecting code, which opens the files and invokes the lseek syscall to restore file offset of the each file, into the destination process. In order to dump and restore TCP connections held by a migrating process, the current FreeBSD kernel must be modified to allow to access the mbuf buffer and the tcpcb structure data from user-space, moreover, adjust the window size at the restore.

The remaining issue is to support dynamically changing setting of resouce limit and isolation. In Linux, CRIU can capture and restore the current cgroup setting for the target container. In contrast, in our current implementation, runC creates a new destination container with the config file of the source container, instead of capturing state of resource limit and isolation.

Furthermore we would like to realize container migration between FreeBSD and Linux with cooperating the Kubernetes container orchestration.

References

[1] Poul henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*, 2000.

[2] Docker Inc. The Docker Containerization Platform. https://www.docker.com/ . Accessed: 2019-01-24.

[3] LinuxContainers.org. LinuxContainers.org. https://linuxcontainers.org . Accessed: 2019-01-30.

[4] MateuszPiotrowski. Docker on FreeBSD. https://wiki.freebsd.org/Docker . Accessed: 2019-01-30.

[5] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium 2008*, volume 2 of *OLS '08*, pages 85–90, 2008.

[6] Klaus P. Ohrhallinger. Virtual Private System for FreeBSD. 2010.

[7] CRIU Project. Criu main page. https://criu.org/ . Accessed: 2019-01-30.

[8] Hongjiang Zhang. Implement FreeBSD runc with the help of Jail. https://lists.freebsd.org/pipermail/freebsd-jail/2017-July/003400.html . Accessed: 2019-01-30.

Finalizing booting requirements for a guest running under bhyvearm

Nicolae-Alexandru Ivan, Mihai Carabas
Automatic Control and Computers Faculty
University POLITEHNICA of Bucharest
Emails: nicolae.ivan@stud.acs.upb.ro.ro, mihai.carabas@cs.pub.ro

***Abstract*—**

Keeping track of time is an invaluable resource in modern software systems. The vast majority of existing CPUs posses various clocks and timers in order to accommodate time related mechanisms required by software. These same needs apply to virtualized environments, where the guest operating system uses time based events. To this end, a virtualized timer is required. This research project describes implementing such a timer in FreeBSD for the ARMv7 architecture.

***Index Terms*—FreeBSD, bhyve, hypervisor, ARMv7, GIC, vGIC, interrupts, Cubieboard2, Allwinner A20**

I. Introduction

In the current stage, a guest running under the ARM FreeBSD hypervisor (bhyve-arm) isn't able to boot due to lack of a virtual timer implementation and issues with VFP (vector floating point) and WFI instruction. This paper will tackle mainly the timer virtualization and also the remaining issues to boot a guest.

Timed events are a core element of many software systems. Their utility ranges from pre-empting processes while in kernel space to scheduling events in high level programming in user space. It is clear that these types of functionality are also desirable when running software in a virtualized environment.

The need for keeping time has brought about the introduction of new timer hardware, such as the Programmable Interval Timer(PIT), the Real Time Clock(RTC), the Advanced Configuration and Power Interface(ACPI) and the High Precision Event Timer(HPET), each with their own utility.

The above mentioned as well as most other hardware timers have the same basic functionality, as described by Figure 1. An oscillator produces a precise frequency signal. Each cycle of the oscillator updates the counter. When reaching a specific value, it generates an output signal. Usually, this signal is a interrupt that lets the CPU know that some amount of time has passed. Depending on the specific type of timer, there may be additional components.[9]

In the next sections, the following topics are discussed: section two describes the state of the art - how other systems virtualize timers, section three goes into detail concerning the implementation, and the final section concludes with results and plans for further development.

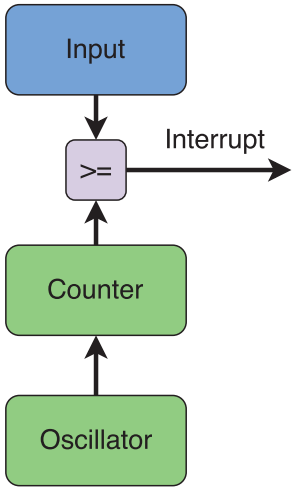


Figure 1. Timer functionality

II. State of the Art

Timer virtualization depends heavily on the underlying architecture. Some hardware platforms have virtualization extensions meant to facilitate guest interaction with hardware features as the timer. Other platforms lack such support and, in this case, the hardware device must be emulated entirely.

A. Platforms without Timer Virtualization Extension

Platforms such as the widely used x86 have no support for timer virtualization. This means that the virtualization infrastructure must emulate all access to the timer, as well as the all interrupts produced by it. Various techniques can be used to achieve this, as follows.

1) VMware: VMware uses proprietary technology, allowing the guest operating system to fall behind and catch up as necessary, without losing functionality. The time which is visible to guest systems is called apparent time. An in-depth description of the functioning of each of the virtual timers present in VMware can be found in the following VMware paper.[9]

2) Xen: Xen approaches this issue differently. It uses paravirtualization - the guest is aware that it is functioning inside

a virtualized environment. The guest kernel is modified to contain a Clock Event Device which schedules events trough the use of hypercalls. The resulting interrupt will be caught in the hypervisor and delivered to the appropriate virtual machine.[3]

3) Linux KVM: KVM supports both fully virtualized and paravirtualized virtual timers. The fully emulated virtual timer uses high resolution timers in the Linux kernel to keep track of guest timer events. When a timer is fired inside KVM, it is flagged that timers have to be taken into consideration upon entering the guest.

As an alternative, KVM also provides a paravirtualized kvm-clock which may be used as a clocksource by guest operating systems.[7]

4) bhyve: bhyve, the FreeBSD hypervisor, supports fully emulated timers for platforms that have no hardware support for virtual timers. Similar to KVM, the bhyve implementation uses the existing high performance event timers from kernel space to handle guest timer events.

B. ARMv7

ARMv7 offers hardware support for virtual timers, thus rendering both the performance penalty on the guest and the amount of work required inside the hypervisor minimal. The guest is allowed to interact directly with the hardware, with no intervention from the hypervisor. Still, the virtualization infrastructure must perform certain operations to ensure correct functionality of the guest.

KVM has a working implementation of virtual timers for ARM. This implementation was used as a reference when implementing the FreeBSD virtual timer.[6]

III. Implementation

Before discussing the actual implementation, the architecture of ARMv7 timer is presented and, also, a very high level overview of the Generic Interrupt Controller is made. These are necessary in order to understand the implementation. Additionally, a summary of encountered issues is made in the ending of this section.

A. ARMv7 Generic Timer Architecture

The Generic Timer present on the ARMv7 platform is a standardized timer which can be used as a system clock. Aside from the usual counter, which in this case is referred to as physical counter, the ARM Generic Timer may also contains a virtual counter, which can be used by virtual machines for time-keeping purposes. The physical counter is at least 56 bits wide and updates at a constant frequency in the range 1-50MHz. The virtual counter holds the value of the physical counter minus a 64-bit offset.

An implementation of the Generic Timer with Virtualization Extension provides four timers per CPU[5]:

- Non-secure PL1 physical timer
- Secure PL1 physical timer
- Non-secure PL2 physical timer
- Virtual timer

Each of the above provides an interrupt signal. Additionally, each has a set of three registers: a CompareValue register - which is a 64-bit unsigned upcounter, a TimerValue register - which is a 32-bit signed downcounter, and a 32-bit Control register.[5]

B. Virtual Generic Interrupt Controller

ARMv7 platforms use a Generic Interrupt Controller in order to manage interrupts. The GIC keeps track of which interrupts are enabled, prioritizes incoming interrupts and delivers them to the appropriate CPU.[4] Since there is no hardware support for a virtual GIC, it must be emulated by the hypervisor. This means that any access to the GIC from within the guest, as well as any interrupt that should be delivered to the guest must pass through the emulated controller, also called vGIC.

C. Virtual Timer Implementation

Before describing the implementation, the table below describes the registers used.[5]

Name	Description
CNTV_CTL	Virtual Timer Control register. Used by the guest to interact with the timer hardware
CNTV_CVAL	Virtual Timer CompareValue register
CNTHCTL	Controls access to the physical registers. In particular, the PL1PCTEN and PL1PCEN are used to disable access to the physical timer registers
CNTVOFF	Virtual Offset register - specifies value to be subtracted from physical counter in order to obtain virtual counter

Figure 2 constitutes an overview of the workflow.

The workflow of the virtualization process is as follows:

1) At guest initialization (state 0), the CNTVOFF register is initialized with the current value of the physical timer, rendering the virtual counter 0 for the newly created virtual machine
2) Before entering the guest (transition from state 1 to state 2), the hypervisor internal state for the virtual timer is checked in order to determine whether any interrupts

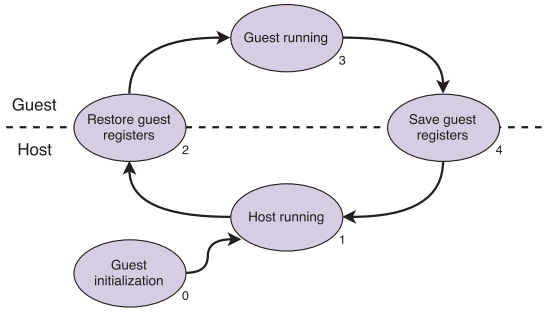


Figure 2. Virtual Timer Workflow

should have been triggered by the timer and need to be injected by the vGIC

3) Upon entering the guest (state 2), the hypervisor enables the virtual timer if necessary, disables access to the physical timer, and restores the CNTVOFF register for the current virtual machine, as well as restoring CNTV_CVAL and CNTV_CTL
4) While running, the guest accesses the virtual timer with no intervention from the hypervisor (state 3); any interrupts triggered here are sent to the vGIC, which will inject them accordingly
5) When exiting the guest, the CNTV_CVAL and CNTV_-CTL registers are saved (state 4) and the hypervisor internal state is updated (transition from state 4 to state 1
6) The host continues to use the physical timer until the guest is run again, when the process resumes at step 2

Notice that only the CompareValue register is saved. This is because both read and write operations on the TimerValue register are translated into reads and writes on the CompareValue.

The implementation of the aforementioned flow is relatively straight forward. The internal structure for a virtual machine is used to memorize whether the virtual timer is enabled in the respective guest and to store the value of CNTVOFF. Similarly, the values for CNTV_CVAL and CNTV_CTL are stored within a per-cpu structure.

One noteworthy aspect is determining whether an interrupt needs to be injected upon re-entering a guest. When syncing the internal bhyve state with the hardware state, it is first checked whether the counter has reached the CompareValue already. If so, the interrupt is injected on the next guest entry. Otherwise, the remaining number of cycles is calculated and a callout[1] event is scheduled. If the callout is executed before running the virtual machine again, the interrupt is injected. In the case where the guest is executed again before the callout, the latter is simply cancelled.

D. Encounterd Issues

One major issue encountered was caused by desynchronising a number of assembly symbols from the C code. The saving and restoring of the timer registers is done directly in assembly code, in which the respective symbols are used to calculate the memory locations of various fields from the hypervisor internal structures. The incorrect offsets used in these calculations corrupted other internal fields and eventually caused the virtual machine to crash.

Another blocking issue, which at the time of writing this paper has not yet been solved, concerns the vGIC. Although the interrupt injection flow is triggered and executes correctly, the guest does not receive the interrupt. Most likely, not all necessary guest registers are updated.

E. Vector Floating Point

Another kernel subsystem which needs to be initialized as part of the boot process is the Vector Floating Point (VFP) module.

Vector Floating Point is a coprocessor which supports arithmetic operations on floating point numbers. It has a set of control registers, as well as a bank of registers which are used to store operands.

When changing worlds between host and guest, the state of this coprocessor needs to be saved. However, due to the limited use of the VFP architecture, it can be assumed that saving/restoring its state is not required at every entry/exit. Instead, the trapping mechanism is once again used. When an access to the floating point coprocessor occurs, the current state is saved and the saved state for the guest is loaded. When returning from the guest, the reverse operation is performed.

F. WFI Handler

At the very end of the boot process, FreeBSD executes a wait for interrupt (WFI) instruction. This blocks the execution until an interrupt is received by the CPU. Normally, this happens very quickly due to the timer interrupt arriving.

On the virtualized system, this instruction requires special handling. The simple method is to ignore the instruction and continue execution. However, taking into account that both the virtual interrupt controller and virtual timer were already implemented, there was no reason to choose this option.

Therefore, the hardware behaviour was mimicked as closely as possible: the virtual cpu was set to sleep and wake up periodically in order to check whether any new interrupts had arrived. While the virtual cpu is asleep, the hypervisor yields control of the cpu so that other processes may execute[2]. Upon receiving an interrupt, the virtual cpu resumes normal execution.

IV. Results

Timed events are a core element of many software systems. Their utility ranges from preempting processes while in kernel space to scheduling events in high level programming in user space. It is clear that these types of functionality are also desirable when running software in a virtualized environment.

Through the virtualization of the ARM Generic Timer described in the previous section, the guest operating system is able to achieve time-keeping and scheduling functionality close to what a system running directly on the underlying hardware.

Therefore, a key part of the operating system has been implemented with behavior which closely mimics that of the physical component.

The final result is to boot a minimal FreeBSD guest. Below is the output for each of the steps of running a virtual machine.

First, the kernel module needs to be loaded.

Listing 1. Loading the vmm module

```
# kldload boot/kernel/vmm.ko
vgic0: <Virtual Generic Interrupt
    Controller> on gic0
vgic0: Cannot setup Maintenance Interrupt
    . Disabling Hyp-Mode... 0
```

There is still an unresolved issue regarding registering the maintenance interrupt. The issue is circumvented by executing the handler upon switching from guest to host context. Also, until the issue is resolved, hyp-mode is not disabled by this failure, as it would prevent the rest of the guest execution.

The second step is creating the virtual machine. This is done using the `bhyveload` utility. At the end of this step, the virtual machine is set up and the guest code is loaded into memory.

Listing 2. Creating the virtual machine

```
# bhyveload -k kernel.bin test
lpae_vmmmap_set n: 4096 27904
lpae_vmmmap_set n: 4096 23808
lpae_vmmmap_set n: 4096 19712
lpae_vmmmap_set n: 4096 15616
lpae_vmmmap_set n: 4096 11520
lpae_vmmmap_set n: 4096 7424
lpae_vmmmap_set n: 4096 3328
lpae_vmmmap_set n: 4096 4096
```

Finally, the `bhyve` utility is used to commence the execution of the guest.

Listing 3. Booting the virtual machine

```
# bhyve -b test
initarm: console initialized
 arg1 kmdp = 0xc170fbd0
 boothowto = 0x00000000
 dtbp = 0xc1654568
 lastaddr1: 0xc1734000
loader passed (static) kenv:
 no env, null ptr
KDB: debugger backends: ddb
KDB: current backend: ddb
Copyright (c) 1992-2017 The FreeBSD Project.
Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989,
    1991, 1992, 1993, 1994
        The Regents of the University of California.
             All rights reserved.
FreeBSD is a registered trademark of The FreeBSD
    Foundation.
FreeBSD 12.0-CURRENT #0 52c583799(projects/bhyvearm)
    -dirty: Fri Jul 14 20:23:04 EEST 2017
    root@bsd:/usr/obj/arm.armv6/root/git-bhyvearm/
        sys/FVP_VE_CORTEX_A15x1_GUEST arm
FreeBSD clang version 4.0.0 (tags/RELEASE_400/final
    297347) (based on LLVM 4.0.0)
WARNING: WITNESS option enabled, expect reduced
    performance.
WARNING: DIAGNOSTIC option enabled, expect reduced
    performance.
CPU: ARM Cortex-A15 r2p0 (ECO: 0x00010000)
CPU Features:
  Multiprocessing, Thumb2, Security, Virtualization,
       Generic Timer, VMSAv7,
  PXN, LPAE, Coherent Walk
Optional instructions:
  SDIV/UDIV, UMULL, SMULL, SIMD(ext)
LoUU:2 LoC:3 LoUIS:2
Cache level 1:
 32KB/64B 2-way data cache WB Read-Alloc Write-Alloc
 32KB/64B 2-way instruction cache Read-Alloc
Cache level 2:
 512KB/64B 16-way unified cache WB Read-Alloc Write-
     Alloc
real memory  = 134217728 (128 MB)
avail memory = 101703680 (96 MB)
arc4random: no preloaded entropy cache
random: entropy device external interface
ofwbus0: <Open Firmware Device Tree>
gic0: <ARM Generic Interrupt Controller> mem 0
    x2c001000-0x2c001fff,0x2c002000-0x2c003fff on
    ofwbus0
gic0: Cannot find Virtual Interface Control
    Registers. Disabling Hyp-Mode...
gic0: pn 0xe8, arch 0x0, rev 0xe, implementer 0x800
    irqs 128
intr_pic_register(): PIC 0xc2207100 registered for
    gic0 <dev 0xc2633b80, xref 1>
intr_pic_claim_root(): irq root set to gic0
generic_timer0: <ARMv7 Generic Timer> irq 0,1,2,3 on
     ofwbus0
Timecounter "ARM MPCore Timecounter" frequency
    24000000 Hz quality 1000
Event timer "ARM MPCore Eventtimer" frequency
    24000000 Hz quality 1000
cpulist0: <Open Firmware CPU Group> on ofwbus0
cpu0: <Open Firmware CPU> on cpulist0
cryptosoft0: <software crypto>
NULL mp in getnewvnode(9), tag crossmp
Timecounters tick every 1.000 msec
WARNING: WITNESS option enabled, expect reduced
    performance.
```

```
WARNING: DIAGNOSTIC option enabled, expect reduced
    performance.
md0: Embedded image 18251776 bytes at 0xc0475f94
Trying to mount root from ufs:/dev/md0 []...
warning: no time-of-day clock registered, system
    time will not be set accurately
Jul 14 17:00:51 init: login_getclass: unknown class
    'daemon'
sh: cannot open /etc/rc: No such file or directory
Enter full pathname of shell or RETURN for /bin/sh:
    random: unblocking device.

Expensive timeout(9) function: 0xc04294b0(0xc2641600
    ) 0.022796458 s
Cannot read termcap database;
using dumb terminal settings.
#
#
```

V. Conclusions and Further Work

The project achieved its goal of completely booting a minimal FreeBSD guest operating system running inside bhyve. In order to reach this objective, a number of mechanisms were implemented. These include: the virtual generic interrupt controller, virtual timer, support for guest vector floating point operations and other less notable changes. This paper tackled especially the virtual timer.

A. Further Work

The bsd kernel contains multiple mechanisms for scheduling events to be run at a future time. The current callout system may be replaced with another mechanism if the latter offers better performance or improves code maintainability.

There are a number of directions which can be pursued for long term future development. These include: adding more hypervisor components to support more virtualization features, implementing support for symmetric multiprocessor (SMP) enabled guests.

At the time of writing this paper, there is an ongoing process with the FreeBSD community to integrate the changes proposed by the current project into the upstream repository. Alexandru Elisei started to create intermediarry patches to split-out on arch-dependent/independent the current bhyve code (libvmmapi, bhyve and vmm module). After this is done, we can create review requests for the actual ARM code.

References

[1] FreeBSD callouts. https://www.freebsd.org/cgi/man.cgi?query=callout&apropos&sektion

[2] FreeBSD msleep. https://www.freebsd.org/cgi/man.cgi?query=msleep&apropos=0&sekti
&manpath=FreeBSD+7.0-RELEASE&format=html.

[3] B. Adamczyk and A. Chydzinski. Achieving High Resolution Timer Events in Virtualized Environment. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4503740/, 2015.

[4] ARM. Cortex-A15 Technical Reference Manual, 2011.

[5] ARM. ARM Architecture Reference Manual - ARMv7-A and ARMv7-R edition, 2014.

[6] C. Dall and J. Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. https://www.cs.columbia.edu/ nieh/pubs/asplos2014_kvmarm.pdf, 2014.

[7] T. Gleixner and D. Niehaus. Hrtimers and Beyond: Transforming the Linux Time Subsystems. https://www.landley.net/kdocs/ols/2006/ols2006v1-pages-333-346.pdf, 2006.

[8] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures, 1974.

[9] VMware. Timekeeping in VMware Virtual Machines. https://www.vmware.com/files/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf, 2011.

Parallel, Multi-Axis Regression and Performance Testing with FreeBSD, OpenZFS, and bhyve

Michael Dexter
editor@callfortesting.org

AsiaBSDCon 2019, Tokyo, Japan

Contemporary Unix, defined as the sum of open source BSD Unix projects, Illumos distributions and GNU/Linux distributions, plus the OpenZFS cross-platform file system, can attribute their success to the collaborative work of like-minded academic, commercial, and volunteer developers around the world. Governed by a mix of licenses, best practices, community norms, and personal passion, open source projects like modern Unix operating systems and OpenZFS largely lack centralized Quality Engineering institutions, deferring Quality Engineering and Quality Control responsibilities to participating developers and the end user. This arrangement promises the widest-possible array of regression and performance testing tools, loads, and procedures, at the expense of providing any guarantees, true to the disclaimers of the licenses under which these projects are distributed. This paper will examine how "parallel, multi-axis" testing, defined as testing multiple software versions, operating systems, "options", compilers, and architectures, or *axes*, *in parallel*, will improve the identification and isolation of reliability and performance regressions.

Identifying a Computing Axis

Borrowing from the mathematical definition of an Axis, *a fixed reference line for the measurement of coordinates*, the quintessential *computing axis* is any given software versioning: it increments, in the case of SVN revisions, from zero to an architectural limit, where the highest number is always the latest revision and any point in the history is easily located and visited. Less-linear, yet equally traversable axes include multiple operating systems, *their* sequential versions, their userland and kernel build "options", their supported computing architectures, and their supported hypervisors. Each of these axes is of equally-unique and identifiable value, enabling for their linear traversal and most importantly, testing in *parallel*. Parallel testing is facilitated by multiple identical hardware machines, unless of dissimilar hardware architecture, and multiple virtual machines executed in parallel. While identical hardware machines will provide the greatest consistency for performance testing, virtual machines are adequate for providing meaningful reliability testing of many computing resources.

The Version Axis

Of the testing Axes within the scope of this paper, the version axis is the most familiar. Incrementing software versioning is provided manually by the developer, or automatically by a version control system. The testing host operating system for this paper, FreeBSD, provides two distinct version identifiers: Named Releases and pre-releases, i.e. 12.0-RELEASE and 12.0-RC1, and incrementing SVN Revisions, i.e. 341707. These named and numeric identifiers allow for unambiguous revision identification, in contrast to the hash-based

"numbering" semantics used in some version control systems. Two distinct challenges exists however, to obtaining open source releases by named binary Release: incomplete historical release preservation and the rising popularity of distributing open source operating systems via content delivery networks (CDNs). The first of these challenges can largely be attributed to the unavailability of terabyte and larger-capacity storage devices until late in each operating system project's history, allowing for centralized and distributed preservation of project history. The second of these challenges is simply the fact that content *distribution* networks are designed for the rapid *distribution* of the latest software releases, and not the *preservation* of historic releases. The result is a rapid expiration of available releases distributed via CDN than with traditional mirrors. Part of this paper's work is a rebuilding of the FreeBSD release history in coordination with users around the World.

The Operating System Axis

The set of available operating systems is extensive, and multiplied by their individual version axes, overwhelming. The scope of this testing will be limited to operating systems that support the OpenZFS file system with a limited number of versions and a goal of all supported architectures. The result is a focus on FreeBSD, NetBSD, Illumos derivatives, primary (not derived) GNU/Linux distributions, macOS, and Microsoft Windows.

The "Options" Axis

Of the target operating systems within the scope of this work, FreeBSD is rich with userland and kernel "source build options" that determine what features are included or excluded from the compiled operating system. Similar to the situation with FreeBSD historic releases however, these options are often under-documented or non-functional, resulting in the first test of this paper: ongoing Build Option Survey (/usr/src/tools/tools/build_option_survey/) runs, and the development of a "STUDENT" kernel configuration file that progressively introduces the options needed to build and eventually boot the FreeBSD kernel under the bhyve hypervisor.

The Compilers Axis

While FreeBSD employs the Clang compiler as its default, in-base compiler, the buildability of the operating system with the GCC and other compilers provides an important validation vector. FreeBSD 8.0 could be built with the Portable C Compiler (pcc) and this testing will facilitate the institutional compilation of FreeBSD with alternative compilers and across the versions axis. By extension, FreeBSD's promise, but not guarantee that each previous and future release of FreeBSD should be buildable under any given release, a forward/backward version axis traversal test should be trivial to conduct.

The Architecture Axis

FreeBSD offers the most OpenZFS-supported architectures of any operating system. The relative low-cost of embedded and used non-Intel machines allow for this testing to include non-Intel architectures including ARM, ARM64, PowerPC, and Sparc64. GNU/Linux distributions are candidates for inclusion when their non-Intel support expands.

The POSIX Testing Environment

Testing in parallel requires, by definition, a consistent testing environment in order to provide meaningful results. It is tempting to consider a cross-platform system orchestration solution such as Ansible or Puppet for the task of abstracting away platform-specific nuances, but these solutions provide high-overhead in exchange for limited domain-specific abilities, notably testing, rather than configuration. In consideration of the fact that the majority of the operating systems in the testing scope are near-POSIX compliant, establishing a common POSIX testing environment is the most reasonable strategy to minimize platform-specific nuances. In service of the goal of traversing the version axis on FreeBSD back to "historic" releases, a POSIX environment becomes a firm requirement for want of modern system orchestration tools on anything but the most recent operating system releases. In service of testing the Windows operating system, the Cygwin near-POSIX environment has proven the most flexible with the widest of array of third party open source packages for the Windows operating system.

With the operating system-specific ABI requirements of a POSIX environment satisfied, a base set of utilities will provide near-identical functionality on all platforms in the scope of the testing. These utilities include at a minimum `sh(1)`, `ssh(1)`, `time(1)`, `date(1)`, `touch(1)`, `dd(1)`, `truncate(1)`, `mkdir(1)`, `rmdir(1)`, `sha512(1)`, `zpool(1)`, `zfs(8)`, and `ztest(1)`. Supplementary utilities include `gdate(1)` for higher-resolution timing, and traditional benchmarking utilities like `bechmarks/fio`, `bechmarks/bonnie++`, and `bechmarks/sysbench`. Of these tools, disk partitioning utilities are the most platform-dependent, but fortunately, any discrepancy in the execution times of partitioning tools across operating systems are not relevant to to the runtime testing of a file system.

Finally, the bhyve Hypervisor and Jail containment system are essential to both preflighting tests prior to their deployment on dedicated hardware and the execution of some tests, such as those on historic versions of FreeBSD.

The ptime(1) Utility

While the POSIX standard is well established, it makes makes no guarantees as to the machine parsability of utility output. This paper proposes the `ptime(1)` or *precision time* utility to provide enhanced, machine-parsable execution reporting to standard I/O shell interpreter pipelines:

```
NAME
            ptime – Precision execution time utility

SYNOPSIS
            ptime [options] [command]

DESCRIPTION

      ptime provides Unix epoch time and utility execution time in seconds,
milliseconds and nanoseconds. It can also provide the time difference
between two files based on their datestamps.

OPTIONS
      -h          Display usage
      -s          Display output in seconds (default without -s)
      -m          Display output in milliseconds
      -n          Display output in nanoseconds
```

```
       -f          First file (requires -l)
       -l          Last file (requires -f)
       -r          Override return value with output

EXAMPLES

       Output Unix Epoch time in seconds (-s implied)

       ptime
       1544000077              <equivalent to date +%s>

       Output Unix Epoch time in milliseconds

       ptime -m
       154849734068255         <equivalent to (( gdate +%s%N ))/1000000>

       Output Unix Epoch time in nanoseconds

       ptime -n
       1548497340682551000     <equivalent to gdate +%s%N>

       Output execution time of 'sha256 -t' time in nanoseconds

       ptime -n sha256 -t
       1548497340682551000

       Output the time difference between two files

       ptime -f /build/firstfile -l /build/lastfile
       123467

       Return Unix Epoch time in seconds

       ptime -r
       echo $?
       1544000077
```

Additional tools include the `bd(8)` block device, and `be(8)` boot environment utilities for the management of block device partitioning and formatting, and OpenZFS boot environments respectively (Dexter, AsiaBSDCon 2018).

Regression and Performance Testing

Equipped with a cross-platform, near-POSIX test environment and support utilities, a baseline of tests can be performed along each axis.

FreeBSD Version and Compiler Axis: Build forward and backward versions of FreeBSD on any given version with the built-in compiler and optional compilers.

FreeBSD Option Axis: Extend the Build Option Survey framework or a new framework to kernel configuration file build options, identifying their interdependencies.

bhyve Hypervisor vCPU Topology: Validate the January, 2019 bhyve vCPU topology improvements (reviews.freebsd.org/D18815 and related) that allow for up to 65 packages/sockets and 255 cores per package. Step through additional packages and cores one by one. This test is performed with a wrapper script that simply boots a virtual machine that is designed to shut down via `/etc/rc.local`. This test should eventually traverse the operating system axis, ensuring that a representative set of non-FreeBSD operating systems are validated with difference vCPU configurations.

OpenZFS Testing along the Operating System and Architecture Axes: Perform a myriad of tests *in parallel* across the operating system axis: repeated zpool creation and destruction, nested directory and file creation, high-count file `touch(1)`ing, cross-platform pool importation, identification of SMB and NFS performance cliffs based on the amount of data transferred, scripted `fio(1)` testing, and execution of the OpenZFS `ztest(1)` suite. With new OpenZFS platforms like Windows emerging, this testing has revealed that basic assumptions cannot be made, such as the success of the `touch(1)` utility.

Conclusions

The parallel, multi-axis testing approach for regressions and performance telemetry should provide new insights into reliability and performance issues that will be overlooked by domain-specific testing. This work is inspired by real-world OpenZFS on FreeBSD performance issues and combined with a version axis bisection strategy, should identify regressions at a faster pace than is possible with traditional testing methods. This testing also aims to accelerate the stability of new OpenZFS platforms including NetBSD and Windows. Finally, all of the tools used in this testing will be available on GitHub or equivalent.

FreeBSD Virtualization - Improving block I/O compatibility in bhyve

Sergiu Weisz
University POLITEHNICA of Bucharest
Splaiul Independenței 313, Bucharest, Romania, 060042
Email: sergiu121@gmail.com

Mihai Carabaș
University POLITEHNICA of Bucharest
Splaiul Independenței 313, Bucharest, Romania, 060042
Email: mihai.carabas@gmail.com

***Abstract*—**

In a world where cloud computing and cloud infrastructures have become a mainstay, virtualization technologies have enabled a secure way to share resources with different users. A snapshoting mechanism is of great use in the area of virtualization, as it enables the backup of virtual machines, or the creation of templates for machine state replication. These virtual machines have many different virtual devices connected to them that need to have their state saved and restored for a system to be truly useful; examples include block devices, USB devices, or system time. For block I/O a virtual machine may use different types of files depending on its use case. This leads to a greater flexibility in terms of features; for example, one can use a file type that enables saving the state of the hard disk in order to be used later, or as a backup. Hypervisors like VirtualBox, VMWare and Hyper-V already have support for multiple disk file formats. This paper will present a way to implement support for the devices mentioned above, and fill readers in on the procedure of saving device states.

I. Introduction

FreeBSD is an open source operating system that is designed with the goal of being a successor to the BSD operating system, and it is the most popular OS in the BSD family. The reason in part is because of its BSD licence, which allows companies to fork the code and modify it without needing to push the modifications upstream or make them public. This makes it useful for companies like Netflix or Sony to use it in their systems, because of security concerns or financial reasons. Another reason for the popularity of FreeBSD is the performance of the network stack, that outperformes competing OS's [1].

bhyve is the hypervisor that comes packed in with FreeBSD. It is a type-2 hypervisor, so it runs over the operating system. In corporate environments bhyve is used by companies such as Joyent or iXsystems because of its support for legacy operating systems and relative small code base compared to other popular hypervisors.

We have a special interest in the snapshoting feature, because we wish to implement a fully featured checkpoint system for the bhyve hypervisor, which comes with the FreeBSD operating system. Our current implementation of the system will stop the execution of the virtual machine. These features are being worked on in an ongoing project at the University POLITEHNICA of Bucharest. This is the only such feature in the FreeBSD project [2].

This paper will present in depth the state of virtualization in the bhyve virtualization, how the snapshot and restore mechanism works, along with its stong points and flaws, and I will present the improvements I have implelementented in this mechanism, along with what problems I have had along the way.

II. State of the Art

Virtualization is the process of running an operating system over an already existing operating system. The operating system that ”runs” directly on the hardware is called a host OS, while the operating system being run over the host is called a guest operating system.

The application that manages the interaction between the host and guest operating systems is called a hypervisor. Depending on the implementation, and the level of optimization, the hypervisor might have components implemented at a kernel/driver level, or it can be fully implemented in user space.

Hypervisors can be split in two major categories by the connection they have with hardware:

- Type 1 hypervisors, which communicate with the hardware directly. ex: Xen
- Type 2 hypervisors, which communicate with the hardware through a fully fledged operating system, like FreeBSD, GNU/Linux or Wwindows. ex: Hyper-V, bhyve, KVM

Snapshoting is the act of saving the state of the virtual machine while it is running. This is done in order to make a backup, or a checkpoint, of the machine that one could roll back the machine to. Another application of this is migrating the virtual machines without shutting them off. By saving their state, moving them to another site, and starting them from the backup, you can make it look, from the point of the virtual machine, that it hasn't been turned off. Hypervisors

that implement these features are Hyper-V, VirtualBox, VMWare, qemu, and others.

In bhyve the act of saving the virtual machine is made of the following steps:

1) Stop virtual CPUs' instruction execution
2) Iterate through all the kernel structures and save their context to a file
3) Iterate through all the used devices and save their contexts to a file
4) Dump the VM memory to a file

For restoring the virtual machine state the hypervisor goes through the following actions:

1) Copy to memory the "old" memory content
2) Copy device information from the restore file
3) Copy kernel structure information stored in restore file
4) Start virtual CPUs' instruction execution

A. Block devices virtualization

A block device is a type of hardware device that is used for I/O operations. Its special characteristic is that reads and writes from it are made in discrete chunks and random access to the address space of the device. Because of the access to all the random access property, they are used for large storage devices, such as HDD's or SSD's.

In bhyve a virtualized block device is either a physical block device that is passed through to the virtual machine, or a file hosted on a physical block device. When a VM needs to access the disk, a request is filled and passed to the hypervisor. The hypervisor in turn receives the request, puts it in a worker pool, and when the time comes, it satisfies the request by calling a read or a write system call.

Currently, the only virtual disk type, also called a disk image, is the "raw" type, which acts like a normal hard drive.

As can be seen from above there is no step where a copy of the virtual storage disk is made during the checkpoint process. If you would want to use the saved machine state for backup purposes, you would have to make a copy of the whole disk image. This can be a problem when you have a large scale service where users backup their VMs like Amazon AWS, or a deployment of Openstack. This makes the need for more complex disk formats quite apparent.

III. Related work

There exist many implementations of block device abstraction. These have all been implemented in various hypervisors in order to offer a more robust and flexible interface to the users, and offer interoperability between hypervisors.

A. QEMU

QEMU [4] (Quick Emulator) is a free and open source emulator. Besides hardware virtualization, it can also do hardware-accelerated virtualization to obtain less overhead than full emulation thanks to the KVM project.

It is the emulator that has provided us with the QEMU Copy-On-Write, file format, which we have used in this paper as a proof of concept for libvdsk. This format has gone through many iterations over the years, and the code that pertains to it is the most complex for this format. It includes complex caching mechanisms that have lead to it being one of the most popular image formats.

B. Palacios

Palacios [5] is an open source VMM that is targeted towards embedded computers. It is built in such a way as to enable its integration into multiple OSs. As of the time of writing, there have not been any new commits in the Palacios main branch since 09 January 2017. It has support for multiple disk file formats, such as RAM disks, or netdisks and QCOW disks, so this present an interest to us.

Palacious VMM's block device abstraction layer works in a similar way to libvdsk. It has a system where a device registers a read, write, open and others, and depending on the file format it calls the specific implementation.

C. VMD

VMD [3] (Virtual Machine Daemon) is the OpenBSD hypervisor. It has a similar approach to the Palacios VMM of using a structure where you register callbacks to read, write and close files. Our implementation for QCOW2 operations were inspired by the implementation in this hypervisor.

IV. Implementation

The work in this paper is based on the libvdsk library, first developed by Marcel Molenaar, and brought up to date by Marcelo Araujo. It is used as an abstraction layer that enables the use of block I/O requests without being concerned about the format of the backing file used by the virtual machine. At the time of the project's start libvdsk only had support for raw image files, files that act exactly like a hard disk.

For each block I/O operation libvdsk implements a function that will be called in the block I/O interface. These functions cover common file operations like open, close, read, write, trim, flush and probe. All of these in turn use a callback to fulfill the operation received. Each file format, be it a raw disk file or

qcow2, has specific functions implemented for all the previously mentioned operations.

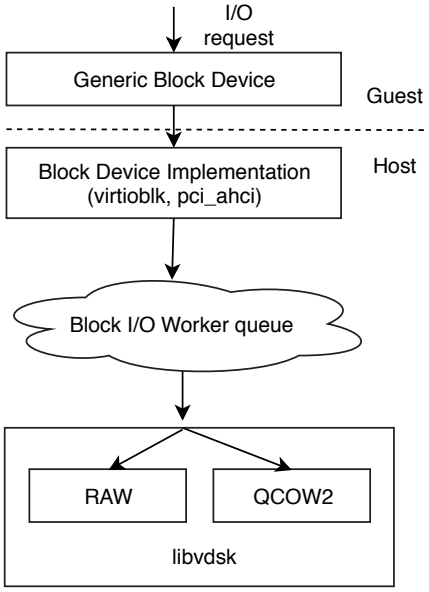


Figure 1. Libvdsk workflow

Figure 1 conveys the whole path of an I/O request through the block devices stack to reach to the libvdsk implementation of a given function. When a request is sent to a block device in the guest VM the emulator implementation (virtio-block, ahci) fills a block I/O request on behalf of the host, and it puts it in a worker queue until it is picked up by a worker thread. After a worker thread runs the job, it calls a generic function in libvdsk which will call the implemented operation for a particular disk file type.

We have been working on implementing these operations for the qcow2 image file format. This format was initially developed for the qemu emulator, but it has since gained a significant following because of features like sparse image files, snapshoting, encryption, and compression. As this is a widely used format with different applications, we have found it the perfect candidate to test libvdsk's capabilities on more complex use cases, other than the raw format that was supported natively in bhyve too.

There are multiple open source hypervisors that have integrated support for qcow2. All of these have a similar implementation to what we have put together, since there aren't many new ways in which one can read data from a file that has a well defined structure. A difference that one may observe is that the workflows for different hyervisors are different. For example, qemu implements a caching mechanism that vmd (the OpenBSD hypervisor), and bhyve do not implement.

Since libvdsk was built, and abandoned, without implicit support for more complex disk types, we needed to add necessary code to it in order to make its design more modular, like adding an additional pointer field to the structure that holds data about the disk that points to an area which has data specific to a disk implementation. This allowed us to store an extra structure related to the qcow2 internal data structures, but it can be used to store structures for any format.

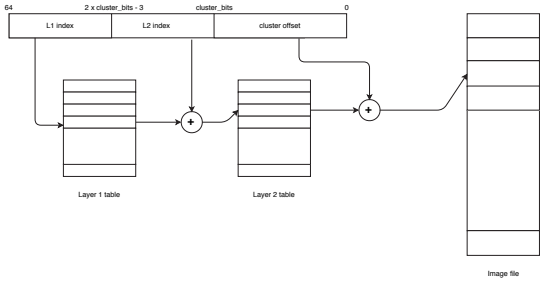


Figure 2. qcow addressing

The qcow2 image file uses a 2 layer addressing scheme, similar to the multi-level page table memory scheme, and it is also uses clusters which are effective physical storage space (analogue to memory pages). A physical address is made up of:

1) An index in the first table, named Layer 1 table (L1), that stores the address of a Layer 2 table
2) An index in the second table, named Layer 2 table (L2), that stores the address of a cluster
3) An offset in the cluster retrieved from an L2 table

This addressing scheme can also be seen in Figure 2. As can be observed in the figure, the cluster size is not a fixed value, it can vary from file to file depending on the configuration. Another detail is the fact that one L2 table is exactly one cluster long, and an entry in it is 8 bytes long.

The layered approach in qcow has de advantage of not needing a large memory allocation when creating the file. Further more, this scheme allows the existence of a Copy On Write mechanism, which is used for maintaining a backup of a disk, since all writes in a backup file will be done on a cluster by cluster basis, so these are the only storage areas that will be duplicated.

Using libvdsk we have implemented the open, probe, read and write operations on the qcow2 image format, with more support to be added in the near future, as this project is of high priority for providing the save/restore functionality with a file backup mechanism.

The read operation on qcow2 follows step by step Figure 2 in order to retrieve the cluster where the desired information is located. If the size being read is more that the cluster size, we try to determine in which part of the read operation the full cluster is located.

The write operation requires the ability to make new clusters on demand if the operation tries to write in

a space where the cluster is not allocated. This will increase the footprint of the image file on the disk as more and more clusters, and eventually L2 tables when the existing one is filled up.

V. Results

The results of the project so far are that we are able to read and write from a disk image that is in the QCOW2 format. In order to test, we converted an existing disk that had information on in to the QCOW format. After, we tested the reads at first reading trying to read the partition table of a disk. We did is by using the fdisk tool. After he did this, we read from different areas from the disk, and we checked the result with the original disk file.

For testing writes, we first tried to write in the partition table, because it is situated in the first sector, so it would rule out offset problems. After we were able to add partitions to the partition table, we checked them using the fdisk tool. After this, we formatted the new partition. We chose to format it at first with the ext2 file system, since it is a simple format that keeps its metadata at the beginning of the disk, but it doesn't have complex mechanisms, such as journalizing. After formatting the partition, we mounted it, created a file, unmounted the partition, and remounted it, to check if the file still had the written data.

As the project is still in development, the only features that have been fully tested are the hooks that go into the libvdsk. We tested this by appending printing functions to the callbacks and checking whether something in printed to the terminal.

Another feature working at the time of writing is the probe function, which prints the header of the disk image format, if it has one.

VI. Conclusion and Further Work

Furthermore, by implementing qcow2 support in bhyve using libvdsk, we aim to show a proof of concept for the library, as it aims to provide a universal way of working with and on virtual disk through an easily extensible API. This will lead in turn to possibilities for implementing varying features for working with disk image files.

Going forward, we aim to implement support for Copy on Write and disk snapshoting in our implementation of qcow2 and integrate this with the checkpointing mechanism outlined in this paper in order to allow for efficient virtual machine backup. Adding to this, we could also add a caching mechanism, similar to the implementation integrated in qemu in order to offload the overhead of translating virtual disk addresses to offsets in the disk vile.

Another avenue worth exploring is implementing support for other virtual disk files, and comparing them to the qcow2 implementation in bhyve.

Acknowledgment

The authors would like to thank Matthew Grooms for hisfinancial support in form of scholarship for Sergiu Weisz. We would also like to address a special "thank you" to Elena Mihailescu an Darius Mihai for their help with debugging various issues that we have encountered, and to Marcelo Araujo who revived the libvdsk project and who maintained continued support for this project.

References

[1] Serving 100 Gbps from an Open Connect Appliance. https://medium.com/netflix-techblog/serving-100-gbps-from-an-open-connect-appliance-cdb51dda3b99.

[2] University POLITEHNICA of Bucharest, Save and Restore Project. https://github.com/FreeBSD-UPB/freebsd/.

[3] Virtual Machine Daemon man page, vmd(8). http://man.openbsd.org/vmd.8.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. *USENIX Annual Technical Conference*, 2005.

[5] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, and P. Bridges. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. *24th IEEE International Parallel & Distributed Processing Symposium*, 2010.

ZRouter: Remote update of firmware

Hiroki Mori
yamori813@yahoo.co.jp
Editor: Mchael Zhilin

ZRouter is a system that builds FreeBSD for small targets like routers. In addition to building ordinary kernel and commands, you can create images that can be used with u-boot etc. In FreeBSD 12R, SDRAM 16M / Flash 4M is the lowest line spec. ZRouter is mainly targeting modules using SOC of mips.

Flash has built-in target, and it starts from there. This is inside of Flash.

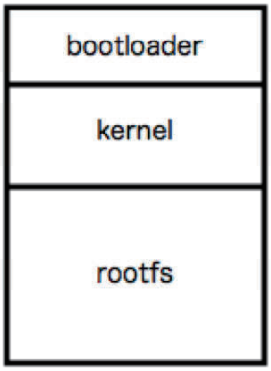


Figure 1: Flash map

Inside embedded modules there is flash memory chip, which contains image with operating system and application programs. This image is read-only and usually compressed, but ability to update image is required.

Image 1: CFI Flash chip

What is the reason why update is necessary? For instance, added new function of the program or fix program defect. It can also correspond to security considerations.

In the early stages of development, it's possible to use UART (serial interface) to force boot loader to update Flash. However, if number of devices increases or serial connection is not available at time, updating of flash image via the network improves operational efficiency.

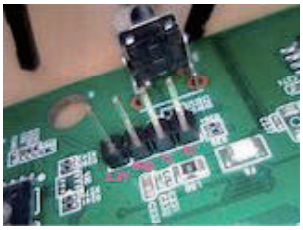


Image 2: UART Connector

Let's consider this method with SOHO (Small Office/Home Office) router based on SoC of MIPS architecture.

ZRouter project provides FreeBSD build tool with routine to update flash image. But there were two problems:

- root partition (rootfs) was mounted and there was possibility of failure while updating flash image (that contains rootfs)
- since libraries were statically linked to the routine, the binary size was enlarged

When updating flash, it is necessary to unmount root partition and resource necessary

for execution should be independent from flash.

The functionality called reroot is present in FreeBSD since 10.3 release. This functionality does not completely reset the system during reboot but it restarts boot process from mounting root partition. This behavior is provided by flag “-r” of “reroot” routine (see reboot(8)).

Using this function, I thought about a method of updating Flash by transferring all resources necessary for execution to memory.

The routine “reroot” identifies the next rootfs from kernel environment variables (also known as kenv(1)). If you put a minimalistic root partition on the memory disk (see mdconfig(8)) and set it as rootfs, you can put the flash resources in a state not used by runtime environment:

```
kenv vfs.root.mountfrom =
    cd9660:md0.uzip
```

If flash chip size is 4 or 8 megabytes, you can copy current compressed flash root partition to the memory disk as is and choose it as next root partition after “reroot” operation.

If flash chip size is 16 megabytes or more, you can copy necessary files to the memory disk and reconstruct rootfs. Because the original compressed rootfs is bigger than UFS filesystem on uncompressed memory disks.

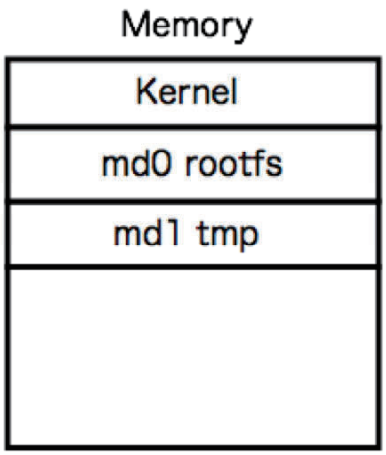


Figure 2: memory layout at flash update

Since the kernel was originally running on the memory and rootfs is also moved to the memory, functions of operating system does not affect any operation with flash chip.

Sometimes rerooting fails when target box has small amount of memory. Thanks to debugging, it has been found out that the function of reroot is using TMPFS, but there was a remaining memory check code in TMPFS and an error occurred due to lack of memory.

As solution, kernel option was introduced to reduce the check size to pass through this process (see review request D13583). If the check size is set to 1 MiB, target boxes with 16 MiB of memory can be rerooted without problem.

After rerooting, we download a new image file from the server via TFTP protocol according to the configuration file, and execute script to write image to Flash via dd & pipe & rc.

This mechanism is provided by ZRouter's profile “reupdate”. It worths to note that It is also possible to develop and execute a command that accepts HTTP download function.

For upgrading, it is necessary to define the area to write the image with geom_map or geom_flash. In Figure 3, the free space that can be written as an upgrade as a partition is taken as a partition.

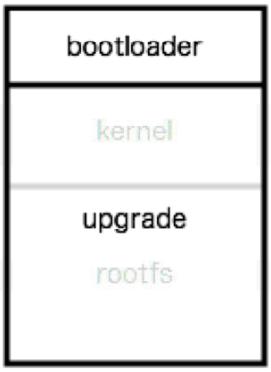


Figure 3: upgrade partition

The flash area is partitioned from geom_map (via hints) and geom_flashmap (via FDT tree). In the case of u-boot image layout, rootfs partition is placed next to kernel partition. Kernel size can be different for firmware versions, so geom_map has a scanning function to identify rootfs partition's starting address. But geom_flashmap has lack of this scanning functions. To add it, patch has been proposed (see review request D13648).

Various targets with SPI flash has been tested. It works with no a problem, but update of CFI flash failed. Patch has bee proposed, but not yet committed (see review request D14279)

ZRouter make image MD5 value at build time. It is thought that by checking the hash value of the image to be updated with the following script, it is possible to prevent security and mistakes.

```
mkfifo /tmp/tftp
MD5=`cat /tmp/tftp | md5&
echo "bin
get Fon_FON2305E_FDT.zimage /tmp/tftp
quit" | tftp 10.10.10.3 69 >/dev/null 2>&1`
echo $MD5
```

Although this mechanism can be used for environments with u-boot boot loader, it is not compatible with the environment with RedBoot boot loader. RedBoot approach is that there is no need to deal with remote control because there is a remote control function in the boot loader itself so that it can be remotely updated.

The different concerns should be considered. At first, fixability and testability: if remote update fails, it is necessary to operate with serial. Then security: if it is a closed network, there is no problem with TFTP usage, but if it is an open network it will be necessary to consider secure protocols different from TFTP. Finally, access to box: if remote update fails, serial operation is required, but if it is stable to a certain extent, remote update seems very useful.

Because it is difficult to read the whole image into memory due to memory constraint, it is difficult to perfect check and it is thought that method registration is necessary.

Consideration about correspondence with NAND memory is also necessary

Finally, this mechanism was developed thanks to the excellent build environment called ZRouter. I would like to thank Oleksandr Rybalko who started ZRouter.
I also thank you for implementing reroot.

Reference

[1] ZRouter.org
https://zrouter.org/

[2] How to put FreeBSD power into small MIPS switch/router
Oleksandr Rybalko
EuroBSDcon2012

Porting Go to NetBSD/arm64

Maya Rashish

Abstract

Go makes the unusual choice of a custom toolchain and hand-written assembly. The rationale behind some of those choices is explained and techniques used for solving problems are mentioned.

1 Introduction

Golang or Go[1] is a statically typed, compiled, garbage-collected language. It enables easy concurrency and cross-compilation. The most popular implementation of Go is self-hosted (written in Go), and is independent of libc. The choices made by Go create difficulties for adapting the compiler for new targets. This paper will discuss the adaptation of Go to a NetBSD Aarch64 machines and the difficulties involved in the process.

2 Difficulties

2.1 Custom tooling

In many languages today, the implementation can be described as the following steps:

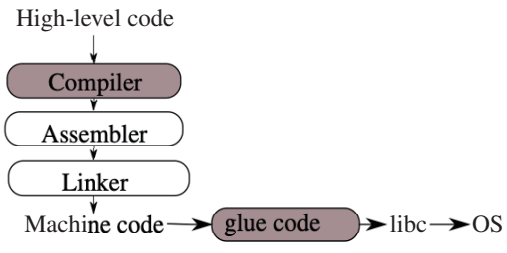


Figure 1: Typical language overview.
Language-specific parts are in brown.

[1] https://golang.org/

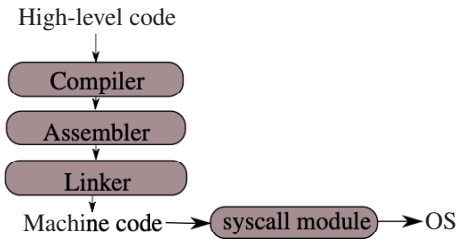


Figure 2: Go top-level overview.
Go-specific parts are in brown.

Go has chosen to take a different approach. Due to limitations to existing tools at the time, Go has chosen to incorporate tools implemented by its authors for the Plan9 project[2]. These include, unusually, a custom assembler with a special assembly syntax[3] and linker. Go doesn't use any external toolchain code.

This allows the Go code to support cross-compilation without needing to adjust any external packages, but comes at the cost of being another component that needs to be taught about new architectures.

Since the port to NetBSD/arm64 was the third OS (after Linux and Darwin), this didn't require any additional changes by the author.

2.2 CGo overhead

Stack The Go stack is not suitable for running arbitrary foreign C code, so another stack is used for foreign function calls.

ABI Go's internal function calls know not to trample certain global state. It must be saved/restored on function calls into foreign code.

Internal accounting Go manages scheduling of goroutines

[2] Plan9 operating system
[3] https://golang.org/doc/asm

and a garbage-collector. These require timing information to handle.

Go code communicates whether a function may block, and foreign code doesn't do this, so it has a special accounting state. We must communicate this to the accounting before switching to the foreign code.

3 Implementation

Custom code to call the operating system

Functions like "sleep for a few seconds" are very commonly used, so as an optimization, it is preferable to use custom Go-like code for them.

For this reason, Go had to be taught a lot of information normally contained within libc, like how to open files, how to exit. These interfaces are specific to the operating system, so code specific to NetBSD had to be written.

The majority of the effort of porting Go to NetBSD/arm64 was spent on teaching Go about how to ask the operating system to do certain things normally done in libc.

Other ports to arm64 exist, Linux and Darwin. The Linux/arm64 implementation was a source of inspiration, as was the cost for NetBSD/amd64 systems. In NetBSD, libc contains system call code and was inspected for comparison.

The Darwin implementation will call into system libraries, as Darwin doesn't offer backwards compatibility for code using system calls directly.

3.1 ABI

Using system calls is typically a matter of passing arguments in a previously agreed upon manner, and calling a special "syscall" instructions which switches into the kernel.

Typical calling convention for Aarch64 functions[4]:

SP	Stack pointer
r0..r7	Input and output registers

Most system calls within NetBSD follow the function calling convention, and would use r0 for the first input argument, r1 for the second, and so forth. Additional arguments are passed on the stack.

However not all system calls followed this convention. SYS_syscall (syscall number #0), which has the syscall number as the first argument, uses r17 for passing the syscall number.

Similarly, linux chooses to use r8 for passing the syscall number, instead of passing the paramter using the Aarch64 syscall instruction ("SVC") paramter.

[4] AArch64 Procedure Call Standard

```
mmap(0, 0x8000, 0x3, 0x1002, 0xffffffff,
0, 0) = 0x7f7ff7ef7000
open("/etc/ld.so.conf", 0, 0x7f7ff7e12768)
Err#2 ENOENT
```

Figure 3: Typical ktrace output

3.2 Debugging

The porting effort consisted of writing around 500 lines of assembly code, a very error-prone effort, prior to any testing. Unsurprisingly, the first attempt to run any code didn't work.

For the purpose of debugging, ktrace[5] was used.

System call numbers appear in their names, and the arguments are enclosed in parentheses, similar to C function calls. Return values or errors are shown after the closing paren.

ktrace was an invaluable tools, as most of the mistakes were within Go code.

4 Results

At the end, simple programs run. Additional work is done to build the compiler natively. The code is available online and is awaiting review by Go upstream. [6]

[5] Running binaries prepended by the ktruss(1) command

[6] https://github.com/golang/go/pull/29398

Improving security of the FreeBSD boot process

Kornel Duleba, Michal Stanek
Semihalf
mindal@semihalf.com, mst@semihalf.com

Abstract

This paper describes recent security additions in the FreeBSD boot process.

TPM 2.0 devices are now supported in FreeBSD. They are most often referred to in the context of Measured Boot, i.e. secure measurements and attestation of all images in the boot chain. The TPM 2.0 specification defines versatile HSM devices which can also strengthen security of various other parts of your system. We describe the basic features of TPM 2.0 which we have made available in FreeBSD. We also mention some caveats and shortcomings of the technology which may have contributed to the limited adoption of TPMs.

The article includes practical TPM use cases such as hardening Strongswan IPSec tunnels by performing IKE-related cryptographic operations within the TPM, using private keys which never leave the chip. Another example will be sealing secrets in TPM NVRAM with specific boot measurements (hashes) stored in PCR registers so that the secrets are locked in to a specific boot chain.

Furthermore, we describe our recent work on UEFI Secure Boot support in the FreeBSD loader and kernel. The loader is now able to parse UEFI databases of keys and certificates which are used to verify a signed FreeBSD kernel binary, using BearSSL as the cryptographic backend. In addition, we employ FreeBSD veriexec capability to verify various userland binaries and configuration files. We have extended veriexec with the ability to use UEFI trust anchors as base for veriexec manifest verification.

1 Introduction

There are several concepts which can improve security of the FreeBSD boot process. Measured Boot may be employed by using a hardware device called TPM (Trusted Platform Module) to make measurements of the code executed at startup and verify its integrity. Another, similar technology is UEFI Secure Boot, which makes use of cryptographic certificates stored in UEFI bootloader to authenticate the image being loaded at each step of the boot process.

After the system has booted, there may be a need for verified execution i.e. authenticating binaries and configuration files being loaded at runtime. Veriexec is a kernel component which provides such guarantees by performing file integrity checks during certain system calls such as open and exec. This ensures that an active adversary has not modified critical system files.

In this paper we present our work on TPM 2.0 support in FreeBSD, as well as integration of UEFI Secure Boot with Veriexec to allow authentication of the loader, kernel, modules and sensitive system files. We also describe several TPM features which may be used by FreeBSD userspace software for secure computation and storage. Finally, we discuss open issues concerning the current implementation of the above technologies in FreeBSD, while listing possible ways of improving it in the future.

2 TPM overview

TPM (Trusted Platform Module) is a technology specified by Trusted Computing

Group which allows systems to validate basic boot properties, such as the integrity of the boot images executed at startup. TPMs are usually discrete hardware components which provide secure storage and secure cryptographic computation. They may be classified as HSMs (Hardware Security Module) or as a limited TEE (Trusted Execution Environment). There have been two major versions of TPM specification - 1.2 and 2.0. Version 1.2 has been supported in FreeBSD since 2010. Version 2.0 is not backwards-compatible with 1.2 and introduces numerous improvements over the initial specification. We created and merged a driver for TPM 2.0 at the end of December[1]. Combined with IBM TSS library[2], which provides a software API to user applications, nearly all TPM features can now be used in FreeBSD.

TPM is most commonly used for Measured Boot. During system startup, each image which takes part in the boot chain measures (hashes) the next image before passing execution to it. The resulting hash is saved to a PCR register in the TPM. Apart from binary images, various other data can be included in the measurements - for example EFI environmental variables. The PCR registers can only be reset by firmware or during reboot. Write operation to PCR does not replace the hash value but only extends it. The TPM takes the new hash measurement, concatenates it with the current PCR value and stores the hash of the concatenation in the PCR. This way a hash chain is formed which serves as proof of a particular chain of boot images, loaded in a particular sequence. As a result, the OS may query the PCR value from the TPM and compare it to a known good hash which represents the desired system state. On mismatch, the OS finds out that some of the boot code was modified and may take appropriate action. In addition, an event log is created. It contains a list of names of hashed objects and their fingerprints. One can compare the digests against expected values and replay the process of extending PCRs to verify the log integrity.

Another TPM feature strictly linked to Measured Boot is attestation. A remote server may request measurements of the local platform as proof that the system can be trusted, before providing up keys, secrets or other information. This is achieved with the Quote operation, which consists of the TPM signing its PCR values along with a nonce provided by the requester to prevent replay attacks. The TPM signs the PCRs using a private key embedded in the chip which is unavailable to software. Several key types are supported such as RSA and ECC.

The measurements are normally made by both the firmware and OS. Depending on UEFI implementation usually all loaded binaries (*.efi files) and some variables are measured. Currently the FreeBSD loader and kernel do not support the extend operation, however depending on UEFI implementation, boot1.efi and loader.efi may be measured. There is currently no possibility to read the event log associated with the measurements in FreeBSD. Userspace software may, however, request signed PCR measurements from the TPM, which were made by UEFI.

3 TPM usage in FreeBSD

Apart of Measured Boot image measurements, there are several interesting features of the TPM which can be used in FreeBSD:

- Strongswan IPSEC

  Strongswan is a multiplatform IPsec VPN implementation available in FreeBSD. It offers optional secure storage of private keys and certificates on smartcards and TPMs. We have created patches for Strongswan which enable FreeBSD users to take advantage of the TPM plugin to secure their VPN networks. We describe an example Strongswan configuration using TPM in later sections.

- Secure NVRAM storage

  TPMs typically contain limited NVRAM memory in the order of several kilobytes for storing secrets and other data. NVRAM

data in the TPM may be locked to a specific password, pin code or a specific set of PCR values. Some operating systems support storing disk encryption keys in TPM NVRAM. Common examples are Bitlocker and LUKS. FreeBSD GELI does not support storing encryption keys in the TPM.

- Data sealing

  The TPM offers a Seal operation which allows encryption of arbitrary user data by the TPM using embedded symmetric keys. The data may also be sealed to a particular set of PCR values. Note that TPM cryptographic operations are usually much slower than software. Therefore, for large data it is best to use the TPM to seal a symmetric key which may then be kept on vulnerable storage in encrypted form and later used to decrypt the actual user data in software.

- SSH key storage

  It is possible to store SSH private keys in the TPM NVRAM. A third-party library is required which works as a PKCS11 provider. We did not evaluate this option.

4 TPM limitations and caveats

One of the biggest shortcomings of discrete TPM chips is their performance, especially regarding asymmetric cryptography. In case of Infineon SLB9665 the operation of signing SHA256 digest with a RSA-2048 key takes approximately 0.15s. TPMs are not supposed to be cryptographic accelerators. Better performance may be achieved with a non-discrete TPM implementation. As an example, fTPM 2.0 is available as part of Intel Management Engine in 5+ gen processors. Another firmware implementation of TPM was created by Microsoft[3]. It leverages ARM TrustZone and is used in all ARM mobile devices running Windows.

Another limitation is the size of NVRAM storage. Usually it is in the order of several kilobytes, which is hardly enough to store a few RSA keys. Infineon SLB9665 contains exactly 7206 bytes of NVRAM.

Since the TPM 2.0 specification was introduced fairly recently, some of the software interacting with it may still lack important features. For example IBM TSS library lacks support for encrypted communication with the TPM, a feature that is defined in the specification. Also, there is no in-tree support for TPM in OpenSSL.

Most TPM implementations are closed-source with limited public information available on the specifics of the hardware. This is often the reason for common mistrust in TPMs and their vendors. Some major companies, however, have released their open-source TPM implementations including Microsoft[4] and Google.

Bus communication with the TPM is not encrypted by default. The most common bus used with discrete TPMs is LPC on Intel systems (I2C/SPI on others) which is a low speed bus integrated in the Southbridge. The TPM specification allows encryption of sensitive parameters within a command (not the entire bus traffic) with AES-CFB together with HMAC for authentication and protection against tampering, as well as rolling nonces for protection against replay attacks. The keys for encryption and authentication must initially be uploaded to the TPM on a trusted system.

TPM boot integrity measurement functionality contains several caveats. It is less flexible than Secure Boot as every TPM object protected by PCR-based authorization policy must be replaced each time a boot image or configuration included in PCR measurements has been updated. Furthermore, security of Measured Boot relies on the assumption that PCR registers are only reset when the entire system is reset. If there is any vulnerability in the software, OS or TPM which allows the attacker to issue a reset to the TPM without resetting the rest of the system, then the PCRs get zeroed out and the attacker may then replay the proper hash sequence into the PCRs, spoofing the integrity

measurements and possibly unlocking secrets. This has been the major method used in attacks on Measured Boot[5].

5 UEFI Secure Boot and the FreeBSD loader

UEFI Secure Boot is a method of ensuring that only authenticated boot images are allowed to run on the system. The security goals of Secure Boot are similar to TPM Measured Boot, however there are significant differences between the two technologies. TPM Measured Boot performs hash measurements of subsequent boot images and configuration, without disturbing the boot process. It is up to the OS and user software to verify PCR measurements and take appropriate action. For example, one could include remote attestation, in which a remote server verifies signed PCR values against its own database using Quote operation[6]. The TPM may also release secrets based on specific PCR values.

UEFI Secure Boot, on the other hand, never passes execution to an unauthenticated image. Furthermore, instead of only hashing, it verifies certificates and signatures of the images it loads, with trust anchors and revoked certificates stored in flash - DB and DBX variables respectively.

UEFI Secure Boot provides an important advantage over TPM Measured Boot from the system administration point of view. In Measured Boot, PCR values will change on firmware update. This requires that any software and/or OS which relies on specific PCR values must be updated with new configuration. Thanks to the usage of certificates in Secure Boot, one must simply sign the new firmware image and it will be successfully authenticated. Certificate chains can be employed to allow for handling of complex certificate trust schemes.

Another significant difference is that UEFI Secure Boot does not require the use of TEE (Trusted Execution Environment) for secure operations, although some ARM systems make use of TrustZone technology along with Arm Trusted Firmware to separate sensitive operations from main execution mode. TPM, however, may be regarded as a TEE as it offers both secure cryptographic operations and isolated secure storage.

The FreeBSD EFI loader is a regular EFI application which can be verified by UEFI as part of Secure Boot. The simplest approach of including the FreeBSD kernel in Secure Boot is to bundle the kernel into the loader binary and make UEFI verify the whole package. The loader would find the kernel image embedded in its binary and pass execution to the kernel. The ideal solution, however, would be to modify the EFI loader to be able to read UEFI certificate lists and verify the kernel on its own. This would ensure the proper chain of trust where each boot element is clearly separated and verifies the next image to be loaded. It would also eliminate the necessity to rebundle the loader with the kernel on each kernel update.

The latter approach is implemented with the help of Juniper veriexec[7].

6 FreeBSD Veriexec and UEFI

Veriexec is a system developed by Juniper Networks which allows restricting execution only to verified code. It uses a signed manifest which is a list containing a path and hash for each verified file. The manifest is passed over to /dev/veriexec char device by /sbin/veriexec[8]. Veriexec consists of two parts, mac_veriexec[7] - a kernel module that manages verification during runtime and libsecureboot[9] which is used by the loader.

Up to now the authenticity of the manifest could only be verified using an embedded trust anchor.

In our contribution we extend it to load trust anchors from UEFI and implement a revocation system which is also based on data stored in firmware[10].

Manifest verification is done in userspace, which opens a time window for someone to inject their own malicious data. To fix this issue, we introduce a kernel module which loads a manifest based on data passed by the loader through environmental variables[11]. This allows verifying the signature of the manifest in kernel space (by the loader) which eliminates previous security concerns.

7 Acknowledgements

The work on TPM 2.0 driver support, libsecureboot and UEFI Veriexec support in FreeBSD was initiated and sponsored by Stormshield who also provided reference hardware.

Work on this paper was sponsored by Semihalf.

8 References

[1] FreeBSD TPM 2.0 driver `https://svnweb.freebsd.org/base/head/sys/dev/tpm/`

[2] IBM TSS `https://sourceforge.net/projects/ibmtpm20tss/`

[3] fTPM: A Software-only Implementation of a TPM Chip `http://ssaroiu.azurewebsites.net/publications/usenixsecurity/2016/ftpm.pdf`

[4] Microsoft TPM 2.0 reference implementation `https://github.com/Microsoft/ms-tpm-20-ref`

[5] Attacks on Measured Boot `https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-han.pdf`

[6] IBM TPM attestation `https://sourceforge.net/projects/ibmtpm20acs/`

[7] mac_veriexec `https://svnweb.freebsd.org/base/head/sys/security/mac_veriexec/`

[8] /sbin/veriexec `https://reviews.freebsd.org/rS344567`

[9] Juniper libsecureboot `https://reviews.freebsd.org/rS344565`

[10] Veriexec UEFI support `https://reviews.freebsd.org/D19093`

[11] Veriexec in-kernel manifest parsing `https://reviews.freebsd.org/D19281`

Another Path for Software Quality? Automated Software Verification and OpenBSD

Moritz Buhl
genua GmbH
Ludwig-Maximilians-Universität München
mbuhl@moritzbuhl.de

Abstract

CPACHECKER is a platform for software-verification created to be extensible by implementing an interface of configurable program analysis. It comes with various configurations for checking a program for correctness or to produce a counterexample after falsification. This paper displays the strengths and weaknesses of CPACHECKER based on the key insights gained from the *Application of Software Verification to OpenBSD Network Modules* [1]. This is to provide a view on the applicability of formal verification on the OpenBSD source code with the goal of improving its quality.

1 Introduction

The OpenBSD project is proud to ensure a high quality standard. This is due to a combination of a strong review policy, guiding development tools as well as additional audits in the BSD community [2], performance and regression tests[1] [3], and other testing methods like fuzzing [4]. Further promising approaches are applied within the BSD community, e.g. kernel sanitizers [5].

However, a rather unproven way of ensuring software quality – that stands in contrast with the usual approach of simplicity in the OpenBSD project – is formal verification. The amount of effort associated with verification and specification as well as the complexity of time and space limitations usually outgrow the benefits of formally proven source code.

Previous work [1] shows that in addition to the usual weaknesses of automated verification, the OpenBSD kernel causes more complications because of assumptions by the tooling based on a GNU/Linux exclusive view on UNIX. E.g. the runtime environment model used during verification is Linux based and CPACHECKER is officially only running on Linux. Further problems emerge since the BSD kernels differ from Linux in their internal implementations. Moreover, the use of inlined assembler is unsupported and therefore the extensive use of assembler adds more complications.

On the other hand, formal verification offers promising analyses for memory and concurrency safety, reachability and termination checks and also overflow detection. Which means that many program defects can be ruled out or counterexamples can be generated. And as the Linux Driver Verification (LDV) project[2] shows, these methods can be applied successfully.

One tool that enables verification and falsification is CPACHECKER. It uses a combined approach of static code analysis and model checking by implementing an interface for program analysis to test if a program satisfies a given specification.

2 CPAchecker

2.1 Configurable Program Analysis

CPACHECKER implements an interface for configurable program analysis (CPA) [6], [7]. A CPA consists of abstract program states, a relation that connects the abstract state to the program code, and two operations, one that decides on new abstract states, and another one to decide when the analysis will terminate. The program code is converted from the source code to a control-flow graph (CFG) to apply a graph algorithm.

Each CPA has different trade-offs in accuracy and supported features. It is possible to combine multiple CPA and use them in combination on different parts of

[1] http://bluhm.genua.de

[2] http://linuxtesting.org/ldv

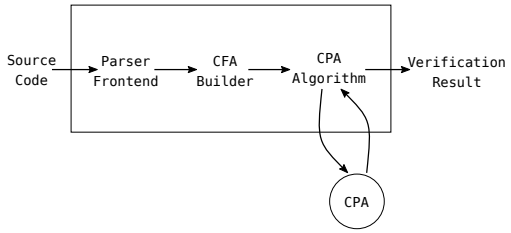


Figure 1: The workflow of CPACHECKER

```
Inline assembler ignored, analysis is probably
    unsound!
Assuming external function err to be a pure
    function.
Function pointer *(&putc) with type int (*)(
    int, FILE *) is called, but no possible
    target functions were found.
Unrecognized C code ...
Using unsound approximation of ints with
    unbounded integers and floats with
    rationals for encoding program semantics.
```

Figure 2: Example error messages from CPACHECKER

the program, this is called partial verification. Examples for the various features of a program and how these affect the tooling can be viewed online[3].

Figure 1 shows the process of verifying a program: first, its source code is converted to a CFA. Then a graph algorithm runs a (possibly multiple) CPA on it and checks if the program satisfies the given specification. If multiple CPA are used it is possible to make them interact with each other or to refine the analysis based on previous results [8].

2.2 Setup and Usage

CPACHECKER[4] uses the Java JVM and therefore is mostly platform independent with exception to the dynamic libraries that are not currently available on OpenBSD. To set up CPACHECKER, a Java environment is needed and the sources need to be built with Apache Ant. On OpenBSD adjusting the maximum data size limit in the `login.conf` file is recommended, especially when analyzing bigger C programs. A port is currently not available.

Once set up, CPACHECKER requires a specification or a configuration file to analyze a C program: `cpa.sh [-config CONFIG_FILE] [-spec SPEC_FILE] SOURCE_FILES`. A selection of configurations is available in the `config` folder.

Source code requires preprocessing before it can be analyzed. GCC and Clang provide an `-E` option to use the C preprocessor on a source file, alternatively the `-preprocess` switch of CPACHECKER can be set. To preprocess the OpenBSD kernel code according to its configuration, it is necessary to use the kernel build system. The necessary modifications are available online[5].

On OpenBSD the following properties either need to be set in a configuration file or with the `-setprop` flag, as other SMT solvers are not available:

```
solver.solver                       SMTInterpol
cpa.predicate.encodeBitvectorAs     INTEGER
cpa.predicate.encodeFloatAs         INTEGER
```

While using CPACHECKER, it is likely to run into the warnings and errors mentioned in Fig. 2. Especially when working with kernel code, warnings will occur due to inlined assembler. These places need to be looked at individually and need to be worked around. Other warnings require implementations of known C functions because a mechanism for linking files is missing. And other times either an analysis cannot work with a specific C construct or the conversion to a CFA was erroneous. The last warning mentioned appears on OpenBSD because other SMT solvers are not supported.

2.3 Working around Problems

To still receive a result from CPACHECKER, it is necessary to work around these errors. This is mostly achieved by adjusting the source code – which means changing it to an extent that it is not easy to prove to behave exactly the same. E.g. by replacing a call to a function pointer with the actual function because function pointers are not always tracked. The LDV project too has to work around the weaknesses and does so by adding another compilation layer with an intermediate language.

Application of Software Verification to OpenBSD Network Modules [1] had the plan to refind previous errata with the prospect of applying the same strategies in the future to find new bugs. But it quickly became clear that CPACHECKER is not practicable for this task as it is not battle-tested on real source code. It is possible

[3] `http://sv-comp.sosy-lab.org`
[4] `http://cpachecker.sosy-lab.org`
[5] `http://github.com/bluhm/preproc`

to refind errors like a double-`free(3)` after swapping the memory management implementation, removing inline assembler, replacing calls to function-pointers, fixing C syntax parsing in CPACHECKER and manually merging compile units together.

3 Conclusion

Using CPACHECKER to easily find new bugs is currently not imaginable. It is possible to reproduce already known bugs in the kernel but as the kernel functions used for resource management require individual abstractions, this is associated with manual labor.

In addition to this, the bugs in CPACHECKER make it uneasy to use on real programs. It tries to support the ISO/IEC 9899:1999 (C99) standard and does so by reprogramming the bugs, other compilers and parsers already made. The manifold configurability is great for developing new approaches but might cause problems with usability when applying it on real programs, as the barrier to entry is increased with the knowledge required on each CPA. The lack of use with real programs is the main reason for this.

4 Future Work

Because of the mentioned problems, it is necessary to use CPACHECKER on real userland programs. Starting with small POSIX programs like `true`, `yes` or `w` before considering to take a look kernel code might be a better approach to fix CPACHECKER.

Especially since OpenBSD introduced a dynamic approach with `pledge(2)` that ensures that a system call cannot be called again after a pledge, it would be interesting to see if a CPA can find violations of a pledge with a static approach.

Another complicated problem that accumulates a lot of different possible states that is known to be tricky is multiprocessing. Approaches in CPACHECKER exist to verify POSIX thread programs. If this can be applied successfully on userland programs, it might be interesting to see, if a similar approach can be used on the kernel.

References

[1] M. Buhl, *Application of software verification to OpenBSD network modules*, Bachelor's Thesis, LMU Munich, Software Systems Lab, Sep. 2018. [Online]. Available: `http://www.moritzbuhl.de/bachelor-thesis/thesis.pdf`.

[2] M. Villard, *Network security audit*, Article, May 2018. [Online]. Available: `http://blog.netbsd.org/tnf/entry/network_security_audit`.

[3] J. Klemkow, *Openbsd testing infrastructure behind bluhm.genua.de*, Presentation, Paris: EuroBSDcon, Sep. 2017. [Online]. Available: `http://klemkow.org/eurobsdcon_2017_obsd_test_infrastructure.pdf`.

[4] A. Lindqvist, *Fuzzing the openbsd kernel*, Presentation, BSD Users Stockholm, Sep. 2018. [Online]. Available: `http://www.openbsd.org/papers/fuzz-slides.pdf`.

[5] S. Muralee and K. Rytarowski, *Taking netbsd kernel bug roast to the next level : Kernel sanitizers*, Presentation, Bucharest: EuroBSDcon, Sep. 2018. [Online]. Available: `http://netbsd.org/~kamil/eurobsdcon2018_ksanitizers.html`.

[6] D. Beyer, T. A. A. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis", in *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007, Berlin, July 3-7)*, W. Damm and H. Hermanns, Eds., ser. LNCS 4590, Springer-Verlag, Heidelberg, 2007, pp. 504–518.

[7] D. Beyer and M. E. Keremoglu, "CPACHECKER: A tool for configurable software verification", in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011, Snowbird, UT, July 14-20)*, G. Gopalakrishnan and S. Qadeer, Eds., ser. LNCS 6806, Springer-Verlag, Heidelberg, 2011, pp. 184–190. [Online]. Available: `https://cpachecker.sosy-lab.org`.

[8] D. Beyer, S. Löwe, and P. Wendler, "Refinement selection", in *Proceedings of the 22nd International Symposium on Model Checking of Software (SPIN 2015, Stellenbosch, South Africa, August 24-26)*, B. Fischer and J. Geldenhuys, Eds., ser. LNCS 9232, Springer-Verlag, Heidelberg, 2015, pp. 20–38. [Online]. Available: `https://www.sosy-lab.org/research/cpa-ref-sel/`.