Hint: Use Ubuntu Host installation

#1 Push Debug and general Information from https://github.com/theuema/docs_zynq_yocto.git
#2 Code is on: https://github.com/theuema/qemu.git

```
      $ git checkout theuema_dev
```

Set up and Run xilinx zynq7 emulation in provided qemu version

Pre Info & Required packages for qemu emulation:

https://www.yoctoproject.org/downloads

```
-> git clone -b pyro git://git.yoctoproject.org/poky.git
```
/ clone pyro version of YP Core

Implies checking out the pyro version of poky.

https://layers.openembedded.org/layerindex/branch/master/layers/

```
-> git clone git://git.yoctoproject.org/meta-xilinx
```
/ clone xilinx meta layer for yocto

The meta-xilinx layer provides additional support for Yocto/OE, adding recipes for various components, refer to their README for additional details.

Consider Yocto Documentation Site: https://www.yoctoproject.org/documentation
Especially http://www.yoctoproject.org/docs/2.3.1/mega-manual/mega-manual.html
-> Required build host packages vary depending on your build machine and what you want to do with the Yocto Project.
The following list shows the required packages needed to build an image that runs on QEMU in graphical mode

sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib \ build-essential chrpath socat cpio python python3 python3-pip python3-pexpect \ xz-utils debianutils iputils-ping libsdl1.2-dev xterm	build host packages needed for yocto

Building an Image for Emulation

```
-> cd ~/poky
-> checkout correct branch (already done)
-> (git checkout -b pyro origin/pyro)

-> source oe-init-build-env /path/to/buildfolder
```
/ Initialize build environment with script

It showed that you better not use the home directory. I also recommend to have 50GB+ on your partition.
My build folder is: `/working/build_yocto_pyro`

(don’t forget to grand permission to your user `sudo chown username: /myfolder`)
From `path/meta-xilinx(master) » vim README.building.md`:

Build Instructions

==================

The following instructions require OE-Core meta and BitBake. Poky provides these
components, however they can be acquired separately.

Initialize a build using the `oe-init-build-env` script. Once initialized configure `**bblayers.conf**` by adding the `meta-xilinx` layer. e.g.:

```
BBLAYERS ?= " \
(<path to layer>/oe-core/meta \) -> not used
<path to layer>/meta-xilinx \
"
```

To build a specific target BSP configure the associated machine in `**local.conf**`:

```
MACHINE ?= "qemu-zynq7"
```

! bitbake sanity error because of inital block of Domain: “example.com”
! this results in not building the kernel via bitbake.
-> switch to poky build folder and create empty sanity config file:
-> `touch sanity.conf`

Build the target file system image using `bitbake`:

```
$ bitbake core-image-minimal
```

-> Needs a few hours to build.

Once complete the images for the target machine will be available in the output
directory `tmp/deploy/images/<machine name>/`.

Additional Information

For more complete details on setting up and using Yocto/OE refer to the Yocto
Project Quick Start guide available at:
http://www.yoctoproject.org/docs/current/yocto-project-qs/yocto-project-qs.html

http://www.wiki.xilinx.com/QEMU+Yocto+Flow / Xilinx QEMU Yocto Flow

#Info

Important files used by bitbake to build our target yocto kernel.

`/build_folder/conf`

-> bblayers.conf
-> local.conf

First Test:
-> `runqemu qemu-zynq7`

#Info

All these useful scripts like “`runqemu`” can be found in the “`scripts`” subdirecotory of the poky folder!

The runqemu script launches QEMU in single-processor mode. Adding qemuparams="-smp 2" starts QEMU with two processor cores.

```
-> runqemu qemu-zynq7 qemuparams="-smp 2"
```

After QEMU boots your Linux system, log in as root user and execute from the command line:

```
cat /proc/cpuinfo | grep processor
```

#Info

```
STRG+A -> X (die Emulation geht in den “Attention Mode” und mit X kann man die Emulation beenden)
```

Although QEMU provides two processor cores, only one processor is shown. You can perform the test again by re-enabling SMP using the menu editor. This time, the command shows two processors.
Running Emulation with my own QEMU:

Files needed for qemu evocation to find in build folder:
`/working/build_yocto_pyro/tmp/deploy/images/qemu-zynq7`

runqemu script command:

```
tmp/work/x86_64-linux/qemu-helper-native/1.0-r1/recipe-sysroot-native/usr/bin//qemu-system-aarch64 / qemu executable

-net nic,netdev=net0,macaddr=52:54:00:12:34:02 -netdev tap,id=net0,ifname=tap0,script=no,downscript=no -initrd

tmp/deploy/images/qemu-zynq7/core-image-minimal-qemu-zynq7-20170725140338.rootfs.cpio  / rootfs → ramdisk
-nographic -serial null -serial mon:stdio  -machine xilinx-zynq-a9  -m 1024 -kernel

tmp/deploy/images/qemu-zynq7/uImage--4.9-xilinx-v2017.1+git0+68e6869cfb-r0-qemu-zynq7-20170725140338.bin / kernel image
-append 'root=/dev/ram0 rw debugshell  mem=1024M ip=192.168.7.2::192.168.7.1:255.255.255.0 console=ttyPS0,115200 earlyprintk '

-dtb tmp/deploy/images/qemu-zynq7/qemu-zynq7.dtb / Device Tree
```

#Info on dtb:
Simply put, device tree is a data structure describing a hardware platform. Rather than hardcoding every detail of devices and their configuration, such as I/O addresses, memory address space, interrupts, and more, into the kernel sources, a data structure is passed tothe kernel at boot time. *Reading Stuff: Section 9.5 - Embedded Linux Systems with the Yocto Project .pdf*

./buildqemu script provided with **docs_zynq_yocto.git**
gdb command:

```
gdb ${ARM64SMMUFOLDER}/qemu-system-aarch64 -ex "break $param" -ex "run -initrd
${ZYNQDEPLOY}/core-image-minimal-qemu-zynq7-20170804100356.rootfs.cpio -serial null -serial stdio -machine xilinx-zynq-a9_enc -m
1024 -kernel ${ZYNQDEPLOY}/uImage--4.9-xilinx-v2017.1+git0+68e6869cfb-r0-qemu-zynq7-20170725140338.bin -append
'root=/dev/ram0 rw debugshell  mem=1024M ip=192.168.7.2::192.168.7.1:255.255.255.0 console=ttyPS0,115200 earlyprintk ' -dtb
${ZYNQDEPLOY}/qemu-zynq7.dtb" -ex "tui enable"
```

→ **CHANGED RUNQEMU SCRIPT**! (see `runqemu vs. runqemu_backup` script in provided repository)

META XILINX Layer ändern des Boardnamens um eigenes zu verwenden

Brauche ein eigenes Recipe.
// meta-xilinx -> recipes-bsp/device-tree/files/qemu-yznq7.dts
→ Unter “compatible” den Namen abändern.
→ builden // bitbake core-image-minimal

cd /working/build_yocto_pyro/tmp/deploy/images/qemu-zynq7
strings qemu-zynq7.dtb | grep -i qemu

```
$ qemu,xilinx-zynq-a9_enc
$ &Zynq A9 QEMU
```

QEMU Änderund und eigenes Board

qemu source code -> hw/arm/zynq
→ neues file erstelln (zynq board kopieren)
→ letzte zeile

```
DEFINE_MACHINE("xilinx-zynq-a9", zynq_machine_init)
```

auf

```
DEFINE_MACHINE("xilinx-zynq-a9_enc", zynq_machine_init_enc)
```

umändern!

Builden..
Check ob er es verwendet… -> **nein**

#Info

QEMU:
Letzte init Methode in den HW’s (/hw/zync) werden immer aufgerufen:
```
************* CORRECT BOARD INITIALIZED: xilinx-zynq-a9_enc
************* Every Board is initialzied: Raspberry Pi 2
************* WRONG BOARD INITIALIZED: xilinx-zynq-a9
```

Problem

```
qemu-system-aarch64: -initrd tmp/deploy/images/qemu-zynq7/core-image-minimal-qemu-zynq7-20170802124531.rootfs.cpio:
unsupported machine type
Use -machine help to list supported machines
```

```
runqemu -machine help
```

Usage: you can run this script with any valid combination
of the following environment variables (in any order):
 KERNEL - the kernel image file to use
 ROOTFS - the rootfs image file or nfsroot directory to use
 MACHINE - the machine name (optional, autodetected from KERNEL filename if unspecified)

Versuch 1

rename in `meta-xilinx/conf/machine/qemu-zynq7.conf`

QD_Machine = “- machine xilinx-zynq-a9“
to
QD_Machine = “- machine xilinx-zynq-a9_enc“

qemu invocation -> ………… -machine xilinx-zynq-a9_enc -m 1024 -kernel tmp/deploy/images/qemu-zynq7/uImage--4.9-xilinx-v2017.1+git0+68e6869cfb-r0-qemu-zynq7-20170725140338.bin -append 'root=/dev/ram0 rw debugshell mem=1024M ip=192.168.7.2::192.168.7.1:255.255.255.0 console=ttyPS0,115200 earlyprintk ' -dtb tmp/deploy/images/qemu-zynq7/qemu-zynq7.dtb

seems to work.

Erstellen der Files

```
include/sysemu/xilinx_mem_enc.h
working/qemu/xilinx_mem_enc.c
```

Ausprogrammieren einer Methode `memory_region_allocate_system_enc_memory_region`() welche `memory_region_ram_init_ops`() aufruft. Zweitere ist eine Mischung aus der bekannten `memory_region_init_ram_device_ptr`() aus der memory.c und ähnlichen Methoden.

Erstellen der memory ops. Wichtig ist dass das `ram_device` gesetzt ist, ansonsten werden die ops nicht aufgerufen.

Problem des Kernel, rootfs und dtb

.rootfs.cpio / rootfs → ramdisk
uImage--4.9-xilinx-v2017.1+git0+68e6869cfb-r0-qemu-zynq7-20170725140338.bin / kernel image
qemu-zynq7.dtb / Device Tree

→ Dieser Code wird von QEMU unverschlüsselt auf meinen ext_ram geladen.

// *INFO* überarbeitet, siehe "Umstrukturierung & derzeitiger Stand"
Im Falle des `uBoot` Images über die Funktionen in `loader.c`:
`load_uboot_image()`
`rom_add_blob_fixed_as()`
`rom_add_blob()`

Es liegt nahe dass auch `rootfs` und `dtb` mit `rom_add_blob()` in den RAM geladen werden.
Modifkation mit Encryption muss auch dort eingebunden werden.

ROOTFS Encryption:

Adding Encryption for rootfs.cpio via `load_ramdisk()`:

`loader.c` <line 933> func: `rom_add_file()`
`apply_crypt(addr, rom->data, rom->datasize);`

incl. restriction with smaller addresses than `0x8000000` (start of rootfs) in `apply_crypt()`

```
if(addr < 0x8000000)
   goto crypt_done;
```

leads to:

```
qemu-system-aarch64: Trying to execute code outside RAM or ROM at 0x00000000ffff0010
This usually means one of the following happened:

(1) You told QEMU to execute a kernel for the wrong machine type, and it crashed on startup (eg trying
to run a raspberry pi kernel on a versatilepb QEMU machine)
(2) You didn't give QEMU a kernel or BIOS filename at all, and QEMU executed a ROM full of no-op
instructions until it fell off the end
(3) Your guest kernel has a bug and crashed by jumping off into nowhere

This is almost always one of the first two, so check your command line and that you are using the right
type of kernel for this machine.
If you think option (3) is likely then you can try debugging your guest with the -d debug options; in
particular -d guest_errors will cause the log to include a dump of the guest register state at this
point.

Execution cannot continue; stopping here
```

Umstrukturierung & letzter Stand

Für unten erwähnte Files etc:

#1 Push Debug and general Information from https://github.com/theuema/docs_zynq_yocto.git

Wichtige Methoden, ***Memory read & write:***

memory.h

address_space_read()

exec.c

address_space_read_continue()

address_space_write_continue()

memory_region_dispatch_read()

memory_region_dispatch_write()

Überall dort wurden meine encryptin/decrypting Methoden eingefügt.
Ein guter Anhaltspunkt für eine read oder write - Operation auf meinem Address Space ist die suche nach "*mr->ram_block*";
-> demnach sollten alle Memory Zugriffe mit derzeitigem stand en- bzw. decrypted werden.

Zugrüffe über dispatch_read und dispatch_write werden über meine folgenden Methoden behandelt:

xilinx_mem_enc.c

memory_region_ram_read()

memory_region_ram_write()

Probleme

exec.c

cpu_physical_memory_write_rom_internal()

läd alle ROM's die zum booten gebraucht werden in folgende Speicherbereiche meines "ext_ram":

~ [No Crypt] addr: 0; size: 40; type cpu_physical_memory_write_rom_internal;
NAME: "Bootloader"

~ [No Crypt] addr: 100; size: 44; type cpu_physical_memory_write_rom_internal;
NAME: "Board Setup"

~ [No Crypt] addr: 8000; size: 3598640; type cpu_physical_memory_write_rom_internal;
NAME: "uImage.bin"

~ [No Crypt] addr: 8000000; size: 5112320; type cpu_physical_memory_write_rom_internal;
NAME: "rootfs.cpio"

~ [No Crypt] addr: 84e1000; size: 63890; type cpu_physical_memory_write_rom_internal;
NAME: ".dtb"

Diese Images werden derzeit nicht encrypted.
Speicher bis zu zur Adresse *0x84F0992* ist ausgenommen da ansonsten der Bootvorgang nicht startet.

INFO: in case *WRITE_DATA*:
wird prinzipiell auch eine Encryption der ROM's durchgeführt werden, dazu folgende condition entfernen:
xilinx_mem_enc.c
if(addr <= 0x84F0992 || addr > 0x3fffffff)

Anscheinend kopiert der Bootloader oder das Board Setup Instruktionen die zum booten benötigt werden an bestimmte stellen unseres ext_ram. Debug findings:

Adressen die während des bootens aufgerufen werden OHNE vorheriges write von QEMU:

<0x84F0992

```
~ [No Crypt] addr: 4000; size: 4; type mem_op_read;
~ [No Crypt] addr: 400c; size: 4; type mem_op_read;
~ [No Crypt] addr: 4030; size: 4; type mem_op_read;
~ [No Crypt] addr: 4008; size: 4; type mem_op_read;
~ [No Crypt] addr: 402c; size: 4; type mem_op_read;
~ [No Crypt] addr: 4004; size: 4; type mem_op_read;
~ [No Crypt] addr: 4028; size: 4; type mem_op_read;
~ [No Crypt] addr: 4024; size: 4; type mem_op_read;
~ [No Crypt] addr: 4010; size: 4; type mem_op_read;
~ [No Crypt] addr: 4014; size: 4; type mem_op_read;
~ [No Crypt] addr: 4018; size: 4; type mem_op_read;
~ [No Crypt] addr: 401c; size: 4; type mem_op_read;
~ [No Crypt] addr: 4020; size: 4; type mem_op_read;
```

too much to show

siehe file:
boot_success_s0x84F0992_addr_output_smaller_cleaned.txt

Adressbereich: 0x400 - 0x7ffc

rootfs bereich, hat vorheriges write;
könnte wsl. encrypted werden:

```
~ [No Crypt] addr: 80c3004; size: 4; type mem_op_read;
~ [No Crypt] addr: 80c3200; size: 4; type mem_op_read;
~ [No Crypt] addr: 80c3bcc; size: 4; type mem_op_read;
~ [No Crypt] addr: 80c300c; size: 4; type mem_op_read;
~ [No Crypt] addr: 80c3014; size: 4; type mem_op_read;
~ [No Crypt] addr: 8260a12; size: 8; type mem_op_read;
~ [No Crypt] addr: 8260a1a; size: 8; type mem_op_read;
~ [No Crypt] addr: 8260a22; size: 8; type mem_op_read;
~ [No Crypt] addr: 8260a2a; size: 8; type mem_op_read;
~ [No Crypt] addr: 8260a32; size: 8; type mem_op_read;
~ [No Crypt] addr: 8260a3a; size: 8; type mem_op_read;
~ [No Crypt] addr: 8260a42; size: 4; type mem_op_read;
~ [No Crypt] addr: 8260a46; size: 2; type mem_op_read;
~ [No Crypt] addr: 80e7058; size: 8; type mem_op_write;
```

too much to show

siehe file:
boot_success_s0x84F0992_addr_output_smaller_cleaned.txt

Adressbereich: 0x80c0000 - 0x82a3ffc

Weitere Adressen auf die zugegriffen wird ohne vorheriges WRITE:

```
~ [No Crypt] addr: 2f004810; size: 4; type mem_op_read;
~ [No Crypt] addr: 2f004818; size: 4; type mem_op_read;
~ [No Crypt] addr: 2fffbc00; size: 4; type mem_op_read;
~ [No Crypt] addr: 2f004820; size: 4; type mem_op_read;
~ [No Crypt] addr: 2f004808; size: 4; type mem_op_read;
~ [No Crypt] addr: 2f004828; size: 4; type mem_op_read;
~ [No Crypt] addr: 2fffdfc0; size: 4; type mem_op_read;
~ [No Crypt] addr: 3fa4aa88; size: 4; type mem_op_read;
~ [No Crypt] addr: 3fa4aba0; size: 4; type mem_op_read;
~ [No Crypt] addr: 3fa4a918; size: 4; type mem_op_read;
~ [No Crypt] addr: 3fa4a8f8; size: 4; type mem_op_read;
~ [No Crypt] addr: 3fa4a8ec; size: 4; type mem_op_read;
~ [No Crypt] addr: 3fa4a958; size: 4; type mem_op_read;
~ [No Crypt] addr: 3fa4abf0; size: 4; type mem_op_read;
```

too much to show

siehe file: boot_success_b0x2f000000_addr_output_bigger_cleaned.txt

Adressbereich: 0x2f004808 - 0x3fa90e98

Idee

Vorerst wird der Bereich von **0x0 bis 0x84F0992** nicht verschlüsselt. Größe: 139.397 MB

-> der Bereich ist teilweise sehr bootrelevan.

Eine Verschlüsselung des Bootloaders bzw. des BoadSetup's führt zu "*undefined behaviour*" der VCPU.

Während dem Boot-Prozess, Daten im Bereich **0x2f004808 - 0x3fa90e98** verschlüsseln**.**
Die Größe des Bereichs ist mit 279.496 MB zu groß um dauerhaft nicht zu verschlüsseln!

Versuch ohne eine solche Verschlüsselung im genannten Bereich führt zu:

```
---
qemu-system-aarch64: Trying to execute code outside RAM or ROM at 0x00000000ffff0010
This usually means one of the following happened:

(1) You told QEMU to execute a kernel for the wrong machine type, and it crashed on startup (eg trying
to run a raspberry pi kernel on a versatilepb QEMU machine)
(2) You didn't give QEMU a kernel or BIOS filename at all, and QEMU executed a ROM full of no-op
instructions until it fell off the end
(3) Your guest kernel has a bug and crashed by jumping off into nowhere

This is almost always one of the first two, so check your command line and that you are using the
right type of kernel for this machine.
If you think option (3) is likely then you can try debugging your guest with the -d debug options; in
particular -d guest_errors will cause the log to include a dump of the guest register state at this
point.

Execution cannot continue; stopping here
---
```

Versuch Verschlüsselung

Der Bootvorgang passiert wie folgt:

Viele Instruktionen werden vom Adressbereich < 0x84F0992 geladen.
 	
Sobald eine Instuktion vom Adressbereich > 0x2f000000 (z.B) geladen wird, wird die Funktion:
 	
exec.c
crypt_boot_data()
via
xilinx.mem_enc.c
apply_crypt()
 	
aufgerufen.
 	
Diese encrypted gewählten Bereich, nachweislich und hoffentlich richtig gestetet im file "relevant_data_for_boot" ersichtlich.
	~ [Crypt] addr: 700c; size: 4; type mem_op_read;
	~ [Crypt] addr: 700c; size: 4; type mem_op_read;
	~ [Crypt] addr: 700c; size: 4; type mem_op_read;
	~ [Crypt] addr: 700c; size: 4; type mem_op_read;
	~ [Crypt] addr: 7004; size: 4; type mem_op_read;
	~ [Crypt] addr: 7c20; size: 4; type mem_op_read;
	--- break point! we need to encrypt the data here
	~ [Crypt] addr: 2f004810; size: 4; type mem_op_read;
	~ [Crypt] addr: 7c20; size: 4; type mem_op_read;
	~ [Crypt] addr: 2f004810; size: 4; type mem_op_read;
	~ [Crypt] addr: 7c20; size: 4; type mem_op_read;
	~ [Crypt] addr: 2f004810; size: 4; type mem_op_read;
	
	too much to show
	
	siehe file: #see_diff_no_crypt_address_log_sequence.txt

Dies führt darauffolgend zu einem falschen Verhalten der CPU.
Es wird eine falsche Adresse im ext_ram gelesen. Vergleich:

w/o encryption:	**w encryption:**
file: #see_diff_no_crypt_address_log_sequence.txt	#see_diff_with_crypt_address_log_sequence.txt
~ [Crypt] addr: 7024; size: 4; type mem_op_read;	~ [Crypt] addr: 7024; size: 4; type mem_op_read;
~ [Crypt] addr: 700c; size: 4; type mem_op_read;	~ [Crypt] addr: 700c; size: 4; type mem_op_read;
~ [Crypt] addr: 700c; size: 4; type mem_op_read;	~ [Crypt] addr: 700c; size: 4; type mem_op_read;
~ [Crypt] addr: 700c; size: 4; type mem_op_read;	~ [Crypt] addr: 700c; size: 4; type mem_op_read;
~ [Crypt] addr: 700c; size: 4; type mem_op_read;	~ [Crypt] addr: 700c; size: 4; type mem_op_read;
~ [Crypt] addr: 7004; size: 4; type mem_op_read;	~ [Crypt] addr: 7004; size: 4; type mem_op_read;
~ [Crypt] addr: 7c20; size: 4; type mem_op_read;	~ [Crypt] addr: 7c20; size: 4; type mem_op_read;
→ ident bis zu dem punkt	*→ ident bis zu dem punkt, hier findet di encryption statt*
~ [Crypt] addr: 2f004810; size: 4; type mem_op_read;	~ [Crypt] addr: 2f004810; size: 4; type mem_op_read;
~ [Crypt] addr: 7c20; size: 4; type mem_op_read;	~ [Crypt] addr: 700c; size: 4; type mem_op_read;
→ korrekte adresse	*→ falsche adresse wird ausgelesen*
~ [Crypt] addr: 2f004810; size: 4; type mem_op_read;	~ [Crypt] addr: 5d94; size: 4; type mem_op_read;
~ [Crypt] addr: 7c20; size: 4; type mem_op_read;	~ [Crypt] addr: 7ffc; size: 4; type mem_op_read;
~ [Crypt] addr: 2f004810; size: 4; type mem_op_read;	~ [Crypt] addr: 2fffdfc0; size: 4; type mem_op_read;
→ start boot:	*→ start einer endlosschleife:*
~ [Crypt] addr: 7004; size: 4; type mem_op_read;	~ [Crypt] addr: 4020; size: 4; type mem_op_read;
~ [Crypt] addr: 7004; size: 4; type mem_op_read;	~ [Crypt] addr: 4020; size: 4; type mem_op_read;
~ [Crypt] addr: 7024; size: 4; type mem_op_read;	~ [Crypt] addr: 4020; size: 4; type mem_op_read;
~ [Crypt] addr: 7024; size: 4; type mem_op_read;	~ [Crypt] addr: 4020; size: 4; type mem_op_read;
~ [Crypt] addr: 7004; size: 4; type mem_op_read;	~ [Crypt] addr: 4020; size: 4; type mem_op_read;
~ [Crypt] addr: 700c; size: 4; type mem_op_read;	~ [Crypt] addr: 4020; size: 4; type mem_op_read;
~ [Crypt] addr: 7004; size: 4; type mem_op_read;	~ [Crypt] addr: 4020; size: 4; type mem_op_read;
~ [Crypt] addr: 7024; size: 4; type mem_op_read;	

Vermutung dass die Encryption wichtige Daten überschreibt hat sich nicht bestätigt;
Es wurde nichts gefunden was zu dem Adressbereich gehört und nicht überschrieben werden darf;

_Debug:
start mit script

```
./buildqemu zedqemudebug crypt_boot_data
(gdb) break apply_crypt
(gdb) p/x addr
```

unencrypted oder encrypted mit key = 0

```
0x7c20
```

encrypted mit wirklichem key:

```
0x700c
```

zurückverfolgt bis zur funktion:
helper.c
get_phys_addr_v6()

Diese holt sich eine physikalische Adresse (sog. “table”) von einer übergebenen virtuellen Adresse.
Die virtuelle Adresse unterscheidet sich wie folgt:

unencrypted oder encrypted mit key = 0
`0xf0804a1c`
else
`0xc0362508`

→ dadurch wird die “table” also die hwaddr falsch an
arm_ldl_ptw() → address_space_ldl_le() → unsere mem ops addr..

Warum passiert das?
Diese virtuelle Adresse `0xf0804a1c wird aus einem cpu register ausgelesen siehe:`

_Debug:
start mit script

```
./buildqemu zedqemudebug crypt_boot_data
(gdb) break cpu_get_tb_cpu_state
```

/target/arm/cpu.h
hier: `*pc = env->regs[]`

Frage ist, wie kommt eine falsche vAdresse ins VCPU Register?
Habe dazu leider keine Antwort gefunden.

Ideen & Vermutungen:

- *Versuch* Deaktivieren des TB-Cache
    - Translated Block Cache - es werden bereits übersetzte TB vom TCG (tiny code generator) gespeichert und über eine hash table bei Bedarf schnell verwendet. Dies macht QEMU so schnell.

- Es muss irgendwo einen Zugriff auf meinen Hauptspeicher geben der nicht über die mem-ops geht.
    - Würde erklären warum genau nach der Verschlüsselung das Problem auftritt OBWOHL jeder Memory Zugriff über die Mem ops und oben genannte Funktionen entschlüsselt oder verschlüsselt werden sollte.

- Direkte Umwandlung meiner virtuellen Host Adresse in eine virtuelle Guest Adresse über den TLB sollte, nach längerem überlegen nicht das Problem sein, da dadurch nur das übersetzen mit der PT wegfällt, nicht aber ein ordentlicher Memory Zugriff

- Es wurde nicht jeder Mem Zugriff im Code gefunden? :(

Information:

- Wichtige Informationen sind im Code einfach mit dem Tag "*theuema*" versehen und so leicht zu finden was verändert wurde, bzw. auch einige nützliche Kommentare.

- Es wurde auch der Code für den QEMU eigenen Bootloader und das Board Setup ausgelesen:

Bootloader:

```
$gdb x/40xib 0x7fff9ba00000
        0x7fff9ba00000:    add   $0xe0,%al
        0x7fff9ba00002:    (bad)
        0x7fff9ba00003:    loop  0x7fff9ba00009
        0x7fff9ba00005:    lock (bad)
        0x7fff9ba00007:    in    $0x0,%eax
        0x7fff9ba00009:    add   %eax,(%rax)
        0x7fff9ba0000b:    add   %al,(%rax)
        0x7fff9ba0000d:    add   %ah,-0x60effb1d(%rax)
        0x7fff9ba00013:    in    $0x4,%eax
        0x7fff9ba00015:    and   %bl,-0x600ffb1b(%rdi)
        0x7fff9ba0001b:    in    $0x32,%eax
        0x7fff9ba0001d:    or    $0x10000000,%eax
        0x7fff9ba00022:    rex.WRX or %r8b,(%rax)
        0x7fff9ba00025:    addb  $0x0,(%rax)
        0x7fff9ba00028:    add   %al,(%rax)
```

Für weitere Informationen siehe file: *booloader0x0*

Board-Setup:

```
$gdb x/40xib 0x7fff9ba00100
        0x7fff9ba00100:    clc
        0x7fff9ba00101:    add   $0xa0,%al
        0x7fff9ba00103:    jrcxz 0x7fff9ba00112
        0x7fff9ba00105:    (bad)
        0x7fff9ba00106:    or    $0x401000e3,%eax
        0x7fff9ba0010b:    jrcxz 0x7fff9ba00115
        0x7fff9ba0010d:    adc   %al,0x41008e5(%rax)
        0x7fff9ba00113:    jrcxz 0x7fff9ba00116
        0x7fff9ba00115:    adc   %al,-0x1d(%rax)
        0x7fff9ba00118:    add   %dl,(%rcx)
        0x7fff9ba0011a:    and   $0x7b,%ch
        0x7fff9ba0011d:    (bad)
        0x7fff9ba0011e:    (bad)
        0x7fff9ba0011f:    jrcxz 0x7fff9ba00121
        0x7fff9ba00121:    adc   %al,-0x1d(%rax)
        0x7fff9ba00124:    add   $0x10,%al
        0x7fff9ba00126:    and   $0x1e,%ch
        0x7fff9ba00129:    ljmp  *(%rdi)
        0x7fff9ba0012b:    loope 0x7fff9ba0012d
        0x7fff9ba0012d:    add   %al,(%rax)
```

Für weitere Informationen siehe file: *board_setup0x100*

- *buildqemu* script kann für debug, build und auführung unseres zedboards verwendet werden. Eine Anleitung zur Handhabung wird wie üblich mit `./buildqemu --help` aufgerufen

```
"-------------------------------------------------------------------"
"build/clean commands: $0 x86 | arm64 | clean"

"gdb QEMU ZEDBOARD debug commands: $0 zedqemudebug break_function_name"

"AARCH64 usermode compile commands: $0 aarch64 filename.c"

"static zedboard ivocation with QEMU console: $0 runzedboard"

"|| builds are stored in: build_x86_64_debug | build_arm_64_debug"
"|| store files to compile in ${TESTFOLDER}"
"-------------------------------------------------------------------"
```

- Wichtige veränderte Config-Dateien zum Builden von Yocto mit Meta-Xilinx Layer sind ebenfalls in der Repo *https://github.com/theuema/docs_zynq_yocto.git* zu finden.
    - Erklärung was daran geändert wurde findet man ab Seite 1;

General Learnings

A Custom Linux Distribution—Why Is It Hard?

Let's face it—building and maintaining an operating system is not a trivial task. Many different aspects of the operating system have to be taken into consideration to create a fully functional computer system:

Bootloader:

The bootloader is the first piece of software responsible for initializing the hardware, loading the operating system kernel into RAM, and then starting the kernel. The bootloader is commonly multistaged with its first stage resident in nonvolatile memory. The first stage then loads a second stage from attached storage such as flash memory, hard drives, and so on.

Kernel:

The kernel, as its name implies, is the core of an operating system. It manages the hardware resources of the system and provides hardware abstraction through its APIs to other software. The kernel's main functions are memory management, device management, and responding to system calls from application software. How these functions are implemented depends on the processor architecture as well as on peripheral devices and other hardware configuration.

Device Drivers:

Device drivers are part of the kernel. They provide application software with access to hardware devices in a structured form through kernel system calls. Through the device drivers, application software can configure, read data from, and write data to hardware devices.

Life Cycle Management:

From power on to shutdown, a computer system assumes multiple states during which it provides different sets of services to application software. Life cycle management determines what services are running in what states and in what order they need to be started to maintain a consistent operating environment. An important piece of life cycle management is also power management, putting a system into energy saving modes when full functionality is not required, and resuming fully operational mode when requested.

Application Software Management:

Application software and libraries make up the majority of software installed on a typical system, providing the end-user functionality. Frequently, many hundreds to multiple thousands of software packages are necessary for a fully operational system

Yocto Project

The Yocto Project has the aim and objective of attempting to improve the lives of developers of customised Linux systems supporting the ARM, MIPS, PowerPC and x86/x86 64architectures. A key part of this is an open source build system, based around the OpenEmbedded architecture, that enables developers to create their own Linux distributionspecific to their environment. This reference implementation of OpenEmbedded is called Poky.

This is where the strengths of the Yocto Project lie.
It combines the best of both worlds by providing you with a complete tool set and blueprints to create your own Linux distribution from scratch starting with source code downloads from the upstream projects. The blueprints for various systems that ship with the Yocto Project tools let you build complete operating system stacks within a few hours. You can choose from blueprints that build a target system image for a basic system with command-line login, a system with a graphical user interface for a mobile device, a system that is Linux Standard Base compliant, and many more. You can use these blueprints as a starting point for your own distribution and modify them by adding and/or removing software packages.