Documentation

Rust and CakeML Communication via the Prosper Hypervisor

July 29, 2023

System Specification, Motivation and Attacker Model

The system consists of three components: The Prosper hypervisor and two guests. The hypervisor has been formally verified to isolate the CPU to only depend on and affect the guest that the CPU is executing. The hypervisor configures the page tables such that in unprivileged mode, the CPU can only access the memory allocated to the guest the CPU is currently executing. The page tables are located in Linux memory but are configured such that Linux cannot modify the page tables. The guests can only communicate via channel provided by the hypervisor based on input and output buffers of the guests.

In addition to the CPU, the network interface controller (NIC) can also access memory. The page tables are configured such that Linux can only read DMA registers, and if Linux attempts to write them, an exception is raised, causing the hypervisor to be invoked, which invokes a NIC monitor. The monitor checks that the attempted DMA register write cannot cause the NIC to access non-Linux memory nor modify the page tables. Regarding formal verification of NIC isolation, isolating conditions of the NIC have been formally verified, and that the transmission part of the monitor respects the verified isolation conditions.

By letting Linux be one guest, and a critical application be the other guest (e.g. CakeML), this system allows a feature rich untrusted software component (e.g. Linux) to interact in a controlled manner with a critical guest, without corrupting that guest. For instance, Linux can provide Internet access via its TCP/IP stack and Ethernet driver, and USB storage via its file system and USB infrastructure, which is probably many tens of thousands of lines of code, without enabling this code to directly affect the critical application.

The attacker model is thus a Linux guest that is completely untrusted and upon which no assumptions are made.

Software Interaction

The Prosper hypervisor provides isolation between itself and two guests. One guest is a paravirtualized (modified/ported) Linux 5.15.13, and the other is a CakeML application. A rust application is running in Linux, and communicates with the CakeML guest via a communication channel provided by the hypervisor.

A step-based description of the guest communication is as follows (see Figure 1):

1. Rust allocates "dynamic" input and output buffers, and stores data in the output buffer.

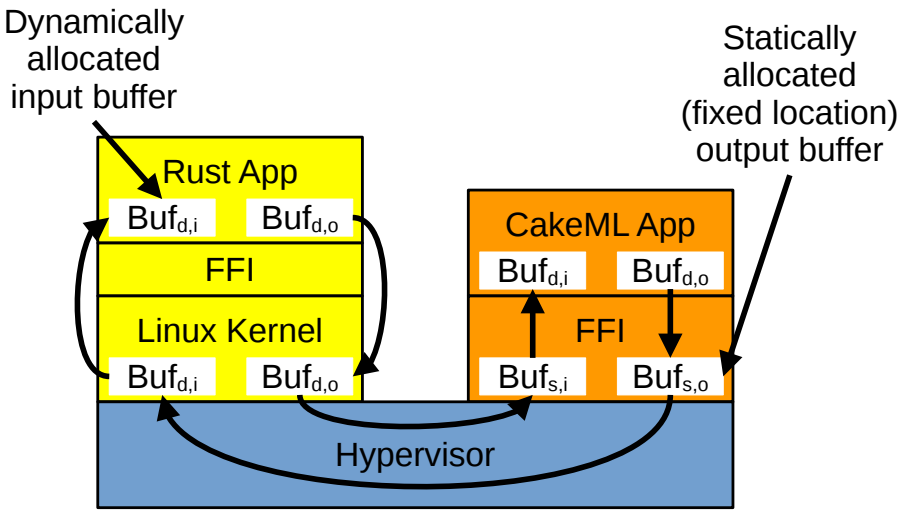

Figure 1: Rust and CakeML interaction through the hypervisor.

2. Rust invokes the FFI, which issues a system call to Linux, specifying the locations of the buffers and their size.

3. The Linux system call allocates kernel buffers, and copies the data in the dynamic Rust output buffer to a Linux kernel output buffer, and invokes the hypervisor.

4. The hypervisor copies the data from the Linux kernel output buffer to a fixed located input buffer of CakeML, and transfers control to CakeML.

5. The CakeML FFI transfers the data in the fixed CakeML buffer to a dynamically allocated CakeML buffer in the CakeML application.

6. CakeML processes the received data and stores the result in a dynamically allocated output buffer and invokes the FFI.

7. The CakeML FFI transfers the data in the dynamically allocated output buffer to the fixed located output buffer and invokes the hypervisor.

8. The hypervisor copies the data in the fixed located output buffer of CakeML to the input buffer of the Linux kernel, and transfers control to the Linux system call.

9. The Linux system call copies the data from the kernel input buffer to the Rust input buffer, and returns to Rust.

10. Rust processes the received data, and the communication cycle goes back to step 1.

The Rust application invokes the hypervisor by means of a foreign function interface written in C:

invoke_cake(*output_buffer*, *output_buffer_size*, *input_buffer*, *input_buffer_size*).

This function takes addresses and sizes of input and output buffers of the Rust application, and performs a system call to Linux. The system call in Linux first allocates Linux kernel input and output buffers, then copies the data of the Rust application output buffer to the Linux kernel output buffer, and third invokes a hypercall of the hypervisor with the physical addresses and sizes as arguments.

The reason for having Linux kernel buffers in addition to the Rust application buffers is to enable the hypervisor to identify the physical addresses of the buffers (using a simple virtual address offset `0xC0000000` to convert virtual addresses to physical addresses and vice versa), in contrast to Linux application buffers that can be at arbitrary locations in memory.

The hypervisor checks that the identified Linux kernel buffer locations are allocated to Linux (and not the hypervisor or CakeML, to preserve isolation; see hypervisor//core//hypervisor//handlers.c). Otherwise, the communication is blocked (an error message is printed and the system freezes). If the buffers are in Linux allocated memory, the hypervisor copies the data in the Linux kernel output buffer to the CakeML input buffer (at the fixed virtual address `0xF0F00000` allocated to the CakeML guest; see hypervisor/core/hw/board/beaglebone/board_mem.c). The hypervisor then invokes CakeML (whose program counter is initialized in hypervisor/core/hypervisor/guest_config/linux_config.c by the macro `TRUSTED_RPC`; see hypervisor-cakeml//cakeml//start.S, where `TRUSTED_RPC` is set to the 6th 4-byte word because the first instruction in start.S is a nop being skipped and the following 5 words are for the size of the buffers, at the moment 4 bytes times 5 words = 20 bytes).

CakeML continues (or starts) its execution, which (if everything goes to plan) copies the data via a foreign function interface from the fixed input buffer to a buffer dynamically allocated by CakeML. CakeML then processes the data and then via another foreign function interface copies the data of dynamically allocated buffer of CakeML to a fixed output buffer location, and invokes the hypervisor. The hypervisor copies the data in the fixed output buffer location to the Linux kernel input buffer, and invokes Linux.

The Linux system call copies the data in the Linux kernel input buffer to the Rust application input buffer, and returns to the Rust application.

Hypervisor Changes

The Rust application triggered execution of some Linux kernel code that causes alignment faults (addressing a memory location whose address is not a multiple of 4). This type of fault causes a data abort exception. Such exceptions were analyzed by the previous version of the hypervisor to check whether they were caused by Linux attempting to write an executable page (the DMMU component of the hypervisor enforces the W XOR X policy to ensure that only trusted software is executed; however, at the moment there is no mechanism checking that executable pages satisfy a certain trusted property). This analysis ignored the possibility of alignment faults (which have nothing to do with access permission violations), and the system crashed. The updated version of the hypervisor forwards data abort exceptions due to alignment faults directly to Linux, which handles them as intended.

Linux Changes

The Rust application is linked by the Rust compiler with precompiled libraries that contain thumb code. This code caused undfined instruction exceptions, which the hypervisor forwards to Linux. This type of exception caused the exception handler of the previous paravirtualized Linux kernel to access uninitialized data structures (since they are not intended to be used by Linux when executed on top of the hypervisor) and perform privileged operations (which can only be performed by the hypervisor, and which in this case are not supported). The Linux code associated with these operations is excluded from compilation by means of a macro.

Compilation for BBB

The following summarizes the compilation steps for running the software on BeagleBone Black. Detailed instructions are listed in the text file 'Installation and run instructions.txt'.

1. Compile the rust application statically (in order to avoid dependence on dynamically linked libraries; invoke_cakeml.c implements the foreign function interface):

   cd <path to cakeml-kth-repo>/cakeml-kth-repo/hypervisor-cakeml/linuxapp/

   <path to buildroot-2021.02.8>/buildroot-2021.02.8/output/host/usr/bin/arm-buildroot-linux-gnueabihf-gcc-8.4.0 -static -c invoke_cakeml.c -o invoke_cakeml.o

<path to buildroot-2021.02.8>/buildroot-2021.02.8/output/host/usr/bin/arm-buildroot-linux-gnueabihf-ar rcs libinvoke_cakeml.a invoke_cakeml.o

<path to buildroot-2021.02.8>/buildroot-2021.02.8/output/host/usr/bin/rustc –target=armv7-unknown-linux-gnueabihf -Clinker=arm-linux-gnueabihf-gcc -Ctarget-feature=+crt-static -L . rust_app.rs -o rust_app

2. Update the filesystem used by Linux by placing the compiled rust application, rust_app, in the root (home directory of the user root) directory:

   cd <path to buildroot-2021.02.8>/buildroot-2021.02.8/output/images

   sudo rm -rf rootfs

   mkdir rootfs

   cd rootfs

   sudo cpio -i -d -H newc -F ../rootfs.cpio –no-absolute-filenames

   cd ..

   sudo chown root rootfs -R

   sudo cp ../../../cakeml-kth-repo/hypervisor-cakeml/linuxapp/rust_app ./rootfs/root/rust_app

3. Compile Linux and copy the binary exeutable image to the linux guest directory of the hypervisor:

   cd <path to linux-5.15.1>

   make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j8

   cd ..

   cat linux-5.15.13/arch/arm/boot/zImage linux-5.15.13/arch/arm/boot/dts/am335i boneblack.dtb >hypervisor/guests/linux/build/zImage.bin

4. Compile the CakeML guest and copy the binary executable image to trusted guest directory of the hypervisor (trusted.S is the binary CakeML code, basis_ffi.c is the implementation of the foreign function

interface, print.c contains routines for printing to the terminal for debugging, start.S contains the declaration of the fixed located input and output buffers of CakeML and the entry code of CakeML):

<path to CakeML compiler>/cake –target=arm7 ¡trusted.cml ¿trusted.S -B

arm-none-eabi-gcc -nostdlib trusted.S basis_ffi.c print.c start.S -T cake_guest.ld -o trusted

arm-none-eabi-objcopy trusted -O binary trusted.bin

cp trusted.bin <path to hypervisor>/hypervisor/guests/trusted/build/trusted.bin

5. Start docker and compile the hypervisor:

   cd <path to source directory of the hypervisor>

   sudo docker run –rm -v "$PWD":/usr/src/myapp -w /usr/src/myapp -it gcc49-arm-docker

   make

6. Run on BeagleBone Black.

Compilation for x86

Invoke the script compile_hypervisor_emulation.sh. It first compiles the CakeML application by defining the macro `EMULATE_HYPERVISOR` (used for basis_ffi.c). It then compiles the Rust application by creating a C library invoke_cakeml.c by defining the macro `EMULATE_HYPERVISOR`, and then the Rust code with the Rust configuration macro `emulate_hypervisor`. Finally it compiles emulation.c, which sets up the pipes and creates a child process, with the parent being replaced with the CakeML application and the child being replaces by the Rust application.